Universität Trier

Fachbereich II: Sprach-, Literatur- und Medienwissenschaften

Masterarbeit

im Studiengang Digital Humanities

zur Erlangung des akademischen Grades
Master of Science

Thema:	Volltext vs. abgeleitetes Textformat: Systematische Evaluation der Performanz von Topic Modeling bei unterschiedlichen Textformaten mit Python
Autor:	Martin Kocula, s2makocu@uni-trier.de MatNr. 1042695
Version vom:	17. August 2021
Betreuer:	Prof. Dr. Christof Schöch
Zweitprüfer:	Prof. Dr. Achim Rettinger

Inhaltsverzeichnis

Abbildungsverzeichnis iv

Tabellenverzeichnis v

Code-Verzeichnis vi

Abkürzungsverzeichnis vii

1 Einleitung 1

2 Topic Modeling und Evaluationsmethoden 5

2.1 Latent Dirichlet Allocation . . . 5
2.2 Bag of Words . . . 8
2.3 Vorverarbeitungsschritte . . . 9
2.4 Bewertung der Topics . . . 11
2.4.1 Menschliche Interpretierbarkeit . . . 11
2.4.2 Unifying Coherence Framework . . . 12
2.4.3 MALLETs Diagnostik-XML . . . 15

3 Die Formate 17

3.1 Term-Dokument-Matrix . . . 17
3.2 Segmentweise Aufhebung der Sequenzinformation . . . 19
3.3 Selektiv reduzierte Information über einzelne Tokens . . . 20
3.4 N-Gramme . . . 21

4 Der Workflow 24

4.1 Das Korpus . . . 25
4.1.1 Quellen . . . 25
4.1.2 Vorverarbeitung . . . 28
4.2 Tagging . . . 31
4.3 Segmenting . . . 34
4.4 Formatting . . . 35
4.4.1 Die Ordner- und Kontrollstrukturen . . . 37
4.4.2 Parameter . . . 38
4.4.3 Selektiv reduzierte Information über einzelne Tokens . . . 39
4.4.4 Segmentweise Aufhebung der Sequenzinformation . . . 41
4.4.5 Term-Dokument-Matrix . . . 42
4.4.6 N-Gramme . . . 42

4.5 Preprocessing 44
4.5.1 Das Original und Selektiv reduzierte Information über einzelne Token 45
4.5.2 Term-Dokument-Matrix 46
4.5.3 N-Gramme 47
4.5.4 Generelles Preprocessing 48
4.6 Topic Modeling 50
4.6.1 Topic Modeling mit Gensim 51
4.6.2 Topic Modeling mit MALLET 55
4.6.3 Palmetto 57
4.6.4 Diagnostik-XML auslesen 60

5 Evaluation 61
5.1 Generelle Analyse der vorverarbeiteten Texte 61
5.2 Evaluierung der Tools MALLET und Gensim 62
5.3 Hypothesenprüfung 64
5.3.1 Der Signifikanztest 64
5.3.2 Das Konfidenzintervall 68
5.4 Datenauswertung 69
5.4.1 Original 69
5.4.2 N-Gramme 71
5.4.3 Vergleich zwischen Original und einfacher Term-Dokument-Matrix 71
5.4.4 Vergleich zwischen Original und segmentweiser Aufhebung der Sequenzinformation 75
5.4.5 Vergleich zwischen Original und selektiv reduzierten Informationen über Tokens 77
5.4.6 Vergleich aller Formate auf Topic-Kohärenz-Ebene 78
5.4.7 Diagnostik-XML 81

6 Fazit 86

Literatur 88

Anhang 93

Eidesstattliche Erklärung 96

Abbildungsverzeichnis

1 Generierung von Topics mit LDA 6
2 Beispiel für Topic Intrusion 11
3 Überblick über das Unifying Coherence Framework 13
4 Nützlichkeit von Textformaten (Quelle: Schöch et al., 2020) 17
5 Extraktion des Korpus TEI 26
6 Der Tagging-Prozess 31
7 Der Segmenting-Prozess 34
8 Der Formatting-Prozess 36
9 Das Preprocessing 45
10 Topic und Coherence Modeling 51
11 Kohärenzverlauf in Bezug auf Topicanzahl. Erzeugt von Gensim und MALLET. 63
12 Histogramm mit KDE-Graph für Durchschnittswerte von Topics, generiert mit `seaborn` und `matplotlib` 66
13 Histogramm mit KDE-Plot für 20 Originalformate mit je 20 Topics . 69
14 KDE-Plots aller TDM- und Original-Modelle mit Konfidenzintervallgrenzen 72
15 KDE-Plots aller SAS- und Original-Modelle mit Konfidenzintervallgrenzen 75
16 KDE-Plots aller TKN- und Original-Modelle mit Konfidenzintervallgrenzen 77
17 Box-Plots aller Formate im Vergleich 79
18 Box-Plots zum Attribut `document_entropy` aus den Diagnostik-XMLs 82
19 Box-Plots zum Attribut `uniform_dist` aus den Diagnostik-XMLs . . 83
20 Box-Plots zum Attribut `corpus_dist` aus den Diagnostik-XMLs . . . 84

Tabellenverzeichnis

1 Ausschnitt aus einer TDM 18
2 N-Gramm-Matrix mit N-Gramm-Länge 3 bestehend aus lemmatisierten Substantiven, Verben und Adjektiven. Erstellt auf Grundlage des Gesamkorpus 23
3 Beispiel für einen Text mit POS und Lemma für jedes seiner Token . 32
4 Beispiel für eine Palmetto-Output-Datei 59
5 Top 3 Topics des Originals 70
6 Worst 3 Topics des Originals 71
7 Segmentgröße und damit verbundene Segmentanzahl 72
8 Top 3 Topics der Term-Dokument-Matrix 74
9 Worst 3 Topics der Term-Dokument-Matrix 74
10 Top 3 Topics der segmentweisen Aufhebung von Segmentinformationen 76
11 Worst 3 Topics der segmentweisen Aufhebung von Segmentinformationen 76
12 Top 3 Topics der segmentweisen Aufhebung von Segmentinformationen 78
13 Worst 3 Topics der segmentweisen Aufhebung von Segmentinformationen 78
14 Extrema und Durchschnittswerte der Dokumententropien 82
15 Extrema und Durchschnittswerte der Uniformdistanzen 83
16 Extrema und Durchschnittswerte der Uniformdistanzen 85
17 Penn-Tagset 93

Code-Verzeichnis

1 Auszug aus einer von MALLET erstellten Diagnostik-XML. 15
2 Ausschnitt eines Textsegements bestehend aus Features mit Zufallssequenz, erstellt aus den Featureset von `acd01.txt` 20
3 Ausschnitt eines Textes, in dem nur Substantive, Adjektive und Verben erkenntlich sind. Erstellt aus den Featureset von `acd01.txt` . . . 21
4 Befüllung einer Date-Property 29
5 TEI-Wurzelelement mit Namespace 29
6 Erstellung eines Dataframes 29
7 Extraktion von Textknoten 30
8 Auszug aus der Tagged-File 33
9 Aufruf des Treetaggers 33
10 Der Segmentierungsprozess. 35
11 Hilfsfunktion zur Prüfung von Segmentdateien 38
12 Feature Splitting 40
13 Randomisierung der Segmente 41
14 Aggregierung der Token für TDM 42
15 Aggregierung der Token für N-Gramme 43
16 Aggregierung der Token für N-Gramme auf Korpusebene aus Einzeltexten 43
17 Einlesen einer Term-Dokument-Matrix 46
18 Vervielfältigung der Features aus einer Term-Dokument-Matrix 46
19 Einlesen einer N-Gramm-Datei 47
20 Generierung eines Textes aus N-Grammen 48
21 Filterung nach Wortarten und anschließende Lemmatisierung 49
22 Funktion zur Entfernung von Stoppwörtern 50
23 Erstellung eines MALLET Topic Models mit Gensim Wrapper. Auszug aus `topic_modeling.py`. 54
24 Erstellung des Import-Befehls für MALLETs Kommandozeile 55
25 Erstellung des Befehls für die TM-Erstellung für MALLETs Kommandozeile 56
26 Extraktion von Topics und ihren Top-Wörtern aus einer MALLET-Keys-CSV. Ausschnitt aus `palmetto_coherence.py`. 57
27 Berechnung der Kohärenz mit Palmetto 58
28 Einlesen der Diagnostik-XML in einen Dataframe 60
29 Erstellung eines Histogramms mit seaborn 66
30 Daten-Fitting 67
31 Aufruf der ttest-Funktion 68
32 Konfidenzintervallberechnung 68
33 Erstellung von kategorisierten Box-Plots 79

Abkürzungsverzeichnis

BOW ..... Bag of Words

CL ..... Computerlinguistik

COST ..... European Cooperation in Science Technology

CSV ..... Comma-separated values

DH ..... Digital Humanities

DNS ..... Domain Name System

ELTeC ..... European Literary Text Collection

HTML ..... Hypertext Markup Language

JAR ..... Java Archive

KDE ..... Kernel Density Estimate

LDA ..... Latent Dirichlet Allocation

LSA ..... Latent Semantic Analysis

LSI ..... Latent Semantic Indexing

MALLET ..... Machine Learning For Language Toolkit

OCR ..... Optical Character Recognition

OTA ..... Oxford Text Archive

PLSA ..... Probabilistic Latent Semantic Analysis

POS ..... Part Of Speech

SAS ..... Segmentweise Aufhebung von Segmentinformationen

TDM ..... Term-Dokument-Matrix

TEI ..... Text Encoding Initiative

TKN ..... Selektiv reduzierte Information über Token

TM ..... Topic Modeling

URI ..... Uniform Resource Identifier

URL ..... Uniform Resource Locator

vDHd2021 ..... Digital Humanities im deutschsprachigen Raum

XML ..... Extensible Markup Language

Z. ..... Zeile

1 Einleitung

> Kopieren ist eine ganz wichtige Kulturtechnik. Ohne die kommen wir im Leben nicht aus. Kopiert wurde immer schon.
>
> Internationale Konferenz am Zentrum für Interdisziplinäre Forschung 2015
>
> *Reinold Schmücker*

Vervielfältigung und Bereitstellung von Textmaterial ist speziell in Forschungsfeldern der Linguistik, Geistes- und Sozialwissenschaften essenziell. Doch zum Schutze geistigen Eigentums von Autorinnen[1] existieren Urheberrechte, die Forschende vor große Hürden stellen.

So behindern in der Computerlinguistik (CL) und den Digital Humanities (DH) Komplikationen mit Urheberrechten nicht nur die Forschungspublikation, sondern auch die Textmaterialbeschaffung moderner Texte an sich ist signifikant erschwert, weil Forschende zusätzliche Bemühungen anstellen müssen, um sie legal von Archiven, Verlagen und Autorinnen erhalten und nutzen zu dürfen. Der Begriff *modern* ist durchaus relativ, denn das Urheberrecht einer Autorin erlischt erst 70 Jahre nach deren Tod. Da z.B. Thomas Manns frühestes Werk *Buddenbrooks* im Jahre 1901 veröffentlicht wurde und der Autor im August 1955 verstarb, ist dieses 120 Jahre alte Werk bis 2026[2] urheberrechtlich geschützt:

> Es öffnet sich um 1800, weil für Materialien vor dieser Zeit die technischen Herausforderungen im Bereich Optical Character Recognition (OCR) und Normalisierung von orthographischer Varianz immer noch so groß sind, dass deutlich weniger umfangreiche beziehungsweise qualitativ weniger hochwertige Textsammlungen zur Verfügung stehen als für die Zeit nach 1800. Und es schließt sich um 1920, weil für Texte, die später erschienen sind, in sehr vielen Fällen (...) das Urheberrecht nach wie vor greift und sowohl das Erstellen als auch das Teilen von Textsammlungen mit Dritten damit deutlich erschwert sind. (Schöch et al., 2020, Abs. 1)

Dass eine Textsammlung nicht ohne Weiteres geteilt werden kann, erschwert nicht nur für Forschende den Zugang zu den benötigten Daten, sondern auch für potenzielle Anschlussforschende, die nicht vollumfänglich auf die Quelldaten zugreifen

[1] Der Autor bemüht sich in dieser Arbeit um geschlechtsneutrale Formulierungen. In den Fällen, in denen es sich nicht vermeiden lässt, wird stattdessen das generische Femininum verwendet.

[2] Angabe auf der Webseite von Projekt Gutenberg https://www.projekt-gutenberg.org/autoren/namen/mannth.html.

können und sich diese ebenfalls mit müßigen Beschaffungsprozessen auseinandersetzen müssen, was den Fokus auf die tatsächliche Forschungsarbeit erheblich stören kann.

Schöch et al. (2020)[3] schlagen daher eine pragmatischen Lösung für urheberrechtlich geschützte Textsammlungen vor: abgeleitete Textformate. Denn besonders im Bereich der linguistischen Datenverarbeitung haben sich Methoden des Text-Data-Minings (TDM) etabliert, die Texte in ihrer ursprünglichen (und damit geschützten) Form nicht nur nicht benötigen, sondern Alternativformate sogar bevorzugen.

In Schöch et al. (2020) Abs. 27 werden die typischen TDM-Verfahren aufgeführt:

- Die Klassifikation und Clustering von Texten, u. a. für die *Autorschaftsattribution*
- Die Extraktion distinktiver Merkmale
- Die semantische Analyse mit *Topic Modeling*
- Die Analyse von Polarität[4] mit *Sentimentanalyse*
- Der Blick auf Figurenbeziehungen mit der *Netzwerkanalyse*
- Die Analyse von Beziehungen zwischen Texten, beispielsweise beim *Text-Re-Use*
- Sowie allgemein der Einsatz von Sprachmodellen für verschiedenste Aufgaben

Ein abgeleitetes Textformat ist eine Entfremdung oder Umgestaltung des Originals, sodass es nicht mehr unter das Urheberrecht fällt. In Schöch et al. (2020) Abs. 43 werden mehrere (nicht zwangsläufige vollständige) Faktoren aufgeführt: die Menge an zusammenhängendem Text liegt unter einer bestimmten Schwelle, der Werkgenuss durch einen Menschen ist ausgeschlossen, die Wiedererkennbarkeit für Lesende gering und die Rekonstruktion des Textes ist nicht trivial oder mit Unsicherheiten behaftet.

Bei den in Schöch et al. (2020) vorgestellten Ableitungen handelt es sich um:

- einfache Term-Dokument-Matrizen

[3] Es wird der APA-Zitierstil verwendet.
[4] Gemeint sind Grundemotionen, die in einem Text vorherrschend sind: negativ, neutral, positiv.

- Segmentweise aufgehobene Sequenzinformationen
- Selektiv reduzierte Information über Tokens
- N-Gramme auf Korpus- und Teilkorpus-Ebene
- einfache Wordembeddings
- kontextualisierte Embeddings

Es wird erwartet, dass jedes Format durch den Sequenzinformationsverlust in Maßen für bestimmte TDM-Verfahren ausreichend effektiv sind, weshalb sich ihre Informationen gegenseitig ergänzen. Sie ermöglichen Archiven die öffentliche Verbreitung ihrer Textdaten, ohne dass Rezipientinnen sich weiter um Lizenzen oder Abkommen kümmern müssen. Denkbar wäre die Erstellung von Formatpaketen, die für einen Text oder eine Textsammlung zur Verfügung gestellt werden und somit alle Informationen zur Verfügung zu stellen, die für ein breites Spektrum an Forschungsthemen nützlich sind.

Diese Masterarbeit beschäftigt sich mit der Frage, inwieweit sich welches Format[5] für das TDM-Verfahren *Topic Modeling* eignet. Hierzu werden diese zunächst generiert und anschließend evaluiert.

In Kapitel 1 wird erklärt, was Topic Modelle sind, wie der LDA-Algorithmus funktioniert, wie sie bewertet und wie Texte für bessere Ergebnisse vorverarbeitet werden können.

Kapitel 2 stellt die abgeleiteten Textformate vor und formuliert Hypothesen hinsichtlich ihrer Eignung für die semantische Analyse.

In Kapitel 3 wird der Workflow zur Erstellung der Formate skizziert. Es werden zudem Strategievorschläge unterbreitet, wie diese in für Topic Modeling sinnvolle Textstrukturen umgewandelt und die Modelle trainiert werden. Umgesetzt wurde dies mit der Skriptsprache *Python*, da für sie, besonders im Bereich Sprachdatenverarbeitung und Datenanalyse, viele nützliche Bibliotheken existieren. Die einzelnen Schritte werden anhand von Code-Beispielen erklärt und zahlreiche Python-Module

[5] Beide Wordembedding-Formate werden allerdings von der Untersuchung ausgeschlossen, da sie nicht im vorgegeben Rahmen zu bearbeiten sind.

und -Bibliotheken für Sprachdatenverarbeitung und Statistik vorgestellt.

Die Evaluation findet in Kapitel 4 statt, indem primär Kohärenzwerte und Topics paarweise zwischen dem Originaltext und seinen Ableitungen untereinander verglichen werden.

Die Arbeit endet mit einem Fazit, welches die Ergebnisse zusammenfasst und in einem Ausblick beschreibt, welche Fragen offen bleiben und welche Anschlussforschungen sich hieraus ergeben können.

2 Topic Modeling und Evaluationsmethoden

In den folgenden Abschnitten wird erklärt, wie Topic Modeling (TM) funktioniert, welche Tools Modelle erstellen können und wie das Korpus für gute Ergebnisse vorverarbeitet werden muss. Anschließend wird dargestellt, wie die Modelle der Textformate im Vergleich zum Originaltext bewertet werden können. Das grundlegende Verständnis von TM ist Voraussetzung für die Einschätzung der Performanz von Textformaten.

2.1 Latent Dirichlet Allocation

Unter TM versteht man das Erkennen und Extrahieren versteckter (latenter) wiederkehrender Strukturen in sonst unstrukturierten Textkorpora unter Zuhilfenahme von statistischen Verfahren und ist damit eine einschlägige Methode des Text-Data-Minings. TM basiert auf der Annahme, dass jedes Dokument aus einer festen Anzahl von Topics besteht und ein Topic sich aus einer häufigen Kookkurrenz mehrerer Wörter je Dokument ergibt, was von einem Menschen als Thema interpretiert werden kann. Diese Strukturen werden ohne menschliches Zutun bestimmt, was selbst die thematische Erschließung unbekannter, großer Textsammlungen ermöglicht (Block, 2020). Es handelt sich also bei diesem Verfahren um eine Art des *unüberwachten maschinellen Lernens*.

Das ist besonders nützlich für die Forschung an großen Korpora, deren Lektüre allein durch die schiere Größe und dem damit verbundenen Leseaufwand für einen Menschen schwierig oder nicht aufzubringen wäre. Auf diese Weise kann solch ein Korpus durch TM erschlossen werden.

Man stelle sich ein Beispiel aus dem politischen Alltag vor, wie einen Korpus bestehend aus Bundestagsreden der letzten drei Jahrzehnte. Mit TM können die Themenschwerpunkte dieser Debatten erkannt und, wenn auch die notwendigen Metadaten wie Bundestagsmitglied, Fraktionszugehörigkeit, Jahresangaben vorhanden sind, ausgewertet werden. Stellt man auf dieser Grundlage Grafiken auf, kann u.a. analysiert werden, welche Themen im Allgemeinen diskutiert werden und welche Fraktion mit seinen Mitgliedern eine bestimmte Meinung vertritt.

Aber es gibt geradezu beliebig viele Anwendungsfälle für semantische Untersuchungen: z.B. Feedback aus Fragebögen, Nachrichtenportale, Tweets, Korrespondenzen und nicht zuletzt literarische Texte in Form von Dramen, Erzählungen und Lyrik (wobei hier besonders der Fokus auf metaphorische Sprache erschwerend hinzukommt).

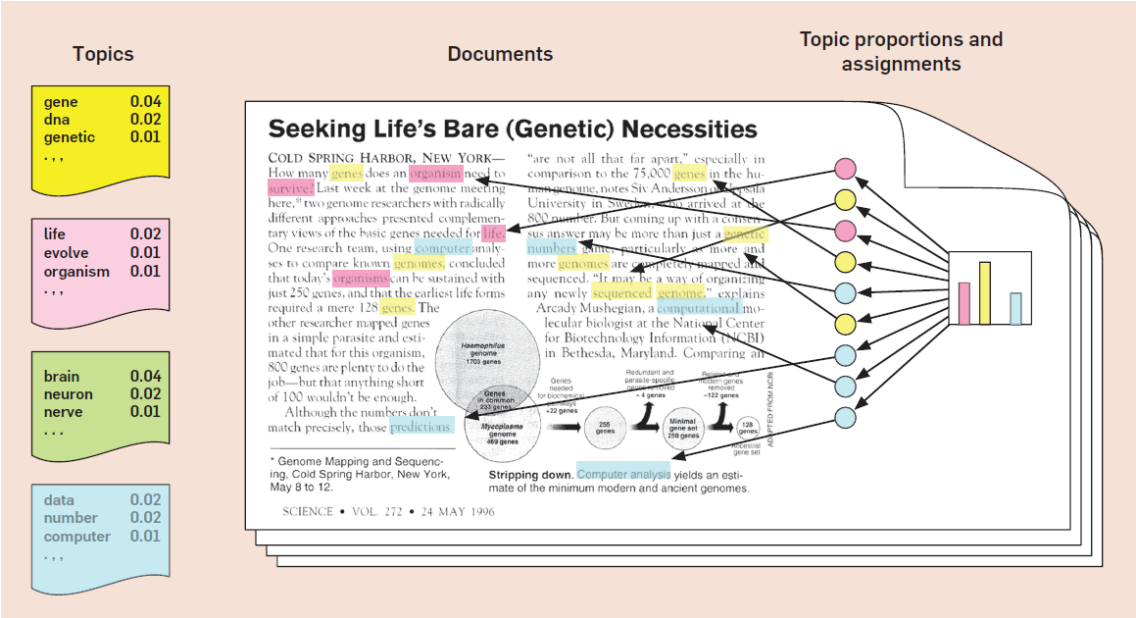


Abbildung 1: Generierung von Topics mit LDA (Ausschnitt aus Blei, 2012, S. 78)

Der von Blei et al., 2003 vorgestellte TM-Algorithmus *Latent Dirichlet Allocation* (LDA) erfreut sich in den DH (aber auch weit darüber hinaus) so großer Beliebtheit, dass LDA und TM von vielen Forschenden synonym gebraucht werden (s. z.B. Meeks & Weingart, 2012, S. 2f; vgl. Schmidt, 2012, S. 49) und damit standardmäßig verwendet wird[6]. Das liegt an der höheren erzielten Genauigkeit der generierten Topics gegenüber der Vorgängerverfahren *Latent Semantic Analysis* (LSA)[7] und *Probabilistic Latent Semantic Analysis* (PLSA), deren Ursprung in der *Information Retrieval* liegt[8].

In Abbildung 1 wird veranschaulicht, welche Annahmen LDA trifft. Es wird angenommen, dass jedes Dokument eine zufällige Mischung aus mehreren, unbekannten

[6] Weingart, 2011 schreibt, dass laut Google Scholar Bleis Paper bereits 3540 mal referenziert wurde. Nach heutigem Stand hat sich diese Zahl mit über 37361 Zitationen mehr als verzehnfacht.

[7] In der Information-Retrieval-Community taucht auch der Begriff *Latent Semantic Indexing* (LSI) auf, welcher Synonym gebraucht wird.

[8] Als Exkurs über die Unterschiede zwischen den Algorithmen empfiehlt sich https://medium.com/nanonets/topic-modeling-with-lsa-psla-lda-and-lda2vec-555ff65b0b05 und im Speziellen über die Weiterentwicklung von LSI zu LDA Blei et al., 2003.

Topics ist, deren Anzahl ebenfalls nicht bekannt ist. Jedes Topic besteht aus Wörtern, die jeweils eine bestimmte Warscheinlichkeit besitzen, diesem oder auch einem anderen Topic zugehörig zu sein. Auf der linken Seite in der Abbildung 1 sind sie unter *Topics* beispielhaft als farbige Kästen stilisiert. In diesen Kästen befinden sich Wörter, die diesem Topic zugehörig sind und die Zahl rechts neben jedem Wort stellt ihre Zugehörigkeitswahrscheinlichkeit dar. D.h. jedes Wort ist für ein Topic unterschiedlich relevant. Beispielsweise ist also das Wort *gene* im gelben Topic wahrscheinlicher als *genetic*. Es wird eine zufällige Topic-Verteilung für die Dokumente angenommen (Balkendiagramm auf der rechten Seite) und jedes Wort wird zufällig einem Topic zugeordnet (bildlich dargestellt durch den farblichen Hintergrund eines Wortes). Also besteht die Annahme darin, dass Texte aus vordefinierten Topics bestehen und jedes Dokument sich durch ihre unterschiedliche Verteilung auszeichnet.

Ruchirawat, 2020 fasst zusammen, wie der LDA-Algorithmus technisch umgesetzt wird:

1. User select K, the number of topics present, tuned to fit each dataset.

2. Go through each document, and randomly assign each word to one of K topics. From this, we have a starting point for calculating document distribution of topics $p(topic\ t|document\ d)$, proportion of words in *document d* that are assigned to *topic t*. We can also calculate topic distribution of words $p(word\ w|topic\ t)$, proportion of *word w* in all documents words that are assigned to *topic t*. These will be poor approximations due to randomness.

3. To improve approximations, we iterate through each document. For each document, go through each word and reassign a new topic, where we choose *topic t* with a probability
$p(topic\ t|document\ d)*p(word\ w|topic\ t)$ based on last round's distribution. This is essentially the probability that *topic t* generated *word w*. Recalculate $p(topic\ t|document\ d)$ and $p(word\ w|topic\ t)$ from these new assignments.

4. Keep iterating until topic/word assignments reach a steady state and no longer change much, (i.e. converge). Use final assignments to estimate topic mixtures of each document (% *words* assigned to each topic within that document) and word associated to each topic (% *times* that word is assigned to each topic overall).

In Schritt 2 wird erklärt, dass die Verteilung bei einer Wiederholung vollkommen zufällig ist und damit keine guten Topics erzielt werden können. Also ist LDA ein iterativer Prozess über Dokumente (Schritte 3 und 4), um stabile Topics zu generieren, die sich ab einer gewissen Anzahl an Iterationen nicht mehr stark verändern und je mehr Dokumente zur Verfügung stehen, desto besser kann das Ergebnis ausfallen.

Die Frage danach, wie viele Topics aus einer Dokumentsammlung ermittelt werden sollen (Schritt 1), ist problematisch, weil sie je nach Korpusgröße und inhaltlicher Kohärenz nicht klar ist. Man kann sich experimentell einer angemessenen Zahl annähern. Eine quantitative Methode besteht darin, mehrere Modelle zu erstellen, dessen Anzahl K bei jeder Iteration inkrementell um eine Schrittgröße bis zu einem im Voraus definierten Maximum erhöht wird. Wie bewertet werden kann, ob ein Modell eine geeignete Anzahl erreicht hat, wird in Kapitel 2.4 ausgeführt.

Durch den Zufallscharakter ist es unvermeidbar, dass Modelle, die auf identischer Textbasis berechnet wurden, unterschiedliche Wort- und Topicverteilungen mit sich bringen. Daher wird es notwendig sein, herauszustellen, in welchem Rahmen sich diese Schwankung bewegt, indem mehrere dieser Modelle erstellt und untereinander verglichen werden.

2.2 Bag of Words

Um die Eignung eines Formates einzuschätzen, muss noch darauf eingegangen werden, dass der LDA-Algorithmus auf dem Bag-of-Words-Modell basiert. Das Bag-of-Words-Modell nimmt an, dass die Wortreihenfolge in einem Dokument nicht wichtig für die Ermittlung des Dokumentinhalts ist (vgl. Blei et al., 2003, S. 994). Für jedes Dokument wird ein solches Bag in Form eines Vektors erstellt, der die Tokenhäufigkeit aufzählt Die Annahme ist nur dann gültig, wenn jedes Wort als Unigram angesehen wird, dass also jedes Wort für sich eine Semantik besitzt. Allerdings ergeben oft mehrere aufeinanderfolgende Wörter eine Sinneinheit (N-Gram), wie *New York*, *New York Times* oder *United States of America*, die durch Auflösung ihrer Reihenfolge auch ihre Semantik verlieren bzw. verändern. Trotzdem sind Bag-of-Words-Modelle auch unter dieser vereinfachten Annahme überraschend mächtig (vgl. Mimno, 2020-02-25). Dies wird für die Einschätzung der Textformate interessant. Einerseits ist

die Wortsequenz innerhalb eines Dokuments nicht relevant, da sie ohnehin bei der Verarbeitung keine Signifikanz hat. Betrachten wir andererseits das Format *einfache Term-Dokument-Matrix* auf Basis der Einzeltexte, sehen wir, dass dadurch der Kontext jedes Wortes verlorengeht.

2.3 Vorverarbeitungsschritte

Für möglichst kohärente Topics muss das LDA-Korpus vorbereitet werden. Ziel der Vorverarbeitung ist die Standardisierung der Informationen, Eliminierung von Störfaktoren und irrelevantem Inhalt. Die für diese Arbeit implementierte Pipeline führt die im Folgenden aufgeführten Schritte aus:

Tokenisierung. Darunter versteht man das Aufteilen des Textes in seine Bestandteile, hierzu gehören Wörter und Interpunktionen. Hier sollten vom Tagger sprachspezifische Eigenheiten, mit Bindestrichen verbundene Wörter und Abkürzungen erkannt und voneinander unterschieden werden.

Part-of-Speech-Tagging ist die Auszeichnung jedes Tokens mit seiner Wortart (Nomen, Verb, Adjektiv, u.s.w.) oder Textfunktion (z.B. Satzbegrenzungszeichen). Wenn diese Informationen vorliegen, kann gezielt nach semantisch wertvollen Wörtern gefiltert werden. Funktionswörter, wie Pronomen, Präpositionen, Konjunktionen und Artikel sind hingegen für das Topic Modelling nicht interessant, da sie semantikarme Wörter sind.

Die *Lemmatisierung* ist die Rückführung eines Wortes auf seine Grundform, z.B. vom Plural in den Singular (*mice* wird zu *mouse*) oder von einem flektierten Verb in sein Lemma (*is* wird zu *be*). Anders als beim *Stemming*, wird bei der Lemmatisierung nicht ausschließlich regelbasiert verfahren[9]. Das bedeutet, dass beim Lemmatisieren ein Lexikon genutzt wird, in dem Lemma-Einträge aufgelistet sind, die vom Programm mit dem vorgefundenen Token verglichen wird. Die Verwendung eines lemmatisierten Textes ist sinnvoll, weil damit garantiert wird, dass der LDA-Algorithmus jedes unterschiedliche Token standardisiert vorfindet und damit korrekt

[9] An dieser Stelle sei der *Porter-Stemmer*-Algorithmus genannt. Die Wortstammberechnung kommt ohne externes Lexikon zustande, was auch im Englischen sehr gute Ergebnisse erzielt. Für einen ersten Überblick über weitere Stemmer-Algorithmen s. https://www.geeksforgeeks.org/introduction-to-stemming.

aggregieren kann. Zudem können so einige Stoppwörter identifiziert und bereinigt werden.

Segmentierung. Bei Dokumentsammlungen, die große Einzeldokumente enthalten, ist es üblich sie in kleinere Dokumente zu unterteilen, da die generierten Topics bei großen Dokumenten nicht spezifisch genug sind (vgl. Sieg, 2019). Die Zeichenzahl eines Segmentes bewegt sich in der Forschung üblicherweise im drei- bis vierstelligen Bereich (s. Schöch, 2017; Weitin & Herget, 2017). Es bleibt also auszutesten, welche Segmentgröße sich für das Korpus eignet. Wichtig ist vorrangig, dass die Textformate, soweit möglich an den gleichen Token segmentiert werden, sodass sie den gleichen Inhalt tragen, wie das Original. Deswegen sollte die Segmentierung im Workflow nach der Lemmatisierung stattfinden.

Stoppwörter entfernen. Als Stoppwörter werden die semantikarmen Textbestandteile bezeichnet und werden daher auch entfernt. Über die o.g. Funktionswörter hinaus, werden auch Zahlwörter, und je nach Analyseschwerpunkt Eigennamen hinzugezählt. Man könnte argumentieren, dass es in Texten um z.B. Figuren in der Literatur oder auch reale Personen aus Zeitungstexten (Politiker, Musiker, Sportler) geht. Idealerweise würden diese als eigenes Topic erkannt und entsprechend eingeordnet werden. Schaut man sich allerdings öffentlich zugängliche Stoppwortlisten an, fällt auf, dass dort häufig oder sogar hauptsächlich Eigennamen zu finden sind..

Filterung nach Wortarten. Wenn die Vorarbeit geleistet wurde, gestaltet sich die Filterung nach Wortarten relativ einfach. So kann bestimmt werden, welche Wortarten (z.B. nur Nomen) letztendlich im vorbereiteten Korpus enthalten sein sollen. Ist die Bedeutung eines Tags bekannt, kann gezielt nach ihm gesucht werden.

Starke sprachliche Abweichungen von der eigentlich verwendeten Standardsprache stellen für all die o.g. Vorverarbeitungsschritte eine Herausforderung dar. Das gilt für Sachtexte in der Regel nicht, ist aber als sprachliches Stilmittel in fiktionalen Werken bei Figurenrede durchaus vorzufinden. Das hier verwendete Korpus enthält z.B. Rede, die Slang immitiert, darunter Sätze wie diese: ”I sez ’dang the tree!’ Us doan’t take no joy in thrawin’ en, mister.”. Es ist höchst unwahrscheinlich, dass solch ein Fall korrekt mit einschlägigen Sprachverarbeitung-Tools nach den vorgestellten Methoden verarbeitet wird. Allerdings hält sich das Vorkommen im Bezug auf das

Gesamtkorpus in solchen Maßen, dass sie in der praktischen Umsetzung vernachlässigt werden können.

2.4 Bewertung der Topics

2.4.1 Menschliche Interpretierbarkeit

Ob ein Topic *gut* oder *schlecht* ist, kann ein Mensch subjektiv zumindest anhand seiner Top-Wörter gut einschätzen. Man spricht auch von der *Kohärenz* eines Topics. Ein Topic sollte dann einen hohen Kohärenzwert haben, wenn sich die Bedeutungen seiner Top-Wörter gegenseitig unterstützen (vgl. Röder et al., 2015, S. 399).

Intuitiv könnte ein Mensch ein kohärentes und damit verständliches Topic unter einer Überschrift zusammenfassen. Beispielsweise dürfte es Lesenden relativ einfach fallen, ein Topic mit den Top-Wörtern `game, sport, ball, team` mit Labels wie *Ballsportart* oder *Fußballspiel* und das Topic `sister, father, mother, brother, child` mit *Familienmitglieder* zu versehen. Es handelt sich also um eine interpretationsbasierte Evaluation.

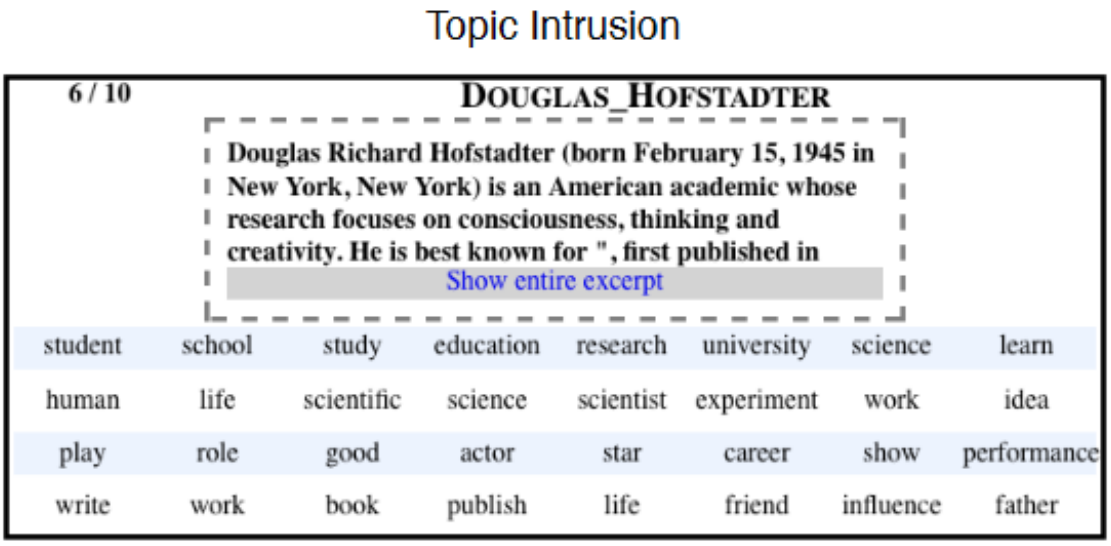


Abbildung 2: Beispiel für Topic Intrusion (Quelle: Chang et al., 2009, S. 4)

Darauf aufbauend stellen Chang et al., 2009 zwei erweiterte Methoden vor, um zu testen, ob ein Topic gut interpretierbar ist. Zum einen *Word Intrusion* auf Topic-Ebene, zum anderen *Topic Intrusion* auf Modellebene. Mit ersterem wird versucht durch Einfügen eines zufälligen Wortes die Interpretierbarkeit eines Topics zu stören.

Fügt man also dem *Familienmitglieder*-Topic ein Wort wie `apple` hinzu, werden die meisten dieses Wort auch als Störer erkennen können, da es sich semantisch (Frucht) deutlich von den Familienmitgliedern unterscheidet. Das deutet darauf hin, dass das Topic eine hohe Kohärenz hat. Ist andererseits durch das Hinzufügen nicht klar erkenntlich, was das Störwort ist, weist das Topic eine geringe Kohärenz auf. Aus dem Topic `{car, teacher, platypus, agile, blue, Zaire}` wird beispielsweise nicht klar erkenntlich, welches Wort im Nachhinein hinzugefügt wurde. Es gibt hier keinen klaren gemeinsamen semantischen Nenner zur Kategorisierung. Daher werden Menschen ein zufälliges Wort wählen und dieser Umstand impliziert eine niedrige Kohärenz. Analog wird auf Modellebene verfahren, indem den Testteilnehmenden ein kurzer Auszug aus einem Dokument mit Überschrift präsentiert werden und bewertet werden soll, welches des gegebenen Topics nicht hinzugehört. In Abbildung 2 ist ein Beispiel für Topic Intrusion zu sehen. Anhand des Einleitungstextes ist klar ersichtlich, dass es um einen Wissenschaftler und seine Arbeit geht, daher ist Topic 3 als Störer zu identifizieren.

Da Topics letztendlich für Menschen erstellt werden, wird in dieser Arbeit davon ausgegangen, dass diese Evaluationsmethode die hilfreichste ist. Für die systematische Evaluation allerdings wird hier stattdessen eine quantitative Methode gewählt, mit der die Kohärenz maschinell berechnet werden kann.

2.4.2 Unifying Coherence Framework

In Röder et al. (2015) wird die vierstufige Pipeline *Unifying Coherence Framework* beschrieben. Mit einem Referenzkorpus kann eine Topic-Kohärenz statistisch errechnet werden. Sie ist geeignet für alle bereits etablierten *Kohärenz-Methoden*, als auch für die bei Schöch et al. (2020) vorgestellte, vektorbasierte Methode[10]. Dabei zeigt sich, dass diese mit menschlichen Bewertungen korrelieren (vgl. Röder, n. d.).

Wie in Abbildung 3 zu sehen, bestehen die Stufen aus *Segmentation*, *Probability Calculation*, *Confirmation Measure* und *Aggregation*. Wörter aus dem Quelltopic t werden zur Segmentmenge aus Paaren S unterteilt, um sie untereinander zu vergleichen. Dabei gibt es viele Arten zu segmentieren, z.B. Einzelwortpaare, Paare aus

[10] Für tiefergehende technisch-mathematische Erläuterungen sei auf Röder et al., 2015 verwiesen.

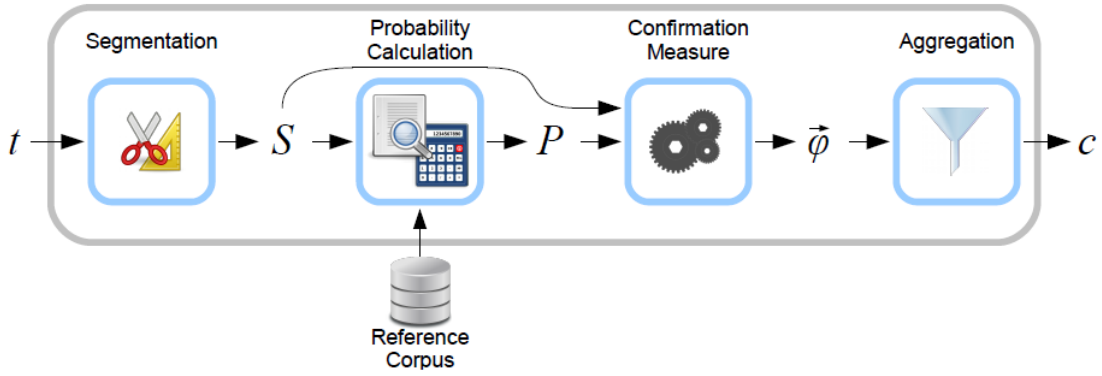


Abbildung 3: Überblick über das Unifying Coherence Framework (Quelle: Röder et al., 2015, S. 406)

Wortpaaren und weitere (vgl. Röder et al., 2015, S. 3f). Das ist von der Entscheidung der Nutzenden bzw. der Implementation des Tools abhängig. Die Wahrscheinlichkeiten P werden abhängig vom gewählten Referenzkorpus berechnet, der idealerweise nicht bloß ein Teil des untersuchten Korpus ist, sondern ein dem Model unbekannter Text. Aus Segmentmenge S und Wahrscheinlichkeiten P werden Bestätigungsmaße (Confirmation Measure) φ berechnet, die letztendlich zum Kohärenzwert c aggregiert werden. Dieser Wert kann mit unterschiedlichen Methoden berechnet werden. Im Grunde handelt es sich entweder um eine Wahrscheinlichkeitsrechnung oder einen vektorbasierten Ansatz, der nach Röder, Both und Hinneburg (2015) die besten Ergebnisse liefert. Wichtig ist für diese Untersuchung, dass nur die Kohärenzmaße verglichen werden, die auf gleiche Art mit den gleichen Parametern ausgerechnet wurden, da die Zahlen sonst nicht miteinander vergleichbar sind.

Das Tool *Palmetto* funktioniert auf Basis dieses Frameworks und ist als Web-Service verfügbar, für den auch der einfach zu bedienende Client `palmetto-py`[11] existiert, um Palmetto in den eigenen Python-Code einzubinden. Die Nutzung des Online-Services kann problematisch sein, wie eigene Erfahrungen zeigen: Zu Beginn der Verfassung dieser Arbeit war der Service nicht erreichbar, doch der Fehler konnte in Zusammenarbeit mit Röder behoben werden und der Service funktioniert zum aktuellen Zeitpunkt wieder. Wenn also Lesende dieser Arbeit auf Komplikationen stoßen, lohnt sich ein Blick ins Repository[12]. Daher steht Palmetto alternativ auch

[11] Siehe Github-Repository: https://github.com/dice-group/Palmetto/wiki/How-Palmetto-can-be-used.

[12] Für weitere Details und Hilfestellungen siehe das Git-Issue: https://github.com/dice-group/Palmetto/issues/51.

lokal als Java-Tool zur Verfügung: Dafür kann der Source-Code heruntergeladen und als Bibliothek für eigene Anwendungen verwendet oder als Java-Jar aus der Kommandozeile heraus gesteuert werden[13]. Der Nachteil besteht hier dabei allerdings darin, dass lokal das über 6 GB große Wikipedia-Referenzkorpus auch heruntergeladen werden muss.

Eine Alternative stellt die Open-Source-Bibliothek *Gensim* (von **Gen**erate **Sim**ilar) dar. Gensim ist ein mächtiges Werkzeug für Textverarbeitung und Erstellung von Topic Models und Word-Embeddings, welche von Radim Řehůřek und seinem Team entwickelt und gepflegt wird (s. Řehůřek & Sojka, 2010). Sie ist in Python geschrieben und erscheint daher gut geeignet für die Integration in diese Untersuchung. Neben den o.g. Funktionen können mit Gensim auch Kohärenzen berechnet werden, indem Objekte namens `CoherenceModel()` erstellt werden. Dafür wurden die Stufen des *Unifying Coherence Frameworks* implementiert und können mit den ebenfalls in Gensim erstellten Topic Modellen ausgerechnet werden. Allerdings wird für Gensims Kohärenzberechnung *kein* externes Referenzkorpus für die Wahrscheinlichkeitsberechnung verwendet (Schritt 2 im Framework). Es handelt sich also um ein intrinsisches Verfahren, dessen Ergebnisse kritisch zu hinterfragen sind, da die Kohärenz auf dem Quelltext berechnet werden. Sinnvoller erscheint die Betrachtung, soweit möglich, mit einem gleichsprachigen externen Korpus. So kann geprüft werden, ob die Kohärenzen nicht nur innerhalb des abgekapselten Quellkorpus sinnvoll sind.

In dieser Masterarbeit prüft der Autor zwei LDA-Implementationen auf ihre Performanz: zum einen Gensims LDA-Modelle und zum anderen die, die von MALLET (**MA**chine **L**earning for **L**anguag**E** Toolkit) erstellt werden können.(s. McCallum, 04/05/2021). Es handelt sich um ein in Java geschriebenes und dadurch plattformunabhängiges Programm, das einen Korpus einliest, ein LDA-Modell darauf trainiert und schließlich Topics generiert. MALLET wurde bereits vor ca. 19 Jahren (2002) publiziert und mutet in der Bedienung für heutige Verhältnisse veraltet bzw. umständlich an. Das liegt in erster Linie daran, dass das Interface lediglich die System-Kommandozeile darstellt und über die Befehle einzeln auszuführen sind. Allerdings

[13] Die Installationsanweisungen sind im gut dokumentierten Palmetto-Wiki auf Github zu finden: https://github.com/dice-group/Palmetto/wiki/How-Palmetto-can-be-used.

können Nutzende den quelloffenen Code in ihre eigenen Java-Applikationen einbinden und dessen API verwenden. Alternativ kann MALLET auch über Gensims Implementation genutzt werden, was aber Einschränkungen in der Parametersetzung mit sich bringt. Zudem sind die Analysedaten von MALLET selbst damit nicht zugänglich, die als separate Dateien auf dem System gespeichert werden. Besonders für kleinere Korpora, wie sie in den Geisteswissenschaften im Vergleich zu den enormen Mengen an Internetdokumenten (z.B. Tweets oder Nachrichtenportale) üblich sind, soll erfahrungsgemäß MALLET bessere Ergebnisse liefern als die LDA-Implementierung in Gensim.

2.4.3 MALLETs Diagnostik-XML

Über die extrinsische Methode, also durch den Vergleich zwischen einem fremden Korpus und dem eigenen, hinaus, kann noch die intrinsische Variante zur Evaluation herangezogen werden, indem sie für jedes Format für sich erstellt und betrachtet werden kann.

MALLET kann eigene Analysen durchführen und speichert diese in mehreren Dateien. Um zu bewerten, inwiefern Topics je nach Textformat besser oder schlechter sind, kann eine von MALLET generierte Diagnose-XML herangezogen werden. Sie enthält für jedes erstellte Modell Informationen über Token und Wörter in Bezug auf die Dokumente. Solch eine Datei hat den folgenden Aufbau:

```
1 <model>
2     <topic id="16" tokens="4960.0000" document_entropy="6.3355" word-
          length="5.2500" co-herence="-598.4534" uniform_dist="4.8426"
          corpus_dist="3.8750" eff_num_words="72.1311" token-doc-diff="
          0.0104" rank_1_docs="0.1478" allocation_ratio="0.0140"
          allocation_count="0.0770" exclusivity="0.4766">
3         <word rank="1" count="377" prob="0.07601" cumulative="0.07601"
              docs="184" word-length="5.0000" coherence="0.0000"
              uniform_dist="0.5701" corpus_dist="0.4515" token-doc-diff="
              0.0025" exclusivity="0.9669">wheat</word>
4     </topic>
5 </model>
```

Code 1: Auszug aus einer von MALLET erstellten Diagnostik-XML.

Wie zu sehen ist, werden für Topics (`<topic>`) und ihre Top-Wörter (`<word>`) je

eigene XML-Elemente generiert (Z. 2 u. 3, `topic` und `word`) und mit zahlreichen Attributen ergänzt. Die Beschreibungen sind auf der offizielle Homepage zu finden (s. McCallum, 2002), von denen eine Auswahl hiermit zusammengefasst ist:

- `tokens` misst die Anzahl der Wort-Token, die diesem Topic zugewiesen sind.
- `document_entropy` beschreibt die Entropie eines Dokuments bzw. seine Vorhersagbarkeit. Ein Topic mit hoher Entropie verteilt sich gleichmäßig über alle Dokumente, eins mit niedriger in wenigen.
- `word-length` ist der Durchschnitt der Wortlänge, gemessen in seinen Characters, der Aufschluss darüber gibt, wie spezifisch ein Topic ist. Es wird angenommen, dass längere Wörter spezifischer sind als kürzere.
- `coherence` ist MALLETs Kohärenzwert, der nicht nach dem Unifying Framework generiert wurde.
- `uniform_dist` misst, wie spezifisch ein Topic ist. Je höher der Wert, desto spezifischer. Berechnet wird die Distanz mit der Kullback-Leibler-Divergenz[14].
- `effective number of words` misst ebenfalls die Spezifizität.
- `corpus_dist` misst die Distinktivität eines Topics. Der Wert korreliert mit der Wortzahl des Korpus. Ebenso mit Kullback-Leibler-Divergenz berechnet.
- `rank 1 documents` misst die Frequenz, mit welcher das Topic meistfrequentiert in einem Dokument ist. Je höher der Wert ist, desto korpusspezifischer.

Interessant sind hauptsächlich die Attribute `document_entropy`, `uniform_dist` und `corpus_dist` für diese Untersuchung und sollten zusammen mit den Kohärenzen ein umfassendes Bild über die Modellunterschiede bieten.

[14] Etabliertes Maß zur Messung von Verteilungsenthropie bei Wahrscheinlichkeiten. Einen anschaulichen Exkurs bietet: https://www.countbayesie.com/blog/2017/5/9/kullback-leibler-divergence-explained.

3 Die Formate

Abbildung 4 stellt eine Übersicht über die erwartbare Nützlichkeit von Textformaten für verschiedene DH-Methoden dar.

abgeleitetes Textformat	Nützlichkeit für die Forschung							rechtliche Beurteilung		
	Stilometrie	Distinktivität	Topic Modeling	Sentiment Analyse	Netzwerkanalyse	Text Re-Use	Sprachmodelle	Schutz vor Wiedererkennbarkeit	Schutz vor Rekonstruierbarkeit	Unmöglichkeit des Werkgenusses
5.1.1. Einfache Term-Dokument-Matrix	+	+	o	–	–	–	–	+	+	+
5.1.2. Segmentweise aufgehobene Sequenzinf.	+	+	+	o	o	–	o	o	o	+
5.1.3. Selektiv reduzierte Information über Tokens	–	o	+	o	o	+	o	o	o	o
5.2.1. N-Gramme auf Teilkorpus-Ebene	–	o	–	–	–	–	+	o	o	+
5.2.2. einfache Wortembeddings	+	–	–	–	–	–	+	+	+	+
5.2.3. kontextualisierte Embeddings	o	o	o	o	o	o	+	+	+	+

Abbildung 4: Nützlichkeit von Textformaten (Quelle: Schöch et al., 2020)

Es ist zu sehen, dass die Erwartungen an die Term-Dokument-Matrix nicht hoch sind. Gut zu eignen scheinen sich aber die beiden Formate segmentweise Aufhebung der Sequenzinformation und die selektiv reduzierte Information über Tokens. Schlecht oder nicht geeignet hingegen sind die N-Gramme.

Im Folgenden wird beschrieben, welche Eigenschaften die Formate aufweisen und Erwartungen über ihre Eignung formuliert, die sich darin bemisst, welche Kohärenzen sie im Vergleich zum Original erreichen können.

3.1 Term-Dokument-Matrix

Die Term-Dokument-Matrix stellt wegen der eingeschränkt vorhandenen Sequenzinformation sicher ein für TM problematisches Format dar und wird weniger kohärente Topics erzeugen als das Original.

Token	Häufigkeit
His_PP$_his	125
England_NP_England	125
other_JJ_other	123
Lucretia_NP_Lucretia	123
us_PP_us 122	122
character_NN_character	120

Tabelle 1: Ausschnitt aus einer TDM, erstellt aus dem Featureset von `ENG18440_Disraeli.txt`

Sie enthält die Häufigkeitsinformationen über jedes im Quelltext vorkommende Token und kann wie in Tabelle 1 dargestellt aussehen. Die auf Basis von Einzeltexten erstellten Formate sind dem über Dokumentgrenzen hinweg erstellten vorzuziehen, denn dort ist der semantische Inhalt noch in grober Weise gegliedert und die Sequenzinformation nicht vollkommen verloren. Auf Basis des Gesamtkorpus ist diese Information nicht mehr enthalten und sie könnte nur mit verschwindend geringer Wahrscheinlichkeit zufällig wiederhergestellt werden, also kann das als ausgeschlossen angesehen werden. Erwartbar sind daher bei der ersten Variante kohärentere Topics als bei letzterer, allerdings dürften die Ergebnisse trotzdem nicht zufriedenstellend sein, da es sich um Romane mit zehntausenden Wörtern handelt. Je kleiner also die Einzelwerke oder -dokumente eines Korpus sind, desto größer die Chance auf bessere Topic Models.

Um einen Text aus einer Matrix zu erstellen muss jeder Term mit seiner Anzahl multipliziert werden, sodass sie tatsächlich als String vorhanden sind. Es wäre sinnlos, solche Strings mit LDA zu verarbeiten, da so über weite Passagen hinweg immer gleiche Token aufeinanderfolgen und somit Dokumente entstehen würden, die u.U. nur aus aufeinanderfolgenden Kommata oder Funktionswörtern (z.B. Konjunktionen) bestehen. Stattdessen kann eine textähnliche Struktur erreicht werden, indem sie zufällig neu angeordnet werden. Dieser Prozess ist technisch relativ einfach umzusetzen, wird aber mit großen Korpora durch die Multiplikation vermutlich längere Rechenzeiten verursachen. Trotzdem sind die durch Zufallsanordnung erstellten Dokumente entweder semantisch unsinnig oder es werden andere Topics aufgedeckt, als sie tatsächlich im Quelltext enthalten waren.

Ein Schlüssel zur Verbesserung dieses Formates liegt in der Segmentgröße. Denkbar

ist die einfache TDM in kleinere Segmente zu unterteilen, indem die Quelldokumente vorsegmentiert werden. Praktisch kann das umgesetzt werden, indem man vor der Erstellung der TDM das Gesamtdokument halbiert und aus der ersten und zweiten Hälfte der Wörter die Matrizen erstellt, z.B. aus einem Text mit 40000 Token zwei kleinere Segmente mit 20000. So bleibt zumindest äußerst grob die Information erhalten, ob die Wörter jeweils eher am Anfang oder am Ende stehen und ist bei großen, epischen Texten unproblematischer als beispielsweise bei Zeitungstexten. Es kann dann rekursiv verfeinert werden, indem jeweils jede Hälfte weiter halbiert, gedrittelt oder geviertelt wird, um die Position der Wörter in Segmenten immer weiter einzukreisen. Alternativ kann vom Nutzer auch eine relativ große Segmentgröße manuell bestimmt werden, doch sollte diese deutlich größer sein, als die für das Topic Modeling übliche Größe von 500-2000 Token. Diese Zahlen sind nur grobe Richtwerte und sind daher nicht als absolute Grenzwerte einzuordnen. Das Format lässt sich daher nur eingeschränkt für Topic Modeling verwenden und bietet allenfalls einen äußerst groben Überblick über den Inhalt des Textes durch inkohärente Topics. Die Strategie der immer feineren Segmentierung sei allerdings nicht in der Praxis empfohlen. Der Vorteil einer TDM liegt in ihrer Kompaktheit, der auf diese Weise stark untergraben wird. Zudem steigt der Rechenaufwand durch die Aggregierung und anschließenden Auflösung der Matrix ins Textformat erheblich an.

Daher bietet sich das im folgenden Unterkapitel vorgestellte Textformat, der segmentweisen Aufhebung der Sequenzinformation (SAS), als bessere Alternative an, da sie den gleichen Informationsgehalt aufweist, wie mehrere segmentierte Term-Dokument-Matrizen.

3.2 Segmentweise Aufhebung der Sequenzinformation

Führt man den Gedanken fort, dass kleiner werdende Dokumente bessere Ergebnisse erzielen und unterteilt somit Einzeltexte in noch feinere Segmente, dann bilden die so erstellten Matrizen gewissermaßen die Basis für dieses Format. Technisch gesehen ist das keine empfehlenswerte Herangehensweise, weil so die Rechenzeit durch die Multiplikation steigt. Stattdessen sollten die Einzeltexte zunächst segmentiert und deren Token zufällig vertauscht werden, denn dieser Prozess ist weniger aufwendig und sollte daher der TDM vorgezogen werden.

```
1 Sherlock_NP_Sherlock chemical_NN_chemical the_DT_the doing_VVG_do
    for_IN_for the_DT_the there_EX_there if_IN_if rooms_NNS_room
    the_DT_the he_PP_he second_JJ_second me_PP_me ._SENT_. Jove_NP_Jove
     at_IN_at hansom_NN_hansom gave_VVD_give lodgings_NNS_lodgings
    comfortable_JJ_comfortable my_PP$_my you_PP_you to_TO_to am_VBP_be
    ._SENT_.
```

Code 2: Ausschnitt eines Textsegements bestehend aus Features mit Zufallssequenz, erstellt aus den Featureset von `acd01.txt`

Wird dieses Format auf der gleichen Segmentbasis erzeugt wie das Original, sollte das Format die gleichen Kohärenzen erzielen können, denn jedes Segment beinhaltet die exakt gleichen Token. Wie oben erläutert, erstellt ein LDA-Algorithmus Bag-of-Word-Modelle auf Basis jedes Dokuments, wodurch ein aus dem Original generiertes Bag-of-Words-Modell keinen Unterschied zu diesem Format aufweisen dürfte.

Diese Eigenschaft bietet zudem den praktischen, einzigartigen Vorteil gegenüber anderen Formaten, dass er für das Topic Modeling auf exakt gleiche Art vorverarbeitet wird, ohne dass eine zusätzliche Programmlogik entwickelt werden muss. Der Zwischenschritt zur Erlangung einer Textform wie bei den anderen Formaten entfällt komplett.

3.3 Selektiv reduzierte Information über einzelne Tokens

Das Format der selektiv reduzierten Information über einzelne Tokens (TKN) ist ebenso vielversprechend und sollte ebenfalls keine signifikant schlechteren Kohärenzen erzielen als das Original.

Es gehen für Topic Modeling praktisch keinerlei Informationen verloren, da für semantische Analysen ohnehin nur bestimmte Wortarten von Interesse sind und enthält im Grunde, sofern auch hier ein Segment aus den gleichen Token besteht, für TM die gleichen relevanten Informationen. Zu den semantisch interessanten Wortformen zählen sicher Substantive, aber eventuell auch Verben und Adjektive und Adverbien. Andere Wörter und Zeichen werden ohnehin herausgefiltert und entfernt.

```
1 IN DT year_NN_year CD PP took_VVD_take PP$ degree_NN_degree IN NP IN NP
    IN DT NP IN NP , CC proceeded_VVD_proceed TO NP TO go_VV_go IN DT
    course_NN_course prescribed_VVN_prescribe IN surgeons_NNS_surgeon
    IN DT army_NN_army SENT VHG completed_VVN_complete
```

Code 3: Ausschnitt eines Textes, in dem nur Substantive, Adjektive und Verben erkenntlich sind. Erstellt aus den Featureset von `acd01.txt`

Wird also ein Format erstellt, bei dem eben nur jene Wortarten erkenntlich sind und die restlichen verschleiert, dann greift dieser Filter ebenfalls und es stehen die relevanten Informationen an denselben Stellen wie im Original. In Code 3 ist schon zu sehen, dass hier lediglich noch die Token entfernt werden müssen, die nicht Teil eines Features (in der Form *Token_Wortform_Lemma*) sind. Dieses Muster in einem Skript zu erkennen stellt keine große Herausforderung dar, da in diesem Beispiel einfach jedes mit Leerzeichen getrennte Wort auf die Länge seiner Bestandteile geprüft werden kann. Ist die Länge gleich 3, wird das Wort behalten, andernfalls wird es entfernt.

Ein Nachteil dieses Formates ist die Unwiederbringbarkeit der herausgefilterten Wortarten. Sind diese erst entfernt worden, kann diese Information nicht wiederhergestellt werden, was bei der SAS anders ist. Dort können die Informationen jederzeit anders gefiltert werden, bei der TKN müsste dafür immer eine neue Datei zur Verfügung stehen, wenn doch andere Informationen benötigt würden.

3.4 N-Gramme

N-Gramme sind sehr problematisch, denn Sequenzinformationen sind zum größten Teil verloren. Lediglich die N-Gramme selbst sind sehr kurze Textsequenzen, die aber kaum verwertbar sein dürften. Es handelt sich nämlich um eine tabellarische Häufigkeitsdarstellung von kurzen Wort- bzw. Tokensequenzen, dessen Länge üblicherweise 2-5 beträgt. Soll beispielsweise für den String `'This is a text with 7 Tokens'` 3-Gramme erstellt werden, wird dies für jedes einzelne erstellt. Die 3-Gramme für diesen String lauten also: `'This is a'`, `'is a text'`, `'a text with'`, `'text with 7'` und `'with 7 Tokens'`, sowie aus den Resten `'7 Tokens'` und `'Tokens'`. Beim Anblick dieser 3-Gramme wird allerdings klar, dass der Text sich vollständig aus die-

sem Format rekonstruieren ließe (man füge nur die ersten Token jedes N-Gramms aneinander) und ist daher aus urheberrechtlicher Sicht äußerst bedenklich. Eine naive Entfremdungsmaßnahme könnte darin bestehen, die Reihenfolge der N-Gramme vollkommen zufällig neu anzuordnen, was bei großen Korpora die Rekonstruktion des Textes erschweren würde.

Es können also nicht alle N-Gramme aufgelistet werden, daher muss Information entfernt werden, indem eine Mindesthäufigkeit (Threshold) definiert wird. Das bedeutet, die N-Gramm-Häufigkeit muss in einem Dokument höher sein, als die definierte Mindesthäufigkeit (`>1`). Damit also beispielsweise das 3-Gramm `'is a text'` bei Mindesthäufigkeit 4 im Format aufgenommen wird, müsste es in einem Dokument mindestens vier mal gezählt werden. Dies ist bereits bei kleineren N-Grammen unwahrscheinlich, aber möglich. Damit wäre die Rekonstruktion nicht mehr möglich, da ein erheblicher Tokenanteil entfernt wurde. Es bleibt aber fraglich, ob mit dem Bruchteil der ursprünglichen Token überhaupt noch sinnvolle Topics erstellt werden können und die Wahrscheinlichkeit, dass beispielsweise ein 5-Gramm in einem relativ kleinen Segment häufiger als ein mal vorkommt, ist verschwindend gering. Das sollte zur Folge haben, dass diese nur dann ermittelt werden können, je größer die Segmentgröße gewählt wird.

Entfernt man sich aber von der Dokumentebene und aggregiert die N-Gramme über (alle) Dokumentgrenzen hinweg, kommt wie bei den Korpus-TDM erschwerend hinzu, dass keinerlei Information darüber bekannt ist, wo sich N-Gramm-Kookurrenzen finden. Die bestmöglichen (das ist in diesem Fall vermutlich noch deutlich schlechter als bei jedem anderen Format) Ergebnisse sollten demnach mit einer Balance aus N-Gramm-Größe, Mindesthäufigkeit und Segmentlänge erzielt werden können. Für nützliches Topic Modeling dürfte dieses Format allerdings nicht geeignet sein und ein Vergleich mit dem Bruchteil der Textmenge mit dem Original kaum sinnvoll.

Tabelle 2 zeigt eine N-Gramm-Matrix aus N-Gramm-Größe 3, welche über alle verfügbaren Dokumente des Korpus aggregiert wurden.

Zudem ist es schwierig, einen klaren Vorschlag zur Rekonstruktion eines textnahen Formates zu definieren. Denkbar ist ein ähnliches Vorgehen wie bei der TDM, nur das

3-Gramm	Häufigkeit
say mind do	10
want say anything	10
day come say	10
shake head seem	10

Tabelle 2: N-Gramm-Matrix mit N-Gramm-Länge 3 bestehend aus lemmatisierten Substantiven, Verben und Adjektiven. Erstellt auf Grundlage des Gesamkorpus

hier jedes N-Gramm als Einheit vervielfältigt wird und dann auch als Einheit wieder zufällig im String angeordnet wird. Durch Beibehaltung der N-Gramm-Einheiten ist zumindest die tatsächliche Kookurrenz dieser Wörter garantiert.

4 Der Workflow

Um die Praktikabilität von Textableitungen für TM zu untersuchen, müssen sie zunächst aus einem Originalkorpus durch Informationsreduktion tatsächlich erstellt und anschließend systematisch aufbereitet werden, sodass eine für TM geeignete Textdatei entsteht. Das bedeutet, dass für jedes Format Algorithmen entwickelt wurden, die die eventuell hinzugefügten Metainformationen (Sequenzinformationen, Part-of-Speech-Tagging, Häufigkeitsangaben) verwerten, filtern und anschließend entfernen. Auf diese Weise werden die Formate in quasi-natürliche Sprache umstrukturiert. Diese Pseudotexte werden anschließend wie Originale für TM vorverarbeitet, mit denen einschlägige TM-Tools schließlich in der Lage sind Topics zu erzeugen. Es werden bewusst Formate erstellt, die, sofern sinnvoll und möglich, nicht bereits an sich für das TM optimiert sind, sondern in einer allgemeinen Form für alle Text-Data-Mining zur Verfügung gestellt werden können. Das heißt, dass alle Formate alle Informationen zu Token, Wortart und Lemma zur Verfügung stellen und nicht etwa schon für TM vorgefiltert sind. Damit wird ein Vorschlag unterbreitet, wie Forschende mit den Formaten umgehen könnten.

Um das zu Erreichen, wird der Prozess als Workflow abgebildet. Dieser Workflow wurde als Aneinanderreihung mehrerer separater Prozesse realisiert, was entscheidende Vorteile mit sich bringt. Durch die schrittweise Verarbeitung der Textdaten können Skripte[15] je einer Stufe dediziert werden, was dem Anwender eine höhere Übersichtlichkeit und Nachvollziehbarkeit über den Gesamtprozess ermöglicht. Die Zwischenschritte werden auf diese Weise im Dateisystem nach und nach persistiert und gehen bei einem unerwartetem Abbrechen des Skriptes nicht verloren. Denn mit dem Anstieg der Datenmengen kommt es bei komplexeren Verarbeitungsschritten dazu, dass sich die benötigte Rechenzeit erhöht. Der Verlust jeglichen Fortschritts ist ein Ärgernis, womit aber mit diesem Vorgehen etwas entgegengesetzt werden kann. Zudem gestaltet sich das Debugging (Ausfindigmachen und Ausmerzen von Programmfehlern) und die Modifikation des Codes als signifikant einfacher, weil so die Logik in kleinere, in sich geschlossene Skripte unterteilt ist. Diese in sich geschlossenen Skripte können auch außerhalb dieser Pipeline für andere Untersu-

[15] Die Skripte liegen im Repository https://github.com/s2mako/master-thesis vor. Auf dem hier beigelegten USB-Stick befinden sich darüber hinaus noch Ordnerstrukturen und einige Textdateien, sowie Topic Modelle.

chungen weiterverwendet werden. Zur Verwaltung solcher Teilskripte bietet es sich an, übergeordnete Kontrollskripte zu erstellen. Diese können je Format und Stufe die Quell- und Zielordner bestimmen und ermöglichen es dem Anwender spezifische Verarbeitungsparameter zu definieren.

Einen graphischen Gesamtüberblick über den Workflow liefert das im Anhang das im Anhang befindliche Flowchart[16]. Diese Grafik ist unterteilt in acht Swimlanes, die jeweils eine Sinneinheit bilden. Die in hellblau hinterlegten Kästen mit wellenförmigen Abschnitt stehen für eine oder mehrere Textdateien. Die gelb hinterlegten Kästen stellen in sich abgeschlossene, externe Programme dar (Tagging und TM) und graue Kästen sind Prozesse, die Bestandteile aus Python-Skripten darstellen und damit selbst geschrieben und implementiert wurden. Die grünen Hexagone stehen für Parameter, die entweder in einer separaten Datei oder im Skript selbst bestimmt werden.

In den folgenden Unterkapiteln werden Ausschnitte daraus präsentiert und im Detail erläutert.

4.1 Das Korpus

4.1.1 Quellen

Noch vor dem Beginn der eigentlichen Pipeline muss das Korpus beschafft werden (Abbildung 5). Die Beschaffung der Quelldaten gilt nur für TEI-Dateien, da sich dieser Vorgang nicht verallgemeinern lässt. Sobald allerdings die Daten einheitlich als Textdateien vorliegen, kann der Prozess auf alle Quellen angewandt werden. Im Folgenden werden die für die Erstellung dieses Korpus ausgeführten Schritte erläutert.

Für die hiesige Auswahl des Korpus sind die Kriterien *Gemeinfreiheit*, *Umfang* und *Homogenität* im Fokus. Die Texte sollen gemeinfrei sein, damit sie ohne rechtliche Komplikationen mit dieser Masterarbeit publiziert werden können. Damit können die interessierten Lesenden das Originalkorpus in seiner Gänze betrachten, die abgeleiteten Textformate mit den Skripten aus dem referenzierten Repository erstellen,

[16] Erstellt mit dem Online-Tool *Lucidchart*: https://www.lucidchart.com.

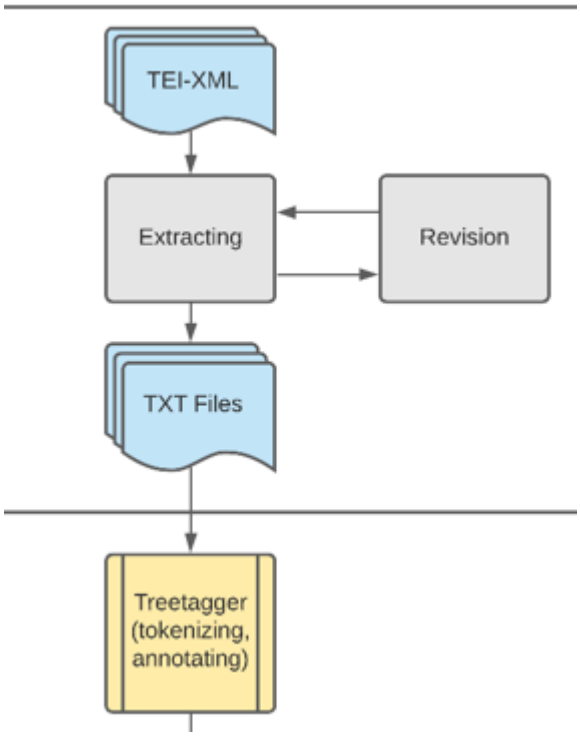


Abbildung 5: Extraktion des Korpus TEI

die hierfür benötigten Schritte selbst nachvollziehen und sie für eigene Forschungszwecke weiterverwenden.

Der Umfang des Korpus ist wichtig, denn ist die Dokumentenanzahl zu klein, mit denen ein Topic Modeling Algorithmus trainiert wird, sind die so erzeugten Topics nur schwierig oder nicht zu interpretieren. Dies kann aus eigener Erfahrung bestätigt werden, denn zu Beginn der Untersuchung wurde das Korpus mit 29 Romanen zu klein gewählt. Die Konsequenz war, dass die Kohärenz der Topics unbefriedigend klein ausfiel, sodass es beim Großteil subjektiv nicht nachvollziehbar war, welche Topics durch die Wörter abgebildet wurden. Zwar ist der Anspruch dieser Masterarbeit nicht die Generierung möglichst kohärenter Topics, doch trotzdem sollte für eine klarere Analyse ein qualitatives Mindestmaß vorhanden sein. Durch Vergrößerung der Textmenge auf 129, stieg die Qualität der Topics dann jedoch deutlich.

Sprachliche Homogenität ist für das Training des TM-Tools wichtig. Homogen sind die Dokumente hinsichtlich ihrer Gattung (Romane), die Hauptsprache ist Englisch[17] und ihre Publikationsdaten erstrecken sich über einen Zeitraum von maximal

[17] Die Entscheidung für englische Texte ist rein pragmatisch, da Palmetto standardmäßig die englische Wikipedia als Referenzkorpus anbietet und somit kein weiterer erstellt werden muss.

110 Jahren. Es kann davon ausgegangen werden, dass sich die englische Sprache in diesem Zeitraum nicht zu stark verändert hat, sodass sie vergleichbar bleiben.

Das Korpus besteht aus 129 englischen Romanen aus dem Zeitraum 1810 bis 1920. Es speist sich zum Großteil aus dem englischsprachigen Teil der European Literary Text Collection (ELTeC) (100 Romane) und zu einem Teil aus dem Oxford Text Archive (OTA) (17 Romane).

Weitere 12 Romane sind aus dem im Rahmen der vDHd2021 stattgefundenen Digital-Humanities-Workshop *Kontrastive Analyse literarischer Texte mit Zeta* (s. Projekt Zeta Team, 2021; vDHd2021 - Experiments Team, 2021). Dabei handelt es sich um alle im Korpus enthaltenen Romane des Autors Arthur Conan Doyle. Da sie bereits als reine Textdateien vorliegen, müssen diese auch nicht weiter vorverarbeitet werden.

Die ELTeC ist ein Github-Repository[18], das von der European Cooperation in Science and Technology (COST Association) für das Projekt Distant Reading for European Literary History erstellt wurde (vgl. Distant Reading for European Literary History, 03/05/2021). Es beinhaltet Texte vieler europäischer Sprachen, die zwischen den Jahren 1840 und 1920 publiziert wurden. Das hat pragmatische Gründe:

> (...) we begin in 1840, because starting at around this time, a number of novels that is sufficient for our purposes has been published in many European languages. And we end in 1920 because this allows us to focus exclusively on texts that are in the public domain. As a consequence, ELTeC can be shared freely and re-used as widely as possible, without restrictions imposed by copyright law. (Schöch et al., 2021, S. 3f)

Das Repository wurde also genau für Untersuchungen wie diese geschaffen.[19] Durch die Publikation über Github sollten diese Dateien auch langfristig verfügbar sein. Die Dateien liegen in TEI[20] kodiert vor und sind durch das Klonen des Repositorys mittels Git oder einem alternativen Git-Client sehr komfortabel und schnell herunterzuladen.

[18] Abrufbar unter: https://github.com/COST-ELTeC/ELTeC-eng.

[19] Es handelt sich bei dem Repository zudem um ein gutes Beispiel für die Notwendigkeit, sich in der Wissenschaft um pragmatischen Umgang mit Urheberrechten zu beschäftigen.

[20] Die Text Encoding Initiative (TEI) ist ein in den (digitalen) Geisteswissenschaften verbreiteter Standard zur Beschreibung von Dokumenten aller Arten. Eine TEI-Datei ist eine XML-Datei, welche nach TEI-Schema strukturiert ist. Dieses Schema ist modular aufgebaut und kann je

Das OTA ist ein Online-Archiv der University of Oxford (vgl. University of Oxford, 2019). Über ihre Suchmaske und Filter lassen sich beispielsweise Sprache und Zeitraum literarischer Werke suchen. Sie liegen uneinheitlich als reine Textdateien, im TEI-Format oder als Kombination aus TEI-Header und Textdatei vor. Die Suchmaske erlaubt es nicht nach gemeinfreien Texten zu filtern, stattdessen sind die Ergebnisse als *Publicly Available* oder *Academic Use* gekennzeichnet. Auf diese Weise ließen sich 17 Texte ausfindig machen, die nicht zu weit in der Vergangenheit zurückliegen, dass sie sich sprachlich zu stark vom ELTeC-Korpus unterscheiden und nicht bereits in ihm vorhanden sind. Diese wurden manuell über einen Browser gesucht und heruntergeladen.

Für größere Datenmengen böte sich die Entwicklung eines Spiders für die automatische Erfassung an. Leider gibt es beim OTA oft Weiterleitungen auf ältere Webseiten mit uneinheitlichen HTML-Strukturen und uneindeutige Benennung der Dateien. Der Entwicklungsaufwand, ist im Hinblick auf den Nutzen für wenige Dateien im Vergleich zum manuellen Download, zu hoch. Alternativ könnten Suchmaschinenergebnisse mit Eingrenzung auf die OTA-Website die Navigation erleichtern.

Im Repository befindet sich eine Tabelle, die die Romane im Einzelnen mit Autorschaft, sowie Publikationsdatum und Herkunft aufgelistet.

4.1.2 Vorverarbeitung

Wie beschrieben, liegen viele der ursprünglichen Quelldateien als TEI-XML vor. Sie beinhalten den Text und Metadaten in Form von XML-Tags. Daher ist es notwendig den Reintext aus der XML zu extrahieren und ausgewählte Metadaten in Form einer Tabelle zu sichern. Hierzu wurden Hilfsskripte auf einer Codebasis des Informatikers Maximilan Konzack (vgl. Konzack, 2019) geschrieben, welches eine TEI-File parst und die Inhalte in einem TEI-Objekt abbildet. Dieses Objekt beinhaltet aus dem TEI-Header die Namensangaben über Autorinnen, den Titel, das Publikationsdatum (siehe Code 4) und den Textinhalt. In Python lässt sich ein

nach benötigtem Detailgrad komplexer oder einfacher strukturiert sein. So können eine Vielzahl von Metainformationen über das Dokument mitgeliefert werden. Um ein auf die eigenen Bedürfnisse zugeschnittenes Schema zu erstellen, kann das *Roma*-Tool online genutzt werden. Es bietet an, aus diversen Modulen (z.B. Gedichte, Musiknoten, etc.) Elemente und Attribute auszuwählen und als gültiges Schema abzuspeichern. Siehe: https://roma2.tei-c.org.

XML-Baum als `ElementTree` aus der nativen Bibliothek `xml` abbilden und dank XPath-Unterstützung durchsuchen. Eine beispielhafte Erklärung am Datumsfeld:

```
1 @property
2 def date(self):
3     if not self._date:
4         xpath = ".//xmlns:bibl[@type='firstEdition']//xmlns:date"
5         self._date = self.root.find(xpath, self.ns).text
6     return str(self._date)
```

Code 4: Befüllung einer Date-Property aus einer TEI-Datei mittels XPath. Auszug aus `TEIFile.py`.

Es ist zu beachten, dass sich jedes Element im Namespace `http://www.tei-c.org/ns/1.0` befindet (siehe Code 5) und somit auch jeder Knoten im XPath mit dem Präfix `xmlns:` versehen ist[21], ansonsten werden die Elemente im Baum nicht gefunden.

```
1 <TEI xmlns="http://www.tei-c.org/ns/1.0" xml:lang="en">
```

Code 5: Beispiel für einen TEI-Wurzelelement mit Namespace. Auszug aus `ENG18400_Trollope.xml`

Die Methode `find()` nimmt als zweites Argument den o.g. Namespace und liefert den ersten Treffer aus der XML zurück (Code 4, Z. 5). Die Analyse zeigt, dass nur das `date`-Element im Abschnitt `bibl` mit Attribut `firstEdition` interessant ist, was hier im XPath beschrieben wird (Code 4, Z. 4). Analog dazu werden die restlichen Metadatenfelder befüllt. Um sie in eine Tabelle zu schreiben wird ein `pandas` Dataframe[22] erstellt, die Spaltennamen festgelegt und die Metadaten übergeben:

```
1 result_csv = pd.DataFrame(csv_entries, columns=["Filename", "Lastname",
2 "Firstnames", "Title", "Date"])
3 result_csv.to_csv(r"tei\metadata.csv", index=False, encoding="utf-8")
```

Code 6: Erstellung eines Dataframes. Auszug aus `importteicorpus.py`.

[21] Es gibt eine Variante mit James-Clark-Notation (vgl. Clark, 1999), also mit der Namespace-URL in geschweiften Klammern vor dem Element. Dies führt aber schnell zu unübersichtlichen XPaths, was schnell anwenderunfreundlich werden kann.

[22] Mehr zu `pandas` in den folgenden Kapiteln. Ein Dataframe ist ein Objekt zur Repräsentation von Tabellen.

Die Extraktion des Autorinnennamens gestaltet sich etwas komplexer, weil die `author`-Elemente einen String nach dem Muster *Nachname, Vorname 1 Vornamen (Geburtsjahr-Sterbejahr)* beinhalten und keine eigenen Felder für die Einzelteile zur Verfügung stehen. Diese können aber mit Regex-Ausdrücken herausgefiltert werden (siehe `Author.py` und `TEIFile.py`).

```
1  @property
2  def text(self):
3      if not self._text:
4          divs_text = []
5          liminal_xp = ".//xmlns:text//xmlns:div[@type='liminal']"
6          body_xp = ".//xmlns:text//xmlns:body"
7          liminalnode = self.root.find(liminal_xp, self.ns)
8          bodynode = self.root.find(body_xp, self.ns)
9          if liminalnode:
10             for text in liminalnode.itertext():
11                 divs_text.append(text)
12         for text in bodynode.itertext():
13             divs_text.append(text)
14         plain_text = " ".join(divs_text)
15         self._text = plain_text
16     return self._text
```

Code 7: Extraktion von Textknoten mit `itertext()`. Auszug aus `TEIFile.py`.

Die Analyse der Quelldaten zeigt, dass sich relevanter Text in den `div`-Tags mit Attribut `type="liminal"` im Abschnitt `front` und in den `div`-Kindern des `body`-Elements befindet. Die mit `liminal` typisierten Elemente[23] enthalten Prologe und das `body`-Element den Haupttext eines Romans. Um nun die Textknoten zu extrahieren, nutzen wir die von Python zur Verfügung gestellte Bibliothek `xml`, die native Unterstützung für XML-Verarbeitung liefert. Sie beinhaltet unter anderem die Funktion `Element.itertext()`, die über die Kindknoten eines Elements iteriert und die reinen Textknoten zurückliefert. In Code 7 ist demonstriert, wie Prologknoten anhand ihres Types mit XPath gefiltert (Z. 5) und als Liste in einer Variable gespeichert werden (Z. 7). Analog dazu wird mit den `body`-Knoten verfahren (Z. 6 und 8). Falls es Prologtext gibt, wird er zuerst in einer Gesamttextliste gespeichert (Z. 9-11) und anschließend die Textknoten des `body` angehängt (Z. 12f). Die Listen-

[23] Im Abschnitt `front` sind noch Cover und Danksagungen zu Beginn eines Romans enthalten, weil sie nicht Bestandteil der eigentlichen Erzählung sind, werden diese nicht in den Korpus aufgenommen.

elemente werden dann zu einem String zusammengefügt (Z. 14) und in Form einer Textdatei persistiert (siehe `import_tei_corpus.py`).

Zuletzt werden die Dateien stichprobenartig angesehen, auf grobe Fehler überprüft und ausgebessert. Das gilt insbesondere für die uneinheitlichen Textdateien aus dem OTA. Bei einigen Dateien wurden Regex-Ausdrücke formuliert, um diese auszubessern. Beispielsweise war eine Datei wie eine Tabelle mit zwei tabulatorseparierten Zahlenspalten aufgebaut , die so ausgebessert werden konnte.

4.2 Tagging

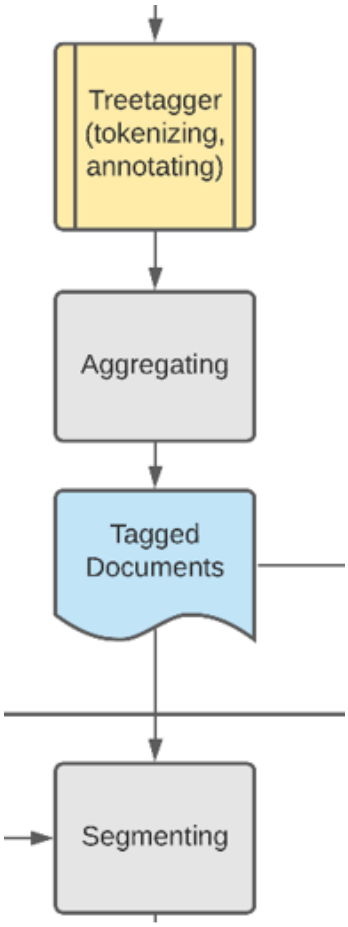


Abbildung 6: Der Tagging-Prozess

Es gibt zahlreiche Online-Tutorials zur Erstellung eines Skripts für die Vorverarbeitung von Texten. Viele empfehlen die Nutzung der populären Bibliothek `spaCy` (vgl. spaCy Dev Team, n. d. a). Sie ist in Python geschrieben und stellt ein mächtiges Werkzeug zur Sprachdatenverarbeitung dar. `spaCy` liefert u.a. eine anpassbare Pipeline für Tokenizing, Tagging und Parsing, die auf vortrainierten Korpora agieren (vgl.

spaCy Dev Team, n. d. b). Ursprünglich hat der Autor ein Skript zur Vorverarbeitung der Originaltexte auf Basis dieser Pipeline erstellt, doch die Formatting-Skripte von Schöch sind auf Basis des *Treetaggers* geschrieben. Um eine vergleichbare Basis durch einheitliche Segmentgrößen und Tagging zu gewährleisten, wurde dieser Ansatz zugunsten des Treetagger jedoch verworfen, um Aufwände zu verringern. `spaCy` sei aber nachdrücklich als gute, moderne Alternative empfohlen.

Der Treetagger ist ein von Helmut Schmid im Rahmen des an der Universität Stuttgart initiierten Projekts *Textual corpora and tools for their exploration* (s. University of Stuttgart, 2021) entwickeltes Tool zur Textannotation mit POS- und Lemmainformationen. Es wird von allen gängigen Betriebssystemen unterstützt und ist in vielen Sprachen anwendbar (Schmid, 1994, 1995). Um es als eigenes Kommandozeilentool zu nutzen, ist der Download von der offiziellen Webseite erforderlich und den spezifischen Installationsanweisungen zu folgen: https://cis.uni-muenchen.de/~schmid/tools/TreeTagger/. Als Python-Paket steht es dann zur Verfügung, wenn der `treetaggerwrapper` über PiPy mit `pip install treetaggerwrapper` installiert wird.

Token	POS	Lemma
There	EX	there
's	VHZ	have
been	VBN	be
a	DT	a
woman	NN	woman
here	RB	here
,	,	,
„	„	„
he	PP	he
cried	VVD	cry
.	SENT	.

Tabelle 3: Beispiel für einen Text mit POS und Lemma für jedes seiner Token

Tabelle 3 repräsentiert beispielhaft eine Ausgabedatei, die mit dem Treetagger erstellt wurde. In der ersten Spalte werden Token aufgelistet, also die Wörter und Zeichen, wie sie tatsächlich im Text vorzufinden sind. Die zweite Spalte enthält die POS-Tags aus dem PENN-Tagset[24]. In der dritten Spalte, Lemma, werden die auf

[24] Die Tag-Beschreibungen können aus Tabelle 17 im Anhang entnommen werden.

ihre Grundform zurückgeführten Wörter aufgelistet, die die Basis für Topic Modeling darstellen. Diese Einheit wird im weiteren Verlauf auch als *Feature* bezeichnet. Interpunktionen werden auch als solche erkannt und Satzgrenzen gesondert mit `SENT` versehen. Interessant ist dabei, dass im Englischen mit Apostroph abgekürzte Wörter ebenfalls auf ihre Grundform zurückgeführt werden und hier *'s* korrekt als *have* und nicht etwa als *be* erkannt wurde.

Das Spaltenabgrenzungszeichen ist standardmäßig der Tabulator. Behält man jenes so bei, ist es logisch, für jeden getaggten Text auch je eine Tag-Datei zu erstellen. Bei großen Korpora führt das dazu, dass eine Vielzahl von Dateien im Filesystem liegen, die der Übersichtlichkeit schaden. Deswegen wurde stattdessen der Unterstrich als Trennzeichen verwendet, damit alle Dokumente zeilenweise gespeichert werden können. Auch die anderen Formate sollen, so weit möglich und sinnvoll, auf diese Art gespeichert werden. Dieser Prozess wird im Chart als *Aggregating* bezeichnet. Als Ergebnis erhält man dann Texte in dieser Form:

```
1 ENG18450_Disraeli\tSybil_NP_Sybil ,_,_, or_CC_or the_DT_the Two_CD_two
    Nations_NPS_Nations\n
2 ENG18410_Sinclair\tThe_DT_the newspapers_NNS_newspaper have_VHP_have
    recently_RB_recently adopted_VVN_adopt\n
```

Code 8: Auszug aus der Tagged-File

Durch Auftrennung am Zeilenumbruch erhält man einen Roman, an einem Tab den Dateinamen und Inhalt und an einem Leerzeichen die Liste aller Features.

```
1 def apply_tagger(text, params):
2     tagger = treetaggerwrapper.TreeTagger(TAGLANG=params["lang"])
3     tagged = tagger.tag_text(text)
4     tagged = [tag.replace("\t", "_") for tag in tagged]
5     tagged = " ".join(tagged)
6     return tagged
```

Code 9: Aufruf des Treetaggers. Auszug aus `formats0_tagging.py`.

Die Erstellung dieser Datei ist denkbar einfach, s. Code 9. Der aus `treetagger` importierten Klasse `Treetagger` wird beim Instanzieren im Konstruktor ein Sprachkürzel übergeben (Standard ist `en`), um festzulegen, in welcher Sprache die Quelltexte geschrieben wurden (s. Kap. 4.4). Dieses Objekt ruft die Methode `tag_text`

mit dem Quelltext als Argument auf und liefert eine Liste mit Zeilen zurück, die aufgebaut sind wie in Tabelle 3, dessen Tabs in Strings mit Unterstrichen ersetzt werden. Die Elemente werden dann in einem String zusammengeführt und als Datei gespeichert.

4.3 Segmenting

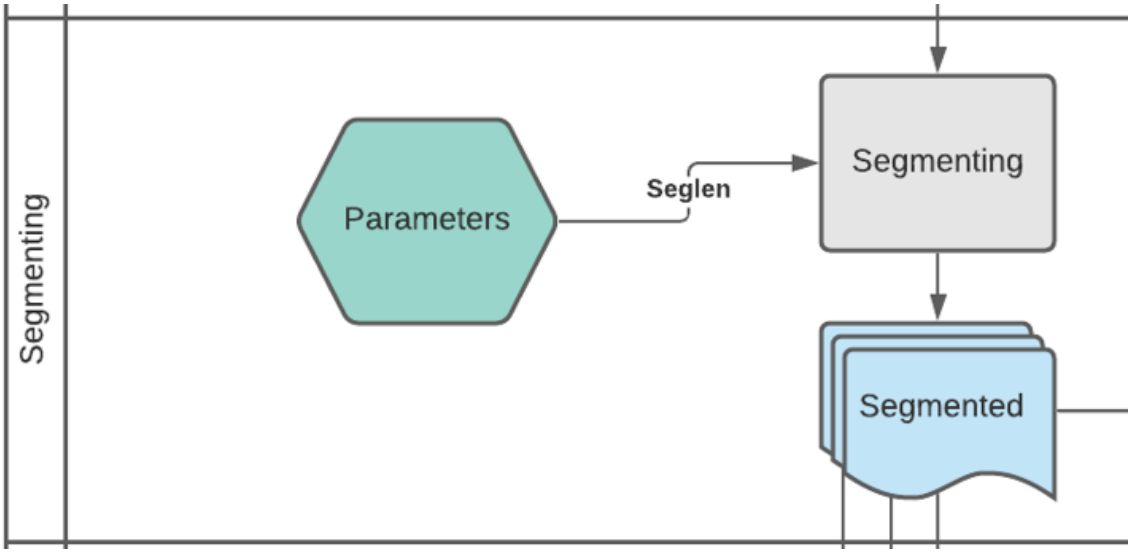


Abbildung 7: Der Segmenting-Prozess

Der wohl wichtigste Parameter bei der Erstellung der Formate, stellt die Segmentierung dar. Mit ihm wird bestimmt wie groß ein Dokument jeweils ist und beeinflusst damit indirekt die Anzahl der Dokumente, denn je größer die Segmentgröße, desto kleiner die Dokumentanzahl und umgekehrt. Das Ziel ist die Erstellung mehrerer segmentierter Dokumente, die als Grundlage der meisten Formate dient.

Hierfür wurde das Skript `formats0_segmenting.py` geschrieben, das die Tagged-Datei einliest, daraus Segmente generiert und diese in eigenen Dateien abspeichert. Der Name einer Segmentdatei setzt sich zusammen aus dem Präfix `segmented-` und der Segmentlänge, die aus der Parameterdatei (s. Kapitel 4.4.2) ausgelesen wird, also z.B. `segmented-500.txt` für Segmentlänge 500.

```
1 def create_segments(tagged, params):
2     tagged = tagged.split(" ")
3     segments = [tagged[x:x+params["seglen"]] for x in range(0, len(
          tagged), params["seglen"])]
4     return segments
```

Code 10: Der Segmentierungsprozess.

Die Funktion `create_segments()` in Code 10 stellt das Herzstück dar. Sie erhält das zu segmentierende Dokument in Form eines Strings, dessen Inhalt an einem Leerzeichen aufgespalten wird (Z. 6). Damit wird eine Liste erzeugt, die alle Features des Dokuments enthält. Die eigentlichen Segmente werden in Z. 7 generiert: es wird in einer List-Comprehension[25] über die Elemente der eben erzeugten `tagged`-Liste iteriert. Der Ausdruck `for x in range(0, len(tagged), params['seglen'])` bedeutet, dass `x` in jeder Iteration um die Segmentgröße (z.B. 500) erhöht wird. Mit `tagged[x:x+params['seglen']` wird bestimmt, welche Features von `tagged` in einer Unterliste zusammengefasst werden sollen. Setzt man `x` der ersten Iteration ein, ergibt sich der Ausdruck `tagged[0:0+500]`. D.h. die Elemente 0-499 sind nun in einem Dokument zusammengefasst (die zweite Zahl, hier 500, ist exklusiv). In der nächsten Iteration ist `x` um die Segmentgröße erhöht, also ist `x == 500`, was bedeutet, dass sich die nächsten Listenelemente aus `tagged[500:500+500]` ergeben. Dies wird so lange fortgeführt, bis die Liste keine Elemente mehr hat. In den meisten Fällen wird das bedeuten, dass das letzte Dokument aus weniger Elementen besteht, sofern die Gesamtanzahl an Features kein Vielfaches der Segmentgröße ist.

Die Funktion `write_to_file()` nimmt die so erzeugten Dokumentlisten und schreibt sie zeilenweise, als String zusammengefügt, in die Ausgabedatei. Jede Zeile erhält noch tabulatorsepariert den Quelldateinamen zur Identifikation und Kontrolle.

4.4 Formatting

Zur Erstellung der Textformate dienen als Grundlage Skripte, die Schöch im DH-Trier-Repository zum Download zur Verfügung stellt (https://github.com/dh-trier/

[25] Pythons List-Comprehensions stellen eine kompaktere Syntax zur Listengenerierung dar. Auf diese Weise können Listen iterativ in einer Codezeile erstellt werden, anstatt eine `for`-Schleife über mehrere Zeilen zu implementieren.

tmr). Diese wurden so modifiziert, dass sie als Teil dieser Pipeline funktionieren können.

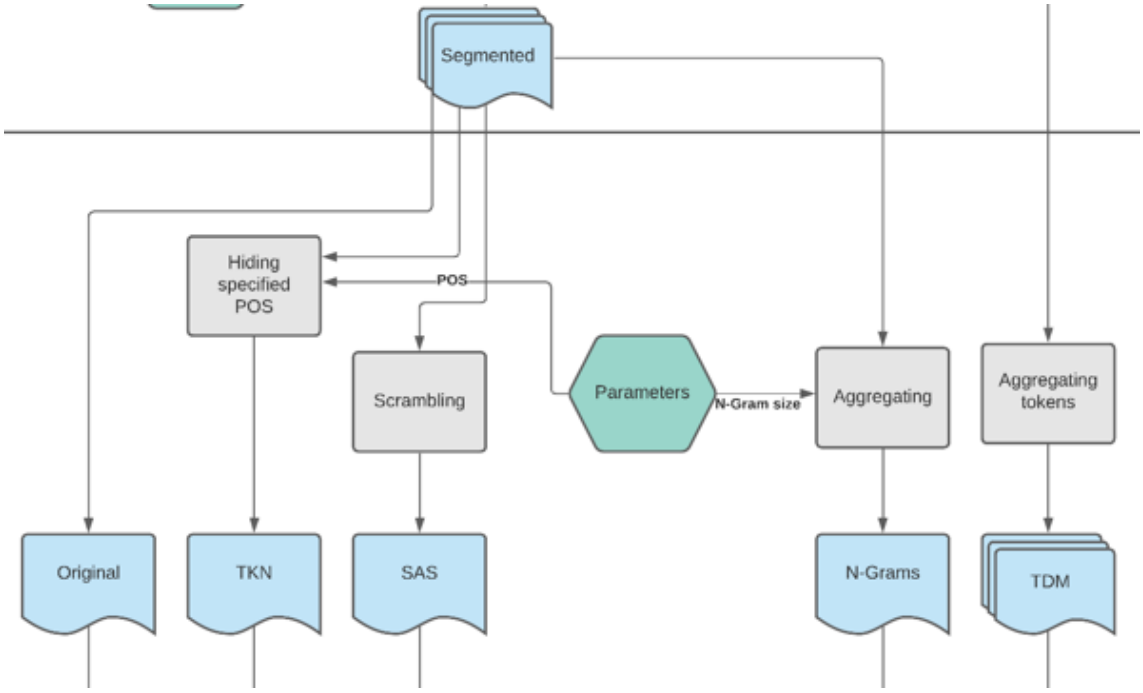


Abbildung 8: Der Formatting-Prozess

Abbildung 8 stellt die Swimlane *Formatting* vor. Damit kann sich eine Übersicht über die Prozessschritte verschafft werden und sie zeigt, wie sich die Formate voneinander unterscheiden.

Auf der linken Seite befinden sich neben dem Original die textähnlichen Formate *Segmentweise Aufhebung der Sequenzinformationen* (SAS) und *Selektiv reduzierte Information über einzelne Tokens* (TKN). Rechtsseitig sind die tabellarischen Formate *N-Gramme* (N-Grams) und *Term-Dokument-Matrix* zu sehen. Mittig sind die Parameter im grünen Hexagon symbolisiert. Von ihm gehen mehrere beschriftete Pfeile zu diversen Prozessschritten aus. Es ist zu sehen, dass alle Formate bis auf die *Term-Dokument-Matrix* (TDM) aus den zuvor erstellten Segmenten generiert werden. Das liegt daran, dass sie nicht auf Subdokumentebene erstellt werden soll, sondern auf Basis ganzer Texte.[26]

[26] Ursprünglich wurden die N-Gramme auch aus den Gesamtdokumenten generiert, doch wurde dieser Ansatz zugunsten der kleineren Segmente verworfen. Die Hoffnung bestand, dass damit die N-Gramme durch die Beibehaltung der Segmentinformation nützlicher sein würden.

4.4.1 Die Ordner- und Kontrollstrukturen

Für die Pipeline wurde eine stufige Ordnerstruktur erstellt:

- `0_source`. Dieser Ordner enthält den nicht-annotierten Korpus in Form von Textdateien für jeden einzelnen Roman.
- `1_tagged`. Das Skript `formats0_tagging.py` erstellt für jedes Token des Originals eine Zeile mit POS-Angaben und ihrer Grundform.
- `2_segmented` enthält Textdateien, die das Resultat des segmentierten getaggten Textes darstellen.
- `3_formats` beinhaltet die von den `formatting`-Skripten erstellten Textformate.
- `4_plain` ist der Ort für die aus den Textformaten erstellten Pseudotexte.
- `5_corpus`. Darin befinden sich die für das TM vorverarbeiteten Texte. Aus diesen werden schließlich die Topics abgeleitet.
- `6_evaluation` enthält diverse Daten für die Evaluation: Modelle, Kohärenzen und Plots.

Ab dem Ordner `3_formats` sind Unterordner pro erstelltem Textformat und Segmentlänge enthalten. Die Unterordnernamen bestehen aus zwei Teilen, z.B. `seglen-500`. Dabei dient `seglen` als Identifizierung, dass es sich um die Segmentlänge handelt und `500` ist die Segmentgröße, mit die darin befindlichen Textdateien erstellt wurden. Die Ausnahme bildet die TDM, weil die ihr zugehörigen Dateien zuerst in einem eigenen Ordner namens `tdm` abgelegt werden. Das liegt daran, dass die TDM nicht aus einer Segmentdatei erstellt wird, sondern aus der Tagged-Datei. In den darauffolgenden Ordnerstufen wird aber auch dieses Format in den jeweiligen Segmentordnern abgelegt, weil es bei der Transformation zur Textform ebenfalls in die jeweiligen Segmente unterteilt wird.

Angelehnt an Schöchs Skript `run_formats.py`, dient `run_preprocess.py` als Steuerung für die Verarbeitungsskripte. Unter `Call imported scripts` werden sie eingebunden und können durch Auskommentierung, z.B. mittels einer Raute (`#`) am Zeilenanfang, aktiviert bzw. deaktiviert werden. Als Argumente erhalten sie die ihnen zugehörigen Ordner (s.o.), deren Pfade relativ zum aktuellen Projektverzeichnis

im Abschnitt `Files and folders` gesetzt werden. Es kann durch Parametrisierung gesteuert werden, welche Formate eingelesen werden.

```
1 def check_inputfile(path, taggedfile):
2     if not path.is_file():
3         print("Segmentfile not found. Creating segmentfile ", params['
              seglen'])
4         formats0_segmenting.main(taggedfile, input_segmentsfolder,
              params)
5     return path
```

Code 11: Hilfsfunktion zur Prüfung von Segmentdateien

Um zu garantieren, dass für jede gewählte Segmentgröße die passende Quelldatei zur Verfügung steht, wurde die Hilfsfunktion `check_inputfile()` (Code 11 in `run_formats.py` eingefügt. Sie prüft, ob die Segmentdatei existiert (Z. 2) und ruft das Segmentierungsskript auf, falls dem nicht so ist. Auf diese Weise wird beim Aufruf des Skriptes immer ein Format mit der richtigen Segmentgröße erzeugt und, was es für Nutzende benutzerfreundlicher macht.

4.4.2 Parameter

Die Parameter werden in der Datei `parameters.py` definiert. Dort befinden sich mehrere vordefinierte Variablen die von Benutzenden befüllt werden müssen, welche dann im Dictionary `params` zusammengefasst und an die `run`-Skripte weitergegeben werden.

In der Variable `seglen` wird die Segmentlänge als Integer bestimmt. Sie ist relevant für alle Prozessschritte die das Label *Segmenting* im Flowchart tragen und für die Dateibenamung. Jede segmentierte Datei erhält zur eindeutigen Identifikation die Segmentlänge als Teil des Namens.

Der Parameter `casing` bestimmt, ob Wörter ihre Groß- und Kleinschreibung behalten (`'original'`) oder ausschließlich kleingeschrieben sein sollen (`'lower'`). Gerade für englische Texte erscheint diese Option nicht besonders relevant, könnte für das Deutsche interessant sein.

Mit `token` wird bestimmt welcher Teil der Features verwendet werden. Dieser Pa-

rameter ist interessant für das Format mit selektiver Aufhebung der Tokens und N-Gramme.

Parameter `pos` ist ein inkludierender Filter, der bestimmt, welche Wortarten im Text bleiben sollen. Alle nicht hier definierten Tags werden entfernt.

Mit `ngram` wird die Länge der N-Gramme im Textformat N-Gramme bestimmt.

Das Dictionary wird von den Kontrollskripten `run_formats` und `run_preprocess` importiert und jeweils an die dort aufgerufenen Skripte weitergegeben. So teilen sich alle Verarbeitungsskripte die gleichen Parameter, was die Erstellung und Weiterverarbeitung der Formate erleichtern soll.

Alle Formatting-Skripte lesen Dokumente zeilenweise aus der jeweiligen Segmentdatei (oder aus der Tagged-Datei) aus, die auf ihre spezifische Weise verarbeitet und anschließend an unterschiedlichen Orten gespeichert werden.

4.4.3 Selektiv reduzierte Information über einzelne Tokens

Dieses Format wird mit dem Skript `formats1_tkn` generiert. Relevant sind die Parameter `token` und `pos`. Die erlaubten Werte für `token` sind `'lemma'`, `'pos'` und `'mixed'`. Wurde der erste Wert bestimmt, werden alle Lemmata zurückgegeben, beim zweiten nur die Wortform und beim letzten wird gefiltert. Dafür wird ausgelesen, welche Wortformen in `pos` erlaubt wurden und nur diese werden als gesamtes Feature in das Format übernommen. Alle restlichen Token werden stattdessen als POS-Tag in den String geschrieben. Dafür bietet sich ein Switch in Form von aufeinanderfolgenden If-Else-Statements an, die testen, welcher Parameter für die Wortartfilterung ausgewählt wurde. Für TM erscheint nur die gemischte Version sinnvoll, da somit sichergestellt ist, dass die semantisch wertvollen Wortformen erhalten bleiben, aber der Text als solcher nicht mehr rekonstruierbar wird.

Die Auswahl des richtigen Teilfeatures geschieht durch Splitting des Feature-Strings am Unterstrich:

```
1 if len(token.split("_")) == 3:
2         if token.split("_")[1] in params["pos"]:
3                 features.append(token)
4         else:
5                 features.append(token.split("_")[1])
```

Code 12: Feature Splitting

In Code 12 ist zu sehen, wie gefiltert wird. Als Eingabe wird beispielhaft `'window_NN_window'` definiert. Wird dieser zerlegt entsteht die Liste `['window', 'NN', 'window']`. In Z. 1 wird zunächst geprüft, ob sich der eingegebene String `token` auch in drei Teile zerlegen lässt. Damit wird vermieden, dass fehlerhafte Features aus der Tagged-Datei übernommen werden. Es zeigte sich, dass an Stellen, an denen der Text nicht richtig gesäubert werden konnte, der Treetagger vom üblichen Tagging abweicht. Dies geschieht beispielsweise, wenn der Tagger fälschlich ein Domain Name System (DNS) erkennt, weil Wörter nicht richtig von einem Punkt abgetrennt sind, z.B. wenn am Satzende `'refresh.Anne'` geschrieben steht. Wenn der Tagger dies als Domain erkennt, wird stattdessen das Feature mit `'replaced-dns_NNS_replaced-dns'` ersetzt. Dahinter wird ein XML-Tag mit dem ersetzten Tag als Attribut gesetzt `<repdns text='refresh.Anne'/>`. Diese Information ist nicht zu gebrauchen und der Tag wird durch die Prüfung in Z. 1 ignoriert. Das hat aber auch zur Folge, dass `'replaced-dns_NNS_replaced-dns'` nicht herausgefiltert wird[27]. Der String wird daher als Stoppwort aufgenommen und verbleibt für das TM nicht mehr im Text. Dass überhaupt solche Fehler auftauchen, liegt vermutlich am Tagging der XML-Datei. Eine Vielzahl dieser Fehler konnte mit einfachen Regex-Ausdrücken korrigiert werden. Daher sei an dieser Stelle nochmal auf die Bedeutung des Prozesses *Revision* in Abbildung 6 für die Erstellung möglichst guter TM verwiesen.

In Z. 2 wird geprüft, ob der zweite Teil des Splits Teil der Wortformmenge ist. Es wird angenommen, dass dem so ist, also wird der ganze Feature-String der Ausgabeliste

[27] Dieses Verhalten kann im Treetagger ausgeschaltet werden. Sieht man in den Code der Funktion `tag_text()`, sieht man dass das Argument `notagdns` den Default-Wert `False` hat. Wird dieser stattdessen auf `True` gesetzt, wird das im Haupttext beschriebene Verhalten nicht ausgelöst. Es wird dann die vermeintliche Domain als einzelnes Token erkannt. Dieses Verhalten ist aber in diesem Szenario sinnlos, da dann ohnehin falsch getaggt und der Fehler eventuell nicht erkannt wird.

hinzugefügt. Im anderen Fall, würde nur die Wortform (`NN`) aufgenommen werden. Diese Liste wird als String mit `' '.join(features)` zusammengesetzt und erhält damit das fertige Format, das abgespeichert werden kann.

Der Name der fertigen Datei beginnt mit dem Formatsprefix gefolgt von einem Bindestrich. Dahinter folgen die definierten Wortartkürzel in alphabetischer Reihenfolge, welche mit je einem Unterstrich getrennt sind. Der fertige Dateiname kann beispielsweise diese Form annehmen: `tkn-NN_NNS.txt`. Es handelt sich also um das Format, bei dem nur noch die Substantive erkenntlich sind. Nachteil bei diesem Vorgehen ist, dass der Dateiname recht lang ausfallen kann, aber dadurch gestaltet sich im späteren Verlauf der Pipeline die Filterung nach dem POS-Set sehr einfach.

4.4.4 Segmentweise Aufhebung der Sequenzinformation

Das Format wird vom Skript `formats2_src.py` generiert und abgespeichert.

Ursprünglich beinhaltete Schöchs Skript den Segmentierungsprozess in dieser Datei, doch wurde dieser wie oben beschrieben in einen eigenen Schritt ausgelagert. Zentral ist daher der Randomisierungsschritt der Token innerhalb eines Segmentes, was mit der Methode `shuffle()` aus der nativen Python-Bibliothek `random` realisiert werden kann:

```
1 import random
2 random.shuffle(seg) # scrambling
3 seg = " ".join(seg)
```

Code 13: Randomisierung der Segmente

`shuffle()` erhält als Argument die zu randomisierende Segmentliste `seg` und speichert das Ergebnis in einem wieder zusammengesetzten String.

Hiermit ist das Format schon erstellt und es benötigt hiernach keine weitere Vorverarbeitung mehr, um als Pseudotext im Ordner `4_plain` gespeichert zu werden.

4.4.5 Term-Dokument-Matrix

Zur Generierung dieses Formates wird das Skript `formats_frq` ausgeführt. Als Parameter erhält es mit `casing` lediglich die Information, ob Groß- und Kleinschreibung beachtet werden soll. Wird der Wert `'lower'` eingetragen, werden alle Wörter kleingeschrieben, bei `'original'` hingegen, bleibt das Casing unverändert. Für TM ist das Casing irrelevant, daher wird dieses Format standardmäßig im Modus `'lower'` ausgeführt. Es könnte im Englischen für Lesende als visueller Hinweis auf Namen dienen und bei deutschen Texten die manuelle Suche nach Substantiven erleichtern, ist darüber hinaus aber uninteressant.

Zentraler Vorverarbeitungsschritt ist die Aggregierung der Tokens. Das Skript liest die Tagged-Datei aus und generiert mit der Funktion `get_counts(features)` das Format:

```
1 import pandas as pd
2 def get_counts(features):
3         counts = pd.Series(Counter(features))
4         counts.sort_values(inplace=True, ascending=False)
```

Code 14: Aggregierung der Token für TDM

Mit Code 14 wird die externe Bibliothek `pandas` (McKinney, 2010; The pandas development team, 2020) importiert. Diese kann beispielsweise mit über das Python Package Index mit `pip install pandas` oder aus einem anderen Python-Repository (z.B. conda-forge) installiert werden. Es handelt sich um eine sehr mächtige Bibliothek zum Erstellen und Bearbeiten von Datensätzen, meist in Form von Tabellen. In Zeile 3 wird aus der Eingabeliste, den Features, ein Series-Objekt erstellt, dass aus einer Menge von Python-Tuples besteht. Diese beinhalten jeweils zwei Elemente: Zum einen das Feature und zum anderen dessen Häufigkeit (z.B. `('the_DT_the', 2317)`). Diese werden dann nach Letzterem sortiert (Z. 4) und in einer Datei tabulatorsepariert gespeichert.

4.4.6 N-Gramme

Das Skript `formats_ngr.py` erstellt N-Gramme auf Basis eines Segmentes und fügt sie in einer Datei zusammen. Ursprünglich beinhaltete das Skript keine Option,

Token-Features als Ganzes beizubehalten. Stattdessen war es nur möglich ihre Einzelteile auszugeben (Lemma, Wortart oder Token). Daher wurde in der Funktion `select_features` die Unterstützung für den `token`-Parameter für den Wert `mixed` hinzugefügt. Das sorgt dafür, dass das gesamte Feature beibehalten wird. Wichtig ist, dass, damit jedes Format auf die gleiche Weise vom Vorverarbeitungsschritt am Ende der Pipeline bearbeitet werden kann.

Die Aggregation findet in der Funktion `create_ngrams()` statt.

```
1 def create_ngrams(ngrams, params):
2     ngrams = zip(*[ngrams[i:] for i in range(params["ngram"])])
3     ngrams = [" ".join(ngram) for ngram in ngrams]
```

Code 15: Aggregierung der Token für N-Gramme

In Z. 2 von Code 15 wird mit `zip()` ein Iterator gepackt, welches N-Gramme der Größe `params['ngram']` als Tuple ausgibt. Als Input dient die Liste `ngrams`, die die gefilterten Lemmata enthält. Dann wird jedes Tuple als ein N-Gramm als String-Element in einer Liste zusammengefügt (Z. 3).

So entstehen alle N-Gramme für dieses Segment. Mit der Funktion `count_allngrams` werden diese gezählt und zusammengefasst.

```
1 def count_allngrams(allngrams, params):
2     from collections import Counter
3     allcounts = dict(Counter(allngrams))
4     filteredcounts = dict()
5     for (key, value) in allcounts.items():
6         if value > params["threshold"]:
7             filteredcounts[key] = value
```

Code 16: Aggregierung der Token für N-Gramme auf Korpusebene aus Einzeltexten

In Code 16 wird in Z. 2 `Counter` importiert. Er nimmt als Argument eine Liste der N-Gramme auf Segmentbasis, addiert gleiche Vorkommen auf und speichert sie als `dict()` mit dem String als `key` und die Summe als `value` ab (Z. 3). Anschließend wird durch jedes Key-Value-Paar iteriert und geprüft, um der die Summe größer ist als der in `'threshold'` definierte Integer. So werden nur N-Gramme übernommen, die das Mindestvorkommen erfüllen (Z. 5-7).

Sie können dann gespeichert werden. Dafür werden die Segmente im CSV-Format in die Datei geschrieben, auf den der Separator-String `<seg>` folgt. Damit sind alle Segmente in einer zusammengefassten Datei gespeichert und trotzdem voneinander unterscheidbar.

4.5 Preprocessing

Das Preprocessing ist der Schritt, bei dem die Formate in Textstrukturen überführt werden. In diesem Kapitel wird beschrieben, welche Schritte für die jeweiligen, aus den Formaten generierten, Pseudotexte durchgeführt wurden, um einen validen Input für das TM zu generieren.

In Abbildung 9 ist schematisch die Übersicht über den gesamten Preprocessing-Prozess gegeben. Die Swimlane *Preprocessing* schließt sich nahtlos an *Formatting* an. Die dort dargestellte Pfeile führen von den Formatdateien weg. Es ist zu erkennen, dass die linksseitigen Formate weniger vorverarbeitet werden müssen, als die tabellarischen Formate N-Gramme und TDM auf der rechten Seite.

Für den Preprocess wurden drei Skripte erstellt, die die Eigenheiten der Formate beachten:

- `seg2plain.py` für den originalen Volltext der Segmente.
- `tkn2plain.py` für die Selektiv reduzierte Information über Tokens
- `frq2plain.py` für die Term-Dokument-Matrix
- `ngrams2plain.py` für die N-Gramme

Für das SAS-Format wird keine spezifische Vorverarbeitung mehr benötigt, es speist sich unmittelbar in die generelle Vorverarbeitung. Die letzten beiden Subprozesse *Lemmatizing/Filtering POS* und *Removing Stopwords* werden in `preprocess.py` durchgeführt. Hierzu ist keine weitere Formatsdifferenzierung notwendig.

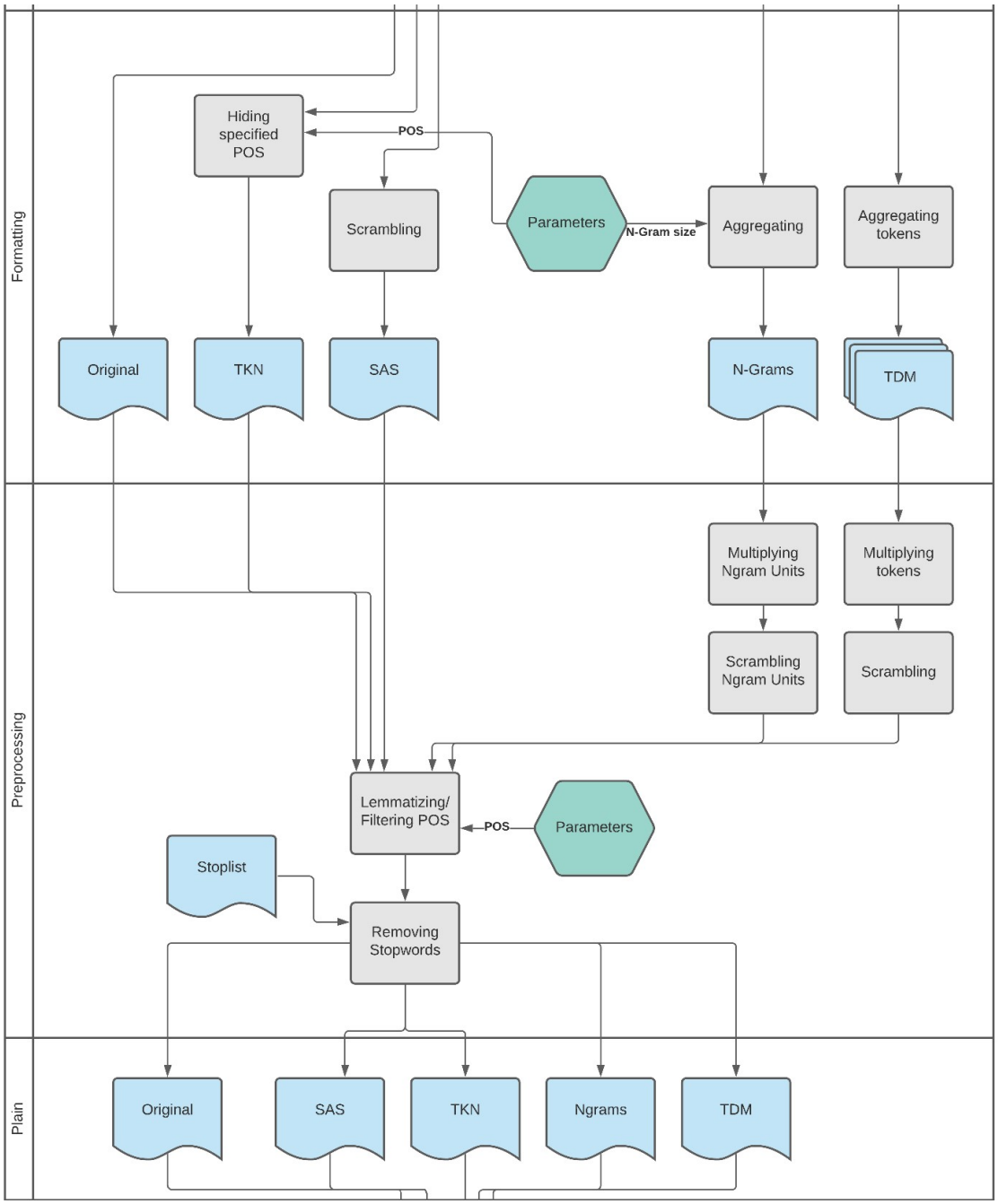


Abbildung 9: Das Preprocessing

4.5.1 Das Original und Selektiv reduzierte Information über einzelne Token

Das Skript `seg2plain.py` ist trivial, denn alles was nötig ist, um eine Segmentdatei zu bereinigen, ist das Entfernen des Quelldateinamens. Damit ist das Originalformat schon bereit für die allgemeinen Vorverarbeitungsschritte.

Das Skript `tkn2plain.py` funktioniert auf die gleiche Weise. Es unterscheidet sich dadurch, dass es beim Aufruf in `run_preprocess.py` den Formatsordner `3_formats` statt einer Segmentdatei erhält und ein Filter eingebaut ist, der im Namen der dort

enthaltenen Dateien das Präfix `tkn` sucht.

4.5.2 Term-Dokument-Matrix

Die TDM stellt eine größere Herausforderung da, weil sie keine Textstruktur besitzt. Als Input dienen mehrere Textdateien, die je eine Matrix beinhalten. Zwischen jeder Inhaltszeile befindet sich eine leere Zeile, die durch die Funktion `to_csv()` im Skript `formats2_frq` erstellt wurden. Sie erstellt aus einem `pandas`-Series-Objekt Zeilen mit einem Separator, in diesem Fall Tabulator. Nach jeder Zeile wird jedoch eine Leerzeile eingefügt, die beim Einlesen wieder entfernt werden sollte.

```
1 file.read().split("\n\n")
```

Code 17: Einlesen einer Term-Dokument-Matrix

Der Code 17 erhält die TDM-Datei als Argument, liest den Inhalt aus und gibt eine Liste mit dem jeweiligen Zeileninhalt zurück. Dieser kann beispielsweise so aussehen: `'a_DT_a\t961'`.

Mit dieser Information kann also der Feature-String durch Multiplikation vervielfältigt werden.

```
1 def create_text(lines):
2     text = []
3     for l in lines:
4         if len(l.split("\t")) == 2:
5             tkn_info = l.split("\t")[0]
6             count = int(l.split("\t")[1])
7             for i in range(count):
8                 text.append(tkn_info)
9     return text
```

Code 18: Vervielfältigung der Features aus einer Term-Dokument-Matrix

In Code 18 ist die Funktion `create_text()` abgebildet. Als Argument nimmt sie eine oder mehrere Zeilen wie oben beschrieben. Sie prüft, ob die Zeile das richtige Format hat, indem sie am Tabulator geteilt wird und zählt, ob die Summe der Teile `2` ist (Z. 4). Es speichert das Feature in einem String und wandelt die Häufigkeit in einen Integer um (Z. 5 und 6). In einer Schleife wird der Feature-String so oft in

eine Liste gespeichert, wie es durch die Häufigkeit definiert ist (Z. 7f). Dies kann je nach Größe des Korpus einige Zeit in Anspruch nehmen.

Anschließend ist die Liste bereit für das zufällige Vertauschen ihrer Elemente. Dies wird mit der Funktion aus Code 13 in Kapitel 4.4.4 mit `shuffle` realisiert. Es entsteht ein Text, dessen Inhalt keinerlei semantische Kohärenz besitzen kann. Als letztes wird dieser Text auf bereits bekannte Art segmentiert, bevor er in den generellen Workflow eingespeist wird.

4.5.3 N-Gramme

Das Vorgehen bei N-Grammen ist dem der TDM ähnlich, da es sich ebenfalls um Matrizen handelt, dessen Werte miteinander multipliziert werden müssen. Daher ist die Strategie die gleiche: Multipliziere den String aus einer Zeile mit seinem Vorkommen und vertausche sie zufällig als N-Gramme.

Dafür muss die N-Gramm-Datei zunächst mit der Funktion `read_file` segmentweise eingelesen werden.

```
1 def read_file(file):
2     content = file.read()
3     content = cleaning(content)  # hacky cleaning
4     segments = []
5     for segment in content:
6         segment = segment.split("\n")
7         if len(segment) > 1:
8             segments.append(segment)
9     return segments
```

Code 19: Einlesen einer N-Gramm-Datei

Code 19 zeigt die einlesende Funktion. Sie erhält als Argument die einzulesende Datei `file`. Als erstes wird der Text von seinen Leerzeilen bereinigt und anschließend an den Separator-Strings `<seg>` aufgeteilt und die so entstandenen Teile jeweils wieder als String zusammengefügt. So entsteht eine Segmentliste, welche bereits jedes Segment als String enthält. Dies geschieht in der Funktion `cleaning`[28] (Z. 3), welche als Argument den aus `file` eingelesenen Inhalt (Z. 2) als String erhält. Über diese

[28] Das Vorgehen ist nicht elegant, aber pragmatisch gelöst.

Liste wird iteriert und in den jeweiligen Strings an Zeilenumbrüchen aufgetrennt (Z. 5f), wodurch man eine Liste aus N-Gramm-Zeilen mit ihren Häufigkeiten erhält. Wenn diese Liste mindestens einen Eintrag enthält, wird sie Teil der übergeordneten Segmentliste. Die übergeordnete Segmentliste enthält damit die N-Gramme mit ihren Häufigkeiten, die nach Segmenten gruppiert sind.

```
1 def create_text(seg):
2     text = []
3     for line in seg:
4         if len(line.split("\t")) == 2:
5             tkn_info = line.split("\t")[0]
6             count = int(line.split("\t")[1])
7             for i in range(count):
8                 text.append(tkn_info)
9     return text
```

Code 20: Generierung eines Textes aus N-Grammen

Anschließend wird diese Liste der Funktion `create_text` (Code 20) übergeben. Dort wird über alle Listenelemente iteriert und am Tabulator aufgeteilt. Wenn die daraus entstandene Liste auch tatsächlich aus 2 Elementen besteht, kann der N-Gramm-String mit der Häufigkeit multipliziert werden und anschließend das Ergebnis gespeichert und zurückgegegen werden. Die so entstandenen Texte werden anschließend gespeichert.

4.5.4 Generelles Preprocessing

Das Skript `preprocess.py` führt die letzten Preprocessing-Schritte aus und ist dabei unabhängig vom Input-Format. Die zuvor erstellten Textsegmente werden aus ihren Dateien zeilenweise eingelesen, verarbeitet und dann in einem Stream (kontinuierlich) in die Zieldatei geschrieben. Die Texte werden so aufbereitet, dass die Filterung nach Wortarten, die Lemmatisierung und das Entfernen von Stoppwörtern für alle möglich ist. Um Stoppwörter zuverlässig identifizieren zu können, müssen lemmatisierte Formen der Wörter zur Verfügung stehen.

```
1 def lemmatize(seg, params):
2     pos = params["pos"]
3     for s in seg.split(" "):
4         components = re.split("_", s)
5         if len(components) != 3:
6             pass
7         elif (components[1].upper() in pos):
8             # yield lemma
9             yield clean_token(components[2].lower())
```

Code 21: Filterung nach Wortarten und anschließende Lemmatisierung

In Code 21 ist die Funktion `lemmatize()` dargestellt. Als Input erhält sie ein Textsegment, welches durch seine Features iteriert (Z. 3). Das Feature wird in seine Einzelteile in eine Liste zerlegt, indem es an seinen Unterstrichen aufgeteilt wird (Z. 4). Wie bereits in Kapitel 4.4.3 gezeigt, kann die Funktion auf Ausnahmen stoßen. Deswegen wird in Z. 5 darauf geprüft, ob die aktuelle Liste drei Strings enthält , um diese, falls nicht, zu ignorieren (Z. 6). Hat es eine Liste durch die Tests geschafft, wird ihr Lemma mittels `yield`[29] zurückgegeben (Z. 7-9).

[29] Pythons `yield`-Keyword erzeugt ein Generator-Objekt, welches im Verlauf eines Skriptes wie ein Iterator aufgerufen wird und als Alternative für sonst übliche `return`-Statements genutzt werden kann. Eine Generator-Funktion erzeugt Output aus allen `yield`-Statements, die im Code benutzt werden. Der Vorteil dieser Funktionen besteht darin, dass Code deutlich kompakter geschrieben werden kann, da nicht erst iterierbare Objekte, z.B. eine Liste, erstellt werden müssen und dann als ein Wert übergeben werden. Üblicherweise geschieht dies durch Initialisierung eines leeren Listenobjektes, die durch eine Schleife nach und nach mit Werten befüllt wird und dann als Ganzes weiterverarbeitet wird, so wie es häufig in den hier vorgestellten Skripten zur Formatgenerierung getan wird. Das kann bei großen Datensätzen zu Speicherproblemen führen, da ein großes Objekt generiert wird. Eine Generator-Funktion hingegen erzeugt die Werte erst beim Aufruf und übergibt sie sofort ohne Zwischenspeicherung, was effizienter ist und die Berechnungszeit deutlich verkürzen kann. Die Zeitersparnis zeigte sich bereits in den ersten Versuchen zur Erstellung von Texten aus Matrizen aus Schöchs Formatskript `formats3_tdm.py`, das TDMs auf Subkorpusebene erstellt. Die Ersparnis durch die Generator-Funktionen ist im Vergleich zu den klassischen `return`-Statements enorm. Zwar nehmen die dortigen Berechnungen immer noch viel Zeit in Anspruch, allerdings konnte die Bearbeitungszeit pro Textdatei um mehrere Minuten verkürzt werden. Es handelt sich um das Skript `csv_to_plain.py`, welches zugunsten einfacher strukturierter Skripte nicht weiter benutzt wurde. Es ist aber trotzdem noch im Repository aufzufinden, damit sich die Lesenden selbst ein Bild davon verschaffen können.

```
1 def remove_stopwords(lemmas, stoplist):
2     with stoplist.open("r") as s:
3         stoplist = s.read().splitlines()
4     for l in lemmas:
5         if (l.lower() not in stoplist):
6             yield l
```

Code 22: Funktion zur Entfernung von Stoppwörtern

Die Lemmata werden dann der Funktion `remove_stopwords()` (Code 22) übergeben. Diese Funktion erhält zusätzlich eine Stoppwortliste in Form einer Textdatei als Argument. Sie ist zu finden in `resources/stoplist.txt` und enthält pro Zeile je ein Stoppwort und kann durch zeilenweises Hinzufügen beliebig von Nutzenden erweitert werden. In dieser Arbeit wurden Stoppwörter manuell eingefügt, die bei der Erstellung erster Topics aufgefallen sind. Sie wurde um eine in einem Blog-Post erzeugte Stoppwortliste speziell für die Sprache englischer Literatur aus dem 19. Jahrhundert erweitert (Jockers, n. d.). Sie umfasst 3346 Wörter, worunter sich übliche Englische hochfrequentierte Wörter und mehrere tausend Namen befinden. Die Funktion liest die Liste zeilenweise ein und speichert sie in einer Liste (Z. 2f). In einer Schleife wird jedes Lemma mit jedem Stoppwort in der Liste abgeglichen und als Stream zurückgegeben, sofern sie nicht darin enthalten ist (Z. 4ff). Dieser Stream wird in `write_to_file()` geöffnet und die gefilterten Lemmata mit Leerzeichen getrennt in eine Datei geschrieben. Sobald ein Segment vollständig abgeschlossen ist, wird ein Zeilenumbruch eingefügt, die Datei erneut geöffnet und das nächste Segment erstellt, bis alle Segmente erzeugt wurden. Die erstellte Zieldatei enthält nun Segmente mit potenziell relevanten Lemmata und ist für TM bereit.

4.6 Topic Modeling

Für das Topic Modeling wurden zwei Wege implementiert, wie in Abbildung 10 zu sehen ist. Dies wurde getan, da nicht abzusehen war, welcher Weg sich für die Arbeit besser eignet. Deshalb werden in den folgenden Kapiteln beide Methoden, Gensim und MALLETs Kommandozeile, beschrieben.

4.6.1 Topic Modeling mit Gensim

Das Ziel dieses Pipeline-Teils ist es, mehrere Modelle zu erstellen und sie als Dateien im System zu persistieren, um sie für verschiedene Visualisierungen und Untersuchungen abrufbar zu halten.[30]

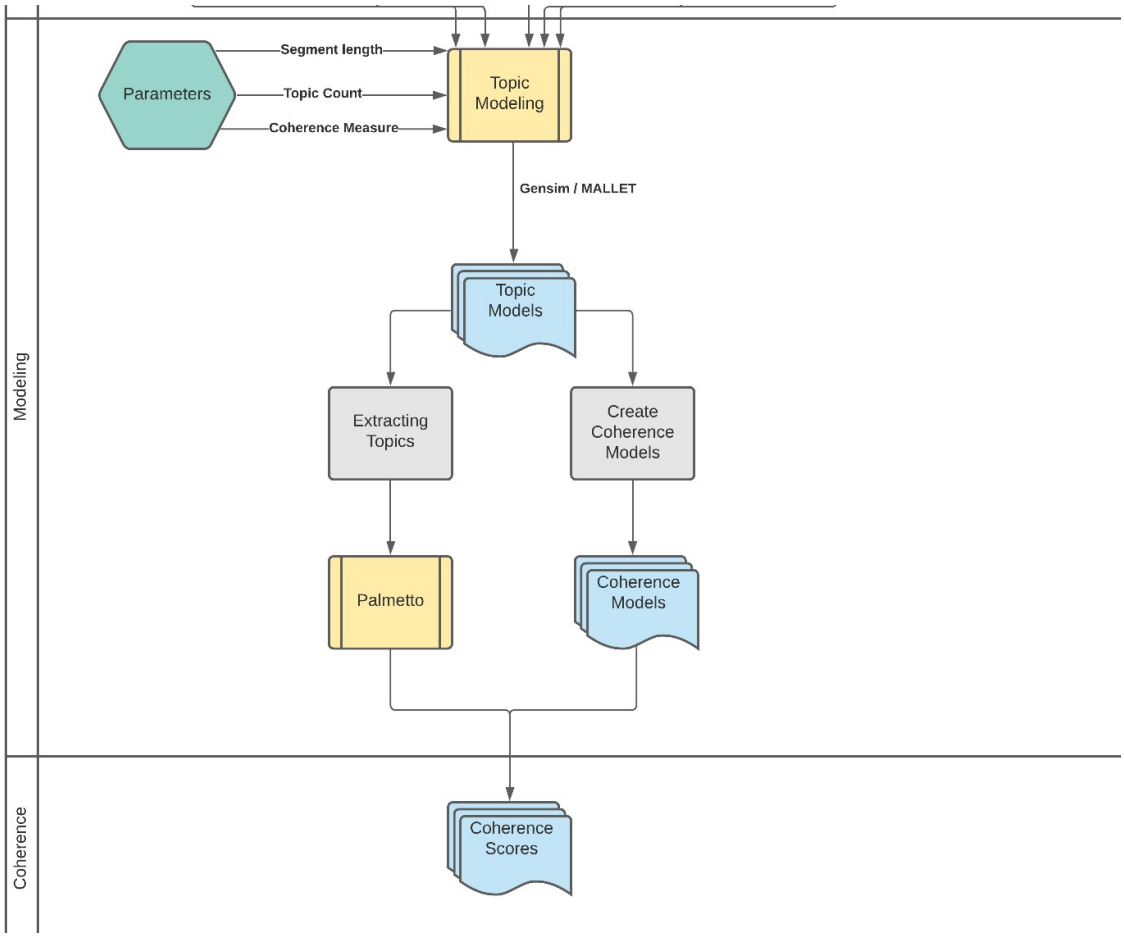


Abbildung 10: Topic und Coherence Modeling

Wie in Kapitel 2.4.2 beschrieben, werden die TM-Funktionen von Gensim verwendet. Dieses Tool ermöglicht es, Topic und Coherence Modelle mit relativ wenig Aufwand zu erstellen. Gensim bietet viele verschiedene TM-Algorithmen an (vgl. Kapitel 2.1), wovon die beiden LDA-Varianten für die Untersuchung angewandt werden können.

Im TM-Prozess werden die Modelle in Form von verschiedenartigen Objekten realisiert, die im Modul `gensim.models` einzusehen sind. Gensims eigene LDA-Modelle heißen `ldamodel` bzw. `ldamulticore`. Letzteres enthält die Funktionalität des Basis-

[30] Wie sich herausstellt, wird Gensims Topic Modeling nicht primär für die Evaluation verwendet. Stattdessen werden sie hauptsächlich über das Kommandozeilen-Tool von MALLET erstellt (siehe folgendes Kapitel). Daher enthält das hierfür verwendete Skript auch experimentelle und ungenutzte Funktionen.

models, erhält im Konstruktor aber zusätzlich das Argument `workers`, welches die Anzahl der zu verwenden Prozessorkerne bestimmt. Laut Gensim-Dokumentation soll die damit einhergehende Parallelisierung der Prozesse die Rechenzeit beim Trainieren des Models erheblich verkürzen (vgl. Řehůřek, 2021). In eigenen Tests hat sich diese Behauptung bestätigt und die Rechenzeit hat sich um bis zu 50% verringert[31]. Die Berechnung besserer Topics in Form von MALLET-Modellen hingegen benötigt erheblich länger. Während ein Multicore-Model zur TM-Erstellung dreier Textformate mit Segmentlänge 500 nur ca. 55 Sekunden benötigte, musste man auf MALLET mit den gleichen Parametern etwas über 500 Sekunden warten. Das ist eine Zeitersparnis von etwa 90%.

Im Laufe der Evaluation wurde die Möglichkeit entdeckt, auch MALLET auf mehreren Threads trainieren zu lassen, was im Kommandozeilen-Tool über das Argument `num-threads`, als auch über Gensims Wrapper mit dem Argument `workers` eingestellt werden kann. Auf diese Weise verringert sich MALLETs Trainingszeit um etwa 50%, was eine gravierende Verbesserung darstellt und daher auch stattdessen genutzt wird. Ursprünglich wurde nur der Wrapper implementiert. Erstens aus dem Grund, dass die Zeitersparnis so deutlich war und damit hunderte Modelle recht schnell trainiert werden konnten. Zweitens erschien die Idee, Gensims Modelle, die als Binärdateien abgespeichert werden, im späteren Verlauf zu laden und zu benutzen, als zielführend. Doch es stellte sich heraus, dass das Speichern und Laden der Gensim Modelle ebenfalls zeitintensiv und der Speicherplatzbedarf hoch ist.

Die schlechteren Topics sollten in diesem Szenario keinen besonderen Einfluss auf die Evaluation haben, sofern die Modelle untereinander vergleichbar bleiben. Trotzdem ist es interessant zu überprüfen, ob die unterschiedlichen LDA-Implementationen einen Einfluss auf die Nutzbarkeit der Formate haben.

Implementiert wurden die TM-Prozesse in Form eines IPython Jupyter-Notebooks in der Datei `topic_modeling.ipynb` für Gensim und im Standard-Python-Programm `run_mallet.py`. Für die Evaluation bietet IPython die angenehme Eigenschaft, dass Tabellen und Grafiken direkt darin erzeugt und angezeigt werden, ohne dass es nötig wäre, sie lokal als Datei abzuspeichern. Der Jupyter-Server wurde lokal über die

[31] Getestet wurde mit einem Intel i7-10750 mit sechs physischen Prozessorkernen.

Entwicklungsumgebung *Pycharm Professional* gehostet, was den Vorteil hat, dass dessen Funktionen benutzt werden können. Das hat auch zur Folge, dass die Parameter nicht in einer separaten Datei gesetzt werden, sondern innerhalb der gleichen Notebook-Datei. Die relevanten Parameter, die Nutzende spezifizieren müssen, sind `type`, `seglen`, `topic_count`, `coherence` und `measure`. Die Variable `type` erhält einen String mit Inhalt `'gensim'` oder `'mallet'` und bestimmt damit den LDA-Algorithmus. `seglen` bestimmt die Segmentlänge, welche vorprozessierten Textdateien aus dem Ordner `5_corpus` ausgelesen werden. `topic_count` wird dem TM-Prozess übergeben und bestimmt, wie viele Topics generiert erstellt werden und `measure` bestimmt das Kohärenzmaß, nach welchem das Coherence Modell erstellt wird. Gensim unterstützt die Maße `'u_mass'`, `'c_v'`, `'c_uci'` und `'c_npmi'`.

Für die Kategorisierung wurde eine eigene, tiefere Ordnerstruktur angelegt. Im Ordner `6_evaluation/models` wird zwischen `gensim` und `mallet` unterschieden. Sie enthalten jeweils Unterordner zur Strukturierung nach Segmentlänge und der Topic-Anzahl. Auf unterster Ebene werden Formats-Ordner erstellt, die die Topic und Coherence Modelle beinhalten. Ein solcher Pfad sieht für ein mit Gensim generiertes Modell mit Segmentlänge 500 und Topic-Anzahl 50 z.B. so aus: `6_evaluation/models/gensim/seglen-500/topics-50/tkn`. Die Struktur erlaubt es später bei der Auswertung gezielt nach Parametern zu suchen und die entsprechenden Modelle zu laden.

Für die TM-Erstellung muss aus einem Text zunächst ein Dictionary-Objekt erstellt werden, was u.a. ein `dict` generiert, das die Token auf ihre einzigartigen Integer mapped. Dafür wird der Konstruktur `Dictionary` aus dem `corpora`-Modul aufgerufen. Er erhält Texte als Argument, bestehend aus Listen auf Dokumentebene mit seinen Token als Elemente, die wiederum Elemente in einer zusammenfassenden Liste sind. Die Funktion `read_file()` liefert solch eine Liste zurück[32]. Das zweite benötigte Objekt ist `corpus`. Es ist ein bag-of-words-Modell, generiert mit der `Dictionary`-Funktion `doc2bow()`. Beide Objekte werden schließlich einem LDA-Konstruktor übergeben, was das Modell erstellt. Hierzu können sie der Funktion

[32] Gensim bietet im Module `utils` eine eigene Funktion namens `simple_preprocess()` an, die eine Datei so zerlegt, doch die Token werden dann noch durch eine kleine Vorverarbeitungspipeline geschickt, was die Rechenzeit spürbar erhöht. Da die Texte aber schon vorverarbeitet sind, bietet diese Funktion keinen Mehrwert.

`create_tm()` zusammen mit dem gewählten Model-Typen (`gensim` oder `mallet`) übergeben werden. Um das extern generierte MALLET-Modell in Gensim nutzen zu können, muss es in einem `wrapper` eingepackt werden. Der Aufruf sieht dann folgendermaßen aus:

```
1 gensim.models.wrappers.LdaMallet(mallet_path=MALLET_PATH, corpus=corpus
    , num_topics=topic_count, id2word=dictionary)
```

Code 23: Erstellung eines MALLET Topic Models mit Gensim Wrapper. Auszug aus `topic_modeling.py`.

Übergeben werden die Parameter `mallet_path` (Pfad zum lokalen MALLET-Verzeichnis), `corpus`, `dictionary` und `topic_count`. Das Gensim-Model wird mit `LdaMulticore()` respektive `LdaModel()` erstellt und erhält bei ersterem das Argument `workers`, was die Anzahl der zu nutzenden Prozessorkerne darstellt.

Gespeichert werden die Modelle mittels `save_tm_model()` im o.g. Pfad unter dem Namen `tm.bin`. Für Gensim-Varianten werden dafür mit der Funktion `LdaModel.save()` insgesamt vier Dateien erstellt, die jeweils andere Dateiendungen erhalten. Für MALLET-Modelle wird je nur eine Datei erstellt.

Coherence Modelle werden erstellt und gespeichert in `save_coherence_model()`. Der Konstruktor `CoherenceModel()` befindet sich im Modul `gensim.models.coherencemodel` und erhält als Argumente das TM und die Dokumentenliste[33]. Diese Objekte enthalten analog zu den Topic Models eine `save()`-Funktion zum Speichern. Der Dateiname besteht aus `coh_` und dem verwendeten Kohärenzmaß, z.B. `cv`, und die Dateien werden im gleichen Ordner wie die TM gespeichert. Wie sich im weiteren Verlauf der Auswertung zeigte, ist es nicht möglich direkt ein `CoherenceModel()` aus einem MALLET-Modell zu erstellen. Zuvor muss es mit der Funktion `malletmodel2ldamodel()` in ein Gensim-Modell konvertiert werden.

[33] Es gibt mehrere Möglichkeiten ein Kohärenzmodel zu erstellen. Die Argumente unterscheiden sich je nachdem, welche Kohärenzmaße verwendet werden sollen. Für die Verwendung der vektorbasierten Maße, dessen Namen mit `c_v` beginnen, sind die Dokumente notwendig.

4.6.2 Topic Modeling mit MALLET

Wie sich herausstellte, wurde kein Weg gefunden, Zugriff auf MALLETs Diagnose-XML (s. Kapitel 2.4.3) über die Gensim-Implementierung zu erhalten. Daher war es nötig, mit `run_mallet` ein Programm zu erstellen, dass Befehle über die System-Kommandozeile ausführt. Mit Pythons Modul `subprocess` kann das realisiert werden. Zunächst ist es notwendig zu verstehen, wie MALLET auf diese Weise bedient werden kann. Dafür muss im Terminal zum MALLET-Hauptverzeichnis navigiert werden (in diesem Fall `C:\mallet`) und der Befehl `bin\mallet` ausgeführt werden[34]. Führt man ihn aus, listet das Programm die möglichen Aktionen auf.

Im ersten Schritt muss die Textbasis importiert und dann in MALLETs eigenes Korpusformat umgewandelt werden. Diese Datei kann mit `import-dir` oder `import-file` erstellt werde. Da die Texte zeilenweise in den Dateien zusammengefasst sind, wird letzterer verwendet, weil MALLET dann zeilenweise importiert. Der erste Befehl wird verwendet, wenn jedes Dokument einzeln in Dateien gespeichert wird, denn dann wird jeder Text im Verzeichnis als Dokument geladen. Mit dem Argument `--input` wird der Quellpfad und mit `--output` der Ausgabepfad bestimmt. Mit `--keep-sequence` wird noch sichergestellt, dass die Dokumentreihenfolge erhalten bleibt.

```
# create MALLETs input files
for file in corpusdir.glob("*.txt"):
    output = mallet_dir.joinpath(f"{file.stem}.mallet")
    # doesn't need to happen more than once — usually.
    if output.is_file(): continue
    print(f"--{file.stem}")
    cmd = f"bin\\mallet import-file " \
          f"--input {file.absolute()} " \
          f"--output {output.absolute()} " \
          f"--keep-sequence"
    subprocess.call(cmd, cwd=MALLET_PATH, shell=True)
print("import finished")
```

Code 24: Erstellung des Import-Befehls für MALLETs Kommandozeile

Code 24 zeigt, wie ein solcher String erzeugt und ausgeführt werden kann. Für jede Input-Datei wird ein String erzeugt, der dynamisch Pfade setzt und in Zei-

[34] Dies gilt nur für Windows-Systeme. Auf Unix basierenden Systeme müssen entsprechend Backslashes durch die nach vorne geneigten ersetzt werden.

le 11 wird dieser dann `subprocess.call()` als erstes Argument übergeben. Das Argument `cwd` erhält den Pfad zum Hauptverzeichnis, womit dieser Befehl dort ausgeführt wird[35]. Das Resultat sind die MALLET-Dateien, welche im Verzeichnis `6_evaluation/models/mallet/seglen-x` unter ihren entsprechenden Namen, beispielsweise `original-500.mallet` abgespeichert werden. In Zeile 5 wird überprüft, ob die Datei bereits existiert und bricht den Vorgang ab, falls dem so sein sollte.

Um nun Modelle zu trainieren, wird für jede so erzeugte Datei der Befehl `train-topics` ausgeführt, welcher mit zahlreichen Parametern angereichert werden kann[36]. Also wird über jede Dateien iteriert und wieder ein String erstellt, der an `subprocess.call()` übergeben wird.

```
1 cmd = f"bin\\mallet train-topics " \
2           f"--input {file.absolute()} " \
3           f"--num-topics {topic_count} " \
4           f"--output-topic-keys {keysfile} " \
5           f"--diagnostics-file {diagnostics_xml} " \
6           f"--num-threads {num_threads}"
```

Code 25: Erstellung des Befehls für die TM-Erstellung für MALLETs Kommandozeile

Code 25 ist ein Ausschnitt einer solchen Iteration. Mit `--num_topics` wird bestimmt, wie viele Topics erstellt werden. Dieser Wert muss vom Nutzenden in der Variable gesetzt werden. Darauf Folgen die Output-Dateien (Z. 4 u. 5), dessen Pfade angegeben werden müssen. Die `topic-keys`-Datei ist eine CSV, die in Spalte 1 die Topic-Nummer enthält, in Spalte 2 den Hyperparameter Alpha[37] und in der dritten die Top-Topic-Wörter. Letztere sind die Wörter, die maßgeblich in einem Topic vertreten sind. Der simple Aufbau dieser Datei gestaltet das Auslesen der Top-Wörter einfach und wird auch primär aus diesem Grund erstellt. Um die Kohärenzen für

[35] Das letzte Argument `shell` ist zumindest auf dem eingesetzten Windows-System nötig, damit das Programm auch ausgeführt wird, da sonst ein Fehler geworfen wird. Recherchen auf der Webseite Stack-Overflow deuten an, dass dies eventuell nicht auf Unix-Systemen vorhanden sein muss.

[36] Informationen über alle möglichen Parameter können mit `bin mallet train-topics -help` aufgelistet werden.

[37] Die Hyperparameter werden mit `-optimize-inverval` bestimmt und sind Gewichtungen für Topics. Mit diesem Argument wird bestimmt, nach wievielen Iterationen MALLET sie automatisch nachjustiert. MALLET empfiehlt hierzu den Standardwert 10 beizubehalten. In dieser Arbeit werden die Standartwerte genutzt. Für mehr Informationen über Hyperparametrisierung siehe den Blog-Post von Schöch, 2016.

ein MALLET-Modell zu ermitteln, werden diese Wörter an den Palmetto-Endpoint gesendet, da Gensims Coherence Modelle nicht kompatibel sind (s. folgendes Kapitel).

Mit `--diagnostics-file` wird bestimmt, wohin die Diagnostik-XML gespeichert werden soll und wie im vorigen Kapitel beschrieben, kann mit `--num-threads` bestimmt werden, wie viele Prozesse parallel berechnet werden können, um Zeit zu sparen.

4.6.3 Palmetto

Für die MALLET-Kohärenzen wird das Tool Palmetto verwendet, das, wie in Kapitel 2.4.2 bereits erläutert wurde, eine extrinsische Berechnungsmethode darstellt.

Das Tool wird mittlerweile in einer aktualisierten Version auf PyPi zur Verfügung gestellt und kann somit bequem mit `pip install palmettopy` über die Kommandozeile installiert werden. Alternativ kann der Code auch über das Repository (URL in Kap. 2.4.2) heruntergeladen und so verwendet werden[38].

```
1 def extract_top_words(file):
2     for l in file.open().readlines():
3         columns = l.split("\t")
4         yield {columns[0]: columns[2].split(" ")}
```

Code 26: Extraktion von Topics und ihren Top-Wörtern aus einer MALLET-Keys-CSV. Ausschnitt aus `palmetto_coherence.py`.

In Code 26 wird als erstes eine MALLET-Keys-Datei eingelesen, in der jede Zeile ein Topic darstellt (Z. 1). Durch Splitting am Tabulator-Zeichen (Z. 3), wird die Zeile in seine Bestandteile zerlegt und in einer Liste gespeichert. Der Hyperparameter Alpha wird ignoriert und es wird ein Python-Dictionary zurückgegeben, das als Schlüssel die Topic-Nummer und als Wert eine Liste mit dessen Wörtern enthält. Das Dictionary wird dann in der Funktion `get_coherences()` verwendet (siehe Code 27).

[38] Es sei hier angemerkt, dass die Antwort des Endpoints pro errechneter Kohärenz eine Weile dauert, was zu einem Fehler durch Timeout führen kann, da dieser im Quellcode zu niedrig gesetzt wurde. Im Palmetto-Objekt kann in der `__init__`-Methode die Sekundenzahl der Variable `timeout` erhöht werden. Es wird empfohlen über 60 Sekunden einzutragen.

```
1  def get_coherences(top_words):
2      palmetto = Palmetto()
3      print("connecting to palmetto service...")
4      for dict in top_words:
5          for topic, words in dict.items():
6              # Try again when connection fails
7              for i in range(0, 10):
8                  try:
9                      print("calculating ", topic, "...")
10                     score = palmetto.get_coherence(words[:10], endpoint
                           )
11                     print(score, words[:10])
12                     break
13                 except:
14                     print("next try: ", str(i+1))
15                     continue
16             yield {topic: [score, words[:10]]}
```

Code 27: Berechnung der Kohärenz mit Palmetto. Ausschnitt aus `palmetto_coherence.py`.

Die eigentliche Implementierung ist simpel und erfordert lediglich die Erstellung eines Palmetto-Objektes (Z. 2), gefolgt vom Aufruf von Palmettos `get_coherence()`. Die Funktion benötigt eine Liste aus maximal 10 Wörtern, welche die ersten 10 aus einem Topic-Dictionary sind, und das Kohärenzmaß, welches in der Variable `endpoint` von Nutzenden bestimmt werden muss (Z. 6). Gültige Werte[39] sind `ca`, `cp`, `cv`, `npmi`, `uci` und `umass`. Die Berechnung dauert ein paar Sekunden und benötigt eine stabile Internetverbindung[40]. Manchmal kommt es unerwartet zu Fehlern, weil der Palmetto-Endpunkt nicht erreichbar war und das Programm deshalb abbricht. Daher wurde ein try-except-Block innerhalb einer Schleife eingebaut (Z. 7-15), der in einem solchen Fall bis zu 9 weitere Versuche startet, in der Hoffnung, dass das Problem nicht erneut auftritt.

Das Ergebnis wird wieder als Dictionary mit `yield` zurückgegeben und dessen Inhalt als dreispaltige CSV-Datei tabulatorsepariert abgespeichert. Die erste Spalte ist die

[39] Mehr zu Palmettos unterstützten Kohärenzmaßen lassen sich hier finden: https://github.com/dice-group/Palmetto/wiki/Coherences

[40] Letztendlich erzeugt der Code eine URI, die wie folgt aussehen kann: http://palmetto.aksw.org/palmetto-webapp/service/cv?words=cake+apple+banana+cherry+chocolate. Damit wird ein Request an den Server geschickt und dessen Rückgabewert zurückgegeben. Die URI setzt sich zusammen aus der Palmetto-Adresse + Kohärenzmaß + Top-Wörtern, für die das Maß berechnet werden soll.

von MALLET vergebene Topic-Nummer, die zweite der Kohärenzwert und die dritte beinhaltet die jeweiligen Top-Wörter, die mit einfachen Leerzeichen voneinander getrennt sind. Tabelle 4 zeigt die ersten 5 Topics eines Beispielmodells, Kohärenzen sind auf die zweite Dezimalstelle gerundet.

0	0.38	tree hill wood road river walk mountain lie valley rock
1	0.35	life people understand truth true feel wrong live religion matter
2	0.35	morning hour walk night time leave evening return house meet
3	0.49	ship sea water boat sail deck river captain vessel wind
4	0.34	letter write paper read hand send note leave word address

Tabelle 4: Beispiel für eine Palmetto-Output-Datei

Wegen der fragilen Natur dieses Codes ist dieses Vorgehen sehr zu empfehlen, damit die aktuell berechnete Kohärenz als Stream[41] in die Datei geschrieben wird, denn sonst droht ein Verlust der Daten.

Die beste Alternative ist jedoch, das Programm auf der lokalen Maschine zu benutzen, um die Abhängigkeit zur Internetverbindung und zum Server gänzlich zu umgehen. Denn es stellte sich wiederholt heraus, dass es selbst mit dem `try-except`-Block noch dazu kommen kann, dass das Skript nicht zuverlässig funktioniert und sollte der Server überhaupt nicht mehr erreichbar sein, dann ist diese Methode gänzlich unbrauchbar. Es gibt es zwei lokale Lösungen, die getestet wurden: das Java-Programm in der Kommandozeile zu starten und Palmetto als Docker-Container lokal zu hosten.

Ersteres birgt zum einen das Problem, dass es nur in der veralteten Version 0.1.0 (zu diesem Zeitpunkt ist die aktuelle Version 1.2.0) als ausführbare JAR[42] zur Verfügung steht. Zum anderen müsste für die Python-Integration der Terminal-Output ausgelesen und formatiert werden, um die relevanten Informationen (mit regulären Ausdrücken) herauszufiltern, was umständlich und fehleranfällig ist.

Wird das Programm lokal in Docker[43] gehostet, kann das vorgestellte Skript weiter

[41] Es wird mit `yield` kontinuierlich je eine neue Zeile für `write_to_file()` generiert, wo das Ergebnis sofort abgespeichert wird.

[42] Java Archive File. Eine komprimierte Datei, die Dateien eines Java-Programms enthält.

[43] Die Einrichtung verlief nicht problemlos. Es wird empfohlen, bei Komplikationen einen Blick in das Git-Issue #49 zu werfen: https://github.com/dice-group/Palmetto/issues/49. Der Code-Maintainer hat die Readme-Datei hinsichtlich des Mounting-Vorgangs aktualisiert und sollte bei Befolgung dazu führen, dass Docker die Wikipedia-Indizes im richtigen Pfad kopiert hat.

verwendet werden, indem im Palmetto-Code die Variable `palmetto_uri` angepasst wird. Statt der Standard-URI wird die Adresse des Docker-Containers angegeben (z.B. http://localhost:7770/service/, wenn er sich auf Port 7770 befindet). Diese Variante wurde letztendlich für die weitere Analyse verwendet, da sie stabil läuft.

4.6.4 Diagnostik-XML auslesen

Für die Evaluation der XML wird die Datei in Python eingelesen. Seit Version 1.3.0[44] ist in `pandas` die Funktion `read_xml()` integriert, mit der der Inhalt einer flach strukturierten XML unkompliziert in einen Dataframe geladen werden kann, dessen Statistik-Funktionen[45] ideal für die Evaluation sind. Dafür muss lediglich der Pfad zur Datei übergeben werden und mit dem Argument `xpath` kann ein XPath definiert werden, das bestimmt, welche Elemente in der Tabelle aufgelistet werden sollen. Die Spalten werden nach Attributen und Elementen benannt. So können mit dem XPath `//topic` eine Tabelle mit allen Informationen über die Topic-Elemente und mit `//word` alle Word-Elemente gespeichert werden.

```
1 def read_diagnostics_xml(format):
2     dfs = []
3     for file in malletdir.rglob(f"{format}/iteration*/diagnostics.xml")
          :
4         dfs.append(pd.read_xml(file))
5     return pd.concat(dfs)
```

Code 28: Einlesen der Diagnostik-XML in einen Dataframe. Auszug aus `boxplotting_xml.py`

Code 28 ist ein Code-Template, das alle XMLs nach Format rekursiv aus dem MALLET-Verzeichnis einliest und sie als Ganzes, d.h. ohne Elementfilterung, und in einem Dataframe speichert. Zunächst wird aus jeder Datei ein Dataframe erstellt und zurückgeliefert wird ein aus den einzelnen Dataframes zusammengefügtes Dataframe. Anschließend kann der Inhalt gruppiert, statistisch ausgewertet und visualisiert werden.

[44] Die zum Verfassungszeitpunkt erst einen Monat alt ist. Siehe https://pandas.pydata.org/pandas-docs/stable/whatsnew/v1.3.0.html.

[45] Es gibt hierfür eine ausführliche Dokumentation: https://pandas.pydata.org/docs/reference.

5 Evaluation

In diesem Kapitel wird die Performanz der Topic Modelle von Gensim und MALLET miteinander verglichen, indem die Durchschnittskohärenzen ihrer Topics gegenübergestellt werden. Anschließend wird ermittelt, ob es sich bei den generierten Daten um eine Normalverteilung handelt, um entsprechend den passenden Signifikanztest auszuwählen. Weil die erstellten Modelle immer einen Zufallsfaktor enthalten, ist noch zu prüfen, wie groß die Spanne normalerweise zwischen den Kohärenzwerten ist. Diese fließt ebenso in den Vergleich mit ein.

5.1 Generelle Analyse der vorverarbeiteten Texte

Bei der ersten Sichtung der Texte fiel auf, dass noch viele Token existieren, die nicht aus Buchstaben bestehen. Zur Bereinigung dieser Zeichen wurde in `preprocess.py` noch die Funktion `clean_token()` eingefügt, die mit der Regular Expression `(?!-)[^a-zA-Z]` alle Sonderzeichen auf Tokenebene bis auf einfache Bindestriche entfernt, denn diese sollen für Wortkompositionen wie *police-officer* oder *well-chosen* bestehen bleiben. Das zeigt, dass die Bereinigung nur eine Annäherung sein kann, da sie vermutlich nie fehlerlos verlaufen wird und Kompromisse eingegangen werden müssen, sobald von der Standardsprache abgewichen wird. Das Token *gov'ner* beispielsweise, eine Verkürzung des Wortes *governor* in wörtlicher Rede, wurde nicht richtig vom Tokenizer erkannt und die Entfernung oder Trennung des Apostrophs führt zu keinem sinnvollen Wort. So muss dem Menschen die Interpretation überlassen werden und dies wird in Kauf genommen. Doch selbst die Beibehaltung der Bindestriche ist problematisch, da sie in manchen Quelltexten auch als Zeilentrennzeichen verwendet werden und somit eigentlich zusammengeschriebene Wörter mit Bindestrich im Korpus auftreten wie beispielsweise *to-morrow*. Allerdings scheint es sich bei stichprobenartiger Untersuchung, um kein sehr häufiges Phänomen zu handeln und wird daher als unbedenklich eingestuft. Nach der Bereinigung erscheinen die Texte deutlich besser, auch wenn vereinzelt Störungen zu finden sind.

5.2 Evaluierung der Tools MALLET und Gensim

Wie bereits erläutert, sind die Trainingszeiten mit Gensims `LdaMulticore`-Modellen geringer, was für die Erstellung jener Modelle spricht.[46] Allerdings stellt dies keine Aussage über die tatsächliche Qualität der Modelle dar. Auch wenn die Kohärenzen der Formatsmodelle relativ zueinander verglichen werden und daher die eigentliche Qualität, solang sie auf gleiche Weise erzeugt wurde, keine hohe Priorität besitzt, ist es vorteilhaft die Modelle zu vergleichen. So kann überprüft werden, ob überhaupt sinnvolle Ergebnisse zu erwarten sind. Wenn ergänzend die intuitive Evaluation durch den Menschen ebenso einfließen soll, dann erschweren inkohärente Topics dies ungemein.

Daher werden auf dem vermeintlich besten Format, dem Original, mehrere Modelle mit je beiden LDA-Algorithmen trainiert. Für einen ersten Überblick werden Modelle mit stetig ansteigender Anzahl an Topics trainiert und die Entwicklung in einem Graphen visualisiert. Der Bereich erstreckt sich von 10 bis 290 Topics, wobei zwischen 10 und 150 die Schrittgröße 5 beträgt, die sich ab 150 auf 10 erhöht. In einem solch unüblich hohen Topic-Bereich ist nicht davon auszugehen, dass noch sinnvollere und kohärente Topics generiert werden. Die Erhöhung der Schrittzahl sorgt pragmatisch für weniger Rechenzeit und falls doch ein Trend ablesbar wäre, so sollte dieser auch damit erkennbar sein.

Für die Erstellung des Graphen[47] wird das Modul `pyplot` aus der externen Bibliothek `matplotlib` (Hunter, 2007) verwendet. Der Basistext wurde auf Substantive, Verben und Adjektive gefiltert, die Segmentgrößte beträgt 500 und das Kohärenzmaß ist Gensims Implementation des vektorbasierten `c_v`-Maßes, welches mit der Methode `get_coherence()` für das gesamte Modell ermittelt wurde. Bei dem gewählten Maß bewegt sich die Skala zwischen 0 und 1.

In Abbildung 11 ist klar zu sehen, dass auf diesem Korpus MALLET (obere blaue Linie) die deutlich kohärenteren Modelle erzeugt als Gensim (untere rote Linie)

[46] Dieser Unterschied hat sich durch die Entdeckung von MALLETs Multithreading-Unterstützung des Autors im Laufe der Untersuchung relativiert.

[47] Für die Generierung eines Kohärenzverlauf-Graphen erweist sich Gensims Wrapper für MALLET-Modelle als äußerst hilfreich, weil die Kohärenzen zur Skriptlaufzeit erfasst werden. Wird MALLET dagegen im Terminal gestartet, kann dessen Output nur als Datei im System gespeichert werden. Diese müsste dann wieder ausgelesen und formatiert werden, sodass dessen Kohärenzen berechnet werden können.

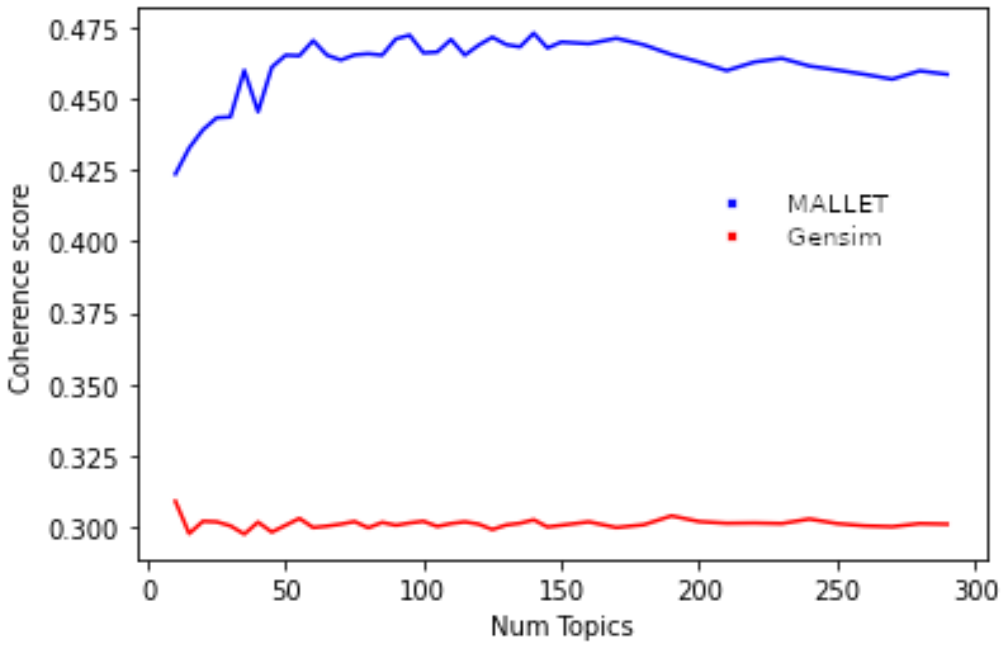


Abbildung 11: Kohärenzverlauf in Bezug auf Topicanzahl. Erzeugt von Gensim und MALLET.

und selbst dessen höchster erzielter Wert[48] mit 0.304 bei 190 Topics (was eine ungewöhnlich große Zahl an Topics ist) um etwa 30% geringer ist als der niedrigste MALLET-Wert mit 0.424 bei 10 Topics. Beim vergleichen des Durchschnittswertes aller erreichten Kohärenzen, der bei MALLET 0.461 und bei Gensim 0.301 beträgt, ergibt sich sogar ein Unterschied von 34.7%. Anhand des niedrigsten (0.4235) und des höchsten Wertes (0.4728), eine Spanne von 0.0493, wirkt sich die inkrementelle Steigerung der Topiczahl auf das MALLET-Modell deutlich stärker aus. Gensims Kurve ist mit einer Gesamtspanne von 0.06 wesentlich flacher, wodurch die Differenzierung und der Wertevergleich erschwert wird. Das deutet darauf hin, dass mit MALLET die Unterschiede in der Topicqualität simpler und deutlicher herauszufinden sind und die erzielte Topic-Kohärenz deutlich höher ist.

Es ist auch zu sehen, dass MALLETs Werte ab der 150-Topic-Marke tendenziell zu sinken beginnen. Daher ist es nicht sinnvoll über diese Grenze hinaus Modelle zu trainieren. Doch bereits ab Topiczahl 60 ist eine Kohärenz von 0.47 erreicht, bevor sie wieder zu sinken beginnt und schließlich bei 140 das Maximum erreicht, welches nur um 0.0026 größer ist. Auch das Gensim-Modell hat erhöhte Kohärenzwerte um

[48] Der höchste hier gemessene Kohärenzwert von 0.309 bei Topicanzahl 10 scheint bei dieser Korpusgröße nicht angemessen zu sein und wird daher ignoriert, zumal der Wert bei 10 Topics in MALLETs Modell der erwartungsgemäß niedrigste ist.

diese Topic-Zahl, auch wenn der Ausschlag zu den vorherigen und nachfolgenden nicht groß sind.

Letztendlich wird für die Erstellung weiterer Modelle auf Gensim in dieser Arbeit verzichtet, denn MALLET liefert die besseren Kohärenzen und stellt die Diagnostik-XML für die Untersuchung weiterer Werte zur Verfügung.

5.3 Hypothesenprüfung

Um festzustellen, ob sich die in Kapitel 3 aufgestellten Hypothesen zur Eignung der Formate in Bezug zum Original erhärten oder widerlegen lassen, wird wie folgt vorgegangen.

Zunächst müssen Datensätze erstellt werden. Sie bestehen zum einen aus Topic-Modellen aus dem Originalformat, ihren Kohärenzen und ausgewählten Werten aus der Diagnostik-XML. Zum anderen werden Modelle für jedes Format generiert, deren Kohärenzen und Diagnostik-XMLs ebenso erstellt werden. Die Parametrisierung ist für jeden Vergleich dieselbe. Das betrifft die Segmentgrößen und Wortartenfilter. Für jedes Format, einschließlich dem Original, werden mehrere Modelle aus dem gleichen Quelltext generiert, damit eine höhere Sicherheit bei der Evaluation gegeben ist. Denn es ist dem Zufallscharakter des LDA-Algorithmus geschuldet, dass trotz derselben Textbasis unterschiedliche Modelle erzeugt werden, was mit sich bringt, dass Kohärenzschwankungen erwartet werden. Getestet wird ein Datensatzpaar und ihre Hypothesen mit den statistischen Verfahren *Signifikanztest* und mit dem *Konfidenzintervall*. Zuletzt kann es von Interesse sein, den Kohärenzverlauf der Formate zu betrachten, um festzustellen, wie sich ihre Kohärenzen entwickeln.

5.3.1 Der Signifikanztest

> Zum statistischen Nachweis von Unterschieden oder Effekten werden häufig Signifikanztests eingesetzt. Das Ergebnis eines solchen Tests wird zumeist als p-Wert ausgegeben. Anhand dieses p-Werts wird entschieden, ob beobachtete Unterschiede statistisch signifikant sind (wenn der p-Wert kleiner ist als das Signifikanzniveau α von zum Beispiel 5%) oder nicht. (Lange und Bender, 2001, S. T42)

Also bietet sich dieser Test für den Vergleich zweier Kohärenzdatensätze an. Doch zunächst ist es formal notwendig eine Nullhypothese zu formulieren. Sie stellt die Behauptung dar, die mit dem p-Wert widerlegt werden soll (z.B.: Es gibt keinen Unterschied zwischen den Kohärenzwerten der einfachen Term-Dokument-Matrix und denen des Originals). Als zweites wird eine Alternativhypothese formuliert, die besagt, dass es einen Unterschied zwischen den beiden Formaten gibt. Sie bildet das Gegenstück zur Nullhypothese (z.B.: die Kohärenzwerte des Originals sind besser als die der Term-Dokument-Matrix) (vgl. Prel et al., 2009, S. 335). Liegt der errechnete p-Wert unter dem o.g. Signifikanzniveau (das standardmäßig bei 1% liegt, wenn es eine strengere Bewertung ist; sonst bei 5%), soll die Nullhypothese mit einem Blick falsifiziert oder bestätigt werden können.

Für den Datensatz wurden 52 MALLET-Modelle auf einer Originaldatei trainiert, ihre Keys-Dateien ausgelesen und die Top-10 Wörter jedes Topics mit Palmetto berechnet. Ausgehend von der Abbildung 11 wurde Segmentgröße 500 und Topic-Zahl 60 gewählt. Die Anzahl der Modelle liegt pragmatisch darin begründet, dass eine nicht zu kleine Anzahl gewählt werden soll, damit eine aussagekräftigerer Datensatz entsteht, aber nicht so groß ist, dass die Kohärenzberechnung zu viel Zeit[49] in Anspruch nimmt. In `confidence_interval.py` werden die Ergebnisse eingelesen und als Dataframe abgespeichert, auf dessen Grundlage diverse statistische Berechnungen durchgeführt werden können.

Die Wahl des Signifikanztests ist abhängig vom Wahrscheinlichkeitsverteilungstyp des Datensatzes, daher muss dieser zuerst ermittelt werden. Der Datensatz soll stellvertretend für alle Modelle aufzeigen, um welche Wahrscheinlichkeitsverteilung es sich generell bei MALLET-Topics handelt. Dafür wird der Graph der Dichtefunktion generiert und geprüft, ob er annähernd die typisch symmetrische Form einer Glockenkurve darstellt. Technisch umgesetzt wird das mit der Bibliothek `seaborn`(vgl. Waskom, 2021), die mächtige Funktionen für Datenvisualisierungen durch `matplotlib`-Integration bietet. Dafür wird der Durchschnitt jeder Datenzeile im Dataframe mit `Dataframe.mean()` erstellt. Das Ergebnis ist ein Series-Objekt (`pandas` Version einer Liste) generiert, in der 52 Durchschnittswerte enthalten sind. Mit Code 29 wird

[49] Allein die Berechnung der Kohärenzen mit Palmetto nimmt relativ viel Zeit in Anspruch. Um diese Daten zu generieren wurden etwa 17 Stunden Rechenzeit benötigt.

ein Histogramm erstellt, dass die Kohärenzvorkommen darstellt. Anhand eines Histogramms kann visuell abgeschätzt werden, wie die Daten verteilt sind.

```
1 import seaborn as sns
2 ax = sns.displot(data=means, kde=True, bins=12)
3 ax.set(xlabel="Coherence")
4 plt.show()
```

Code 29: Erstellung eines Histogramms mit seaborn

In Z. 2 wird das Histogramm auf Basis des Dataframes `means` (die Durchschnittswerte der Topics) erstellt und das Argument `kde` zeichnet einen Kernel-Density-Estimation-Graphen[50] (KDE) über das Histogramm. Sie ist eine geglättete Kurve, die am besten zu den gegebenen Daten passt. Das Argument `bins` legt fest, in wie viele Datenklassen (Balken) das Histogramm zerlegt werden soll. An den Wert 12 wurde sich experimentell angenähert und liefert das übersichtlichste Ergebnis. Mit der letzten Zeile wird der Plot mit `matplotlib` angezeigt. Der Graph in Abbildung 12 ist das Ergebnis dieses Codes.

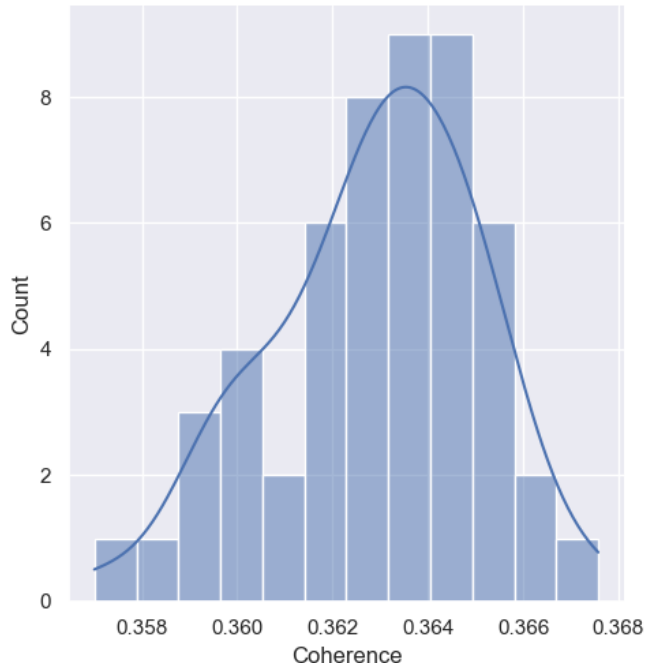


Abbildung 12: Histogramm mit KDE-Graph für Durchschnittswerte von Topics, generiert mit `seaborn` und `matplotlib`

Der Graph stellt keine perfekte Symmetrie dar (die linke Seite ist etwas flacher als

[50] Empfehlenswertes Videotutorial zu KDE-Plots mit `seaborn` https://www.youtube.com/watch?v=DCgPRaIDYXA.

die rechte), doch ist zu erkennen, dass es sich um eine Annäherung an die Normalverteilung handeln könnte, denn die Datenpunkte sind an einem Mittelpunkt am häufigsten bzw. dichtesten (bei ca. 0.363) und die restlichen Punkte verteilen sich in der Nähe. Um die Vermutung zur Gewissheit zu machen, wird das Objekt `Fitter` (Cokelaer, n. d.) genutzt. Es bietet die Möglichkeit, über alle Verteilungstypen, die in der Statistik-Bibliothek `scipy` (Virtanen et al., 2020) hinterlegt sind, zu iterieren und liefert als Ergebnis die bestmöglichen Treffer für den gegebenen Datensatz.

```
1 f = Fitter(means)
2 f.fit()
3 f.summary()
```

Code 30: Daten-Fitting

Code 30 zeigt, wie simpel das Daten-Fitting auszuführen ist. Es wird ein Fitter-Objekt `f` erstellt, das Fitting ausgeführt und anschließend das Ergebnis ausgegeben. Der Output ist eine Tabelle, die zeigt, welche Verteilung am besten zum Datensatz passt. In diesem Fall wird die `gumbel_l`-Verteilung als am passendsten empfohlen, die einer Normalverteilung ähnlich ist. Es wird angenommen, dass ein größerer Datensatz, der noch mehr Modelle enthält, sich der Normalverteilung annähern würde.

Mit diesem Wissen kann ein passender Signifikanztest ausgewählt werden. Ein häufig verwendeter Test ist der sogenannte t-Test:

> Mit dem t-Test kann die Signifikanz beim Vergleich stetiger Zielgrößen geprüft werden, indem die Gleichheit bzw. Verschiedenheit zweier Stichproben anhand der Differenz ihrer Erwartungswerte gemessen wird. Erwartungswerte entsprechen Mittelwerten von (fiktiven) unendlichen Grundgesamtheiten. Die Mittelwerte aus Stichproben sind Schätzwerte für die entsprechenden Erwartungswerte. Vor Durchführung eines Signifikanztests muss festgelegt werden, bei welchem Irrtumsniveau die Nullhypothese abgelehnt werden soll. (Lange und Bender, 2001, S. T43)

Mit `scipy` kann eine große Auswahl an Tests durchgeführt werden, u.a. der t-Test zum Vergleichen zweier unabhängiger Datensätze (The SciPy community, 2021). Die Funktion befindet sich im Modul `stats` und heißt `ttest_ind`. Dieser Funktion werden die beiden Datensätze in Form von Listen übergeben. Diese sollen die Kohärenzmittelwerte für jedes Modell beinhalten. Der Testaufruf ist unkompliziert und lässt sich mit nur einer Zeile (exklusive Import) realisieren und wird in Code

31 dargestellt.

```
from scipy import stats
stats.ttest_ind(means1, means2)
```

Code 31: Aufruf der ttest-Funktion

5.3.2 Das Konfidenzintervall

> The confidence interval is the range of values that you expect your estimate to fall between a certain percentage of the time if you run your experiment again or re-sample the population in the same way. (Bevans, 2020)

Das Intervall beschreibt in diesem Fall den Wertebereich des Mittelwertes mit 95%iger Wahrscheinlichkeit. Es wird hier ergänzend zum t-Test berechnet, denn dieser kann ergänzend zum p-Wert zur Bewertung herangezogen werden. Anders als der p-Wert, kann das Intervall visualisiert werden, indem der bezeichnete Wertebereich als Ober- und Untergrenze eingezeichnet wird.

Eine elegante Lösung zur Berechnung[51] bietet die Statistikbibliothek `statsmodels` (Seabold & Perktold, 2010), wie in Code 32 zu sehen.

```
import statsmodels.stats.api as sms
sms.DescrStatsW(data).tconfint_mean()
```

Code 32: Konfidenzintervallberechnung

Diese Funktion liefert die Ober- und Untergrenze des Intervalls zurück und benötigt zur Berechnung lediglich den Datensatz als listenähnliches Objekt.

Hierfür sollten Datensätze erstellt werden, die eine ähnlich große Anzahl an Datenpunkten beinhaltet. Aus pragmatisch-zeitlichen Gründen wird die Topic-Zahl pro Modell verringert.[52] Für jedes Format werden pro Segmentlänge 20 Modelle mit

[51] Eine gute Übersicht zur Berechnung bietet Bevans, 2020.

[52] Die Generierung der Daten kann viel Zeit beanspruchen und dazu führen, dass der Rechner hinsichtlich seiner Rechenleistung und Arbeitsspeicherkapazität an seine Grenzen kommt. Dies kann im parallelen Gebrauch zu Abstürzen und allgemeinen Performanzeinbußen führen. Bei einem System mit 16 GB RAM und aktuellem i7-Prozessor hat die Datengenerierung wegen diesen Umständen deutlich über 24 Stunden angedauert. Es wurden mehrere Python-Konsolen in der Entwicklungsumgebung *PyCharm* gestartet, die jeweils ein Viertel der Daten mit MALLET trainierten. Es konnte beobachtet werden, dass je Modell etwa 2-4 Minuten Rechenzeit

20 Topics generiert, damit lassen sich Datensätze von je 60 Kohärenzmittelwerten miteinander vergleichen, indem jedes Format paarweise mit dem Original verglichen wird. Die Segmentgrößen sind 500, 1000, 1500 und 2000 und bewegen sich damit im für TM-Analysen üblichen Bereich. Damit wird ein breiteres Spektrum abgedeckt. Der Wortartenfilter umfasst Substantive, Verben und Adjektive. Für jedes Format wird stichprobenartig ihre Textkomposition betrachtet, ihre KDE-Plots, die Top-Wörter ihrer Topics miteinander verglichen und die Signifikanz mit dem jeweiligen p-Werte des t-Tests geprüft.

5.4 Datenauswertung

5.4.1 Original

Es ist sinnvoll eine allgemeine Analyse der Referenzdaten durchzuführen bevor die Formate mit ihnen verglichen werden.

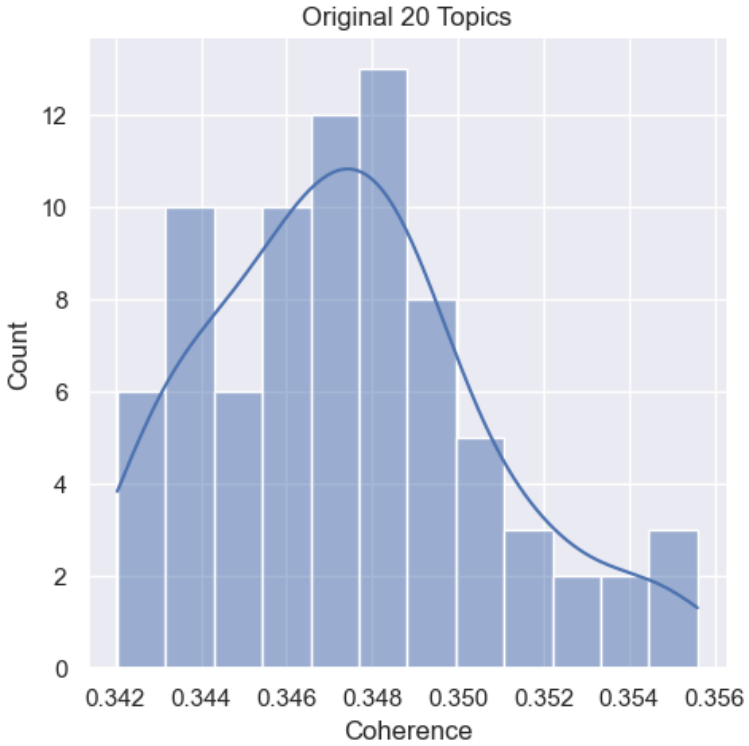


Abbildung 13: Histogramm mit KDE-Plot für 20 Originalformate mit je 20 Topics

benötigt wurde. Die Berechnungszeit mit Palmetto im größeren Stil ist nicht sehr praktikabel, wenn viele Topics berechnet werden sollen, denn dieser Vorgang war in der lokalen virtuellen Maschine von Docker sehr resourcenhungrig und langsam. Der offizielle Server war zu vielen Zeitpunkten nicht erreichbar, daher sei hier ausdrücklich die lokale Nutzung empfohlen. Vor allem die Berechnung mit Palmetto sorgte für die langen Rechenzeiten.

Kohärenz	Topic-Wörter
0.386	water ship boat hand sea foot head freckle lie time
0.373	door room open window house light stand sit wall bed
0.364	child mother dear father poor time hear talk leave aunt

Tabelle 5: Top 3 Topics des Originals

Abbildung 13 zeigt das Histogramm mit KDE-Plot für die Kohärenzen des Originalformates. Im Vergleich zum Graphen in Abbildung 12 zeigt sich, dass die Kohärenzspanne bei 20 Topics insgesamt geringer ist als bei 60 Topics, was auch im Kohärenzverlauf aus Abbildung 11 abgebildet wurde. Das bedeutet, dass es sich hier vermutlich nicht um die ideale Topic-Zahl handelt. Aber solange sie über alle Formate hinweg konsistent bleibt, bleibt auch der Vergleich valide. Die Kohärenzen erstrecken sich insgesamt von 0.342 bis 0.356 und damit in einem Bereich von 0.14. Das Konfidenzintervall mit Konfidenzniveau 95% liegt im schmalen Wertebereich von 0.3466 und 0.3481.

Für die Untersuchung auf Topic-Ebene wird ein Modell, dessen Durchschnittswert noch im Konfidenzintervall liegt und dessen Top-Wörter überprüft. Das Modell[53] aus der Iteration 6 erzielt den Wert 0.3480, wurde aus der Segmentlänge 500 erzeugt und liegt damit im oberen Bereich des Intervalls. Bei der Betrachtung der Top-Wörter (siehe Tabelle 5) fällt auf, dass viele Wörter mehreren Topics zugeordnet wurden, beispielsweise ist `time` in 12, `hand` in 6 und `feel` und `word` in jeweils 5 Topics vertreten. Bei einer Zahl von nur 20 Topics können diese Wörter in die Stoppwortliste aufgenommen und die Topic-Zahl durchaus vergrößert werden, weil sie durch die hohe Frequenz keinen semantischen Mehrwert bieten. In Tabelle 5 sind die Topics mit den höchsten drei Kohärenzwerten aufgeführt. Es fällt bei ihnen nicht schwer, Zusammenhänge festzustellen: das erste Topic beschreibt typische Dinge bei der *Seefahrt*, das zweite *Wohnung* und *Haus* und das dritte zu großen Teilen *Familienmitglieder*. Die Wortwahl erscheint rein aus menschlich-qualitativer Sicht vollkommen nachvollziehbar, auch wenn vereinzelt unpassende Wörter wie `time`, `freckle` und Verben[54], die sich auf *Bewegung* oder *Körperhaltung* beziehen, vorkommen.

In Tabelle 6 sind die Topics mit den schlechtesten Kohärenzen aufgeführt. Hier fällt

[53] Die Tabellen finden sich im Repository.
[54] Diese könnten auch als Stoppwörter deklariert werden.

Kohärenz	Topic-Wörter
0.331	head hand laugh eye turn fellow hear reply time play
0.328	hand speak word heart time turn hear fair stand folk
0.325	country public society year time people family party church power

Tabelle 6: Worst 3 Topics des Originals

die eindeutige Interpretation tatsächlich schwerer. Überraschenderweise aggregiert das letzte Topic Wörter, die menschliches Gemeinde- und Staatsleben beschreiben könnten, aber den anderen beiden einen Sinn zuzuschreiben ist nicht trivial oder schlicht nicht möglich. Das zeigt, dass die Kohärenz womöglich schon in solch einem kleinen Zahlenbereich eine gewisse Aussagekraft über die tatsächliche Qualität aussagen kann. Der Vergleich mit den Topic-Wörtern der folgenden Formate sollte in dieser Hinsicht aufschlussreich sein.

5.4.2 N-Gramme

Erwartungsgemäß sind N-Gramme, zumindest der Größenordnung dieses Korpus, für Topic Modeling nicht geeignet. Der Text, der übrig bleibt, sobald eine Mindesthäufigkeit von 2 vorausgesetzt wird, ist nur ein Bruchteil des Volltextes. Es stellt sich heraus, dass bei der Filterung nach Substantiven, Verben und Adjektiven nur noch 9211 Wörter übrig bleiben.[55] Die N-Gramm-Größe beträgt hierbei 3 und die Segmentlänge 1000. Damit enthält dieses Format lediglich 0.24% des Originalumfangs (3 869 686 Wörter). Mit diesen Daten kann kein fairer Vergleich durchgeführt werden und deshalb wird das N-Gramm-Format aus der folgenden Evaluation ausgeschlossen.

5.4.3 Vergleich zwischen Original und einfacher Term-Dokument-Matrix

Die Umfänge beider Texte sind identisch. Ihre Wortzahl beträgt jeweils 3 869 686 und das Verhältnis zwischen Segmentgröße und -anzahl ist aus der Tabelle 7 zu entnehmen. Diese Werte sind ebenfalls bei beiden Textformaten identisch.

Damit besteht Sicherheit, dass der Code zur Formatsgenerierung in dieser Hinsicht

[55] Der Wortformfilter könnte noch Adverbien einschließen, jedoch dürfte das keinen signifikanten Unterschied für die übrigbleibende Textlänge machen.

Segmentgröße	Segmentanzahl
500	36329
1000	18193
1500	12156
2000	9129

Tabelle 7: Segmentgröße und damit verbundene Segmentanzahl

funktioniert hat. Die Texte sind somit quantitativ vergleichbar und unterscheiden sich lediglich durch die Anordnung der Token segment- und gesamttextübergreifend.

Als nächstes werden die KDE-Plots in einem Schaubild zusammengetragen, um sich ein erstes Bild über die Kohärenzverteilung zu machen. Abbildung 14 zeigt, dass sich die erzielten Kohärenzwerte zwischen 0.33 und 0.36 einordnen lassen und damit beträgt die Spanne nur 0.03. Wenn also ein signifikanter Unterschied bestehen sollte, sei an dieser Stelle hervorgehoben, dass es sich um kleine Werte handelt. Das bestätigt den Verdacht, dass für die Kohärenzuntersuchung die bestmögliche Berechnungsstrategie verwendet werden sollte, um besser interpretierbare Ergebnisse zu erhalten.

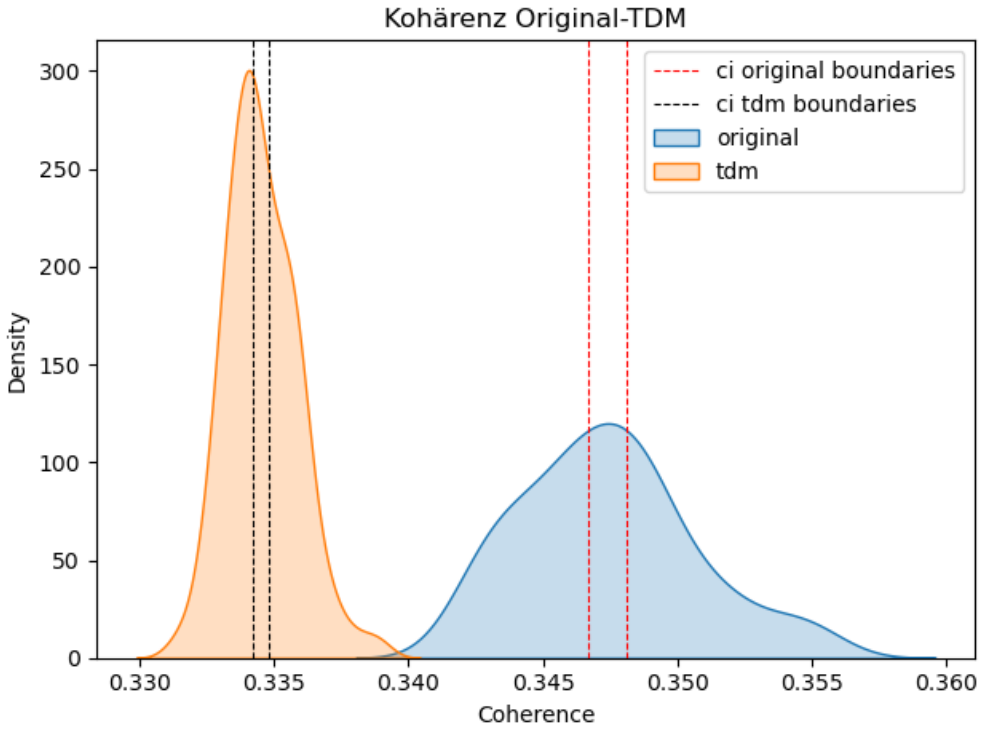


Abbildung 14: KDE-Plots aller TDM- und Original-Modelle mit Konfidenzintervallgrenzen

Es ist auch deutlich zu erkennen, dass beide Plots kaum Überschneidungen aufweisen, bis auf die maximalen Werte der TDM und minimalen Werte des Originals. Damit erhärtet sich die Vermutung, dass es signifikante Unterschiede zwischen den Mittelwerten gibt. Auffällig ist auch, dass die Kohärenzen der TDM deutlich weniger streuen und sich stattdessen deutlich stärker um das Konfidenzintervall verdichten. Dies könnte daher rühren, dass die Topics sich pro Iteration nicht stark voneinander unterscheiden, weil das Modeling durch die zufällige Wortkookurrenzen keine differenzierten Topics bilden kann.

Für den t-Test wird als Nullhypothese die Gleichheit beider Formate angenommen. Die Alternativhypothese lautet, dass das Original signifikant bessere Kohärenzwerte liefert als die TDM. Der Test liefert für die Mittelwerte einen sehr niedrigen Wert mit Endung `e-73` zurück. Diese Python-Notation mit der Endung `e-73` bedeutet, dass die davorstehende Zahl mit dem negativen 73. Exponenten von 10 multipliziert wird und damit 73 führende Nullen beinhaltet. Das Ergebnis ist (auf die zweite Nachkommastelle gerundet) $3.45 \cdot 10^{-73}$ selbst signifikant kleiner als der strengere Grenzwert $\alpha = 0.01$. Streng nach Beurteilung des p-Werts ist somit die Nullhypothese widerlegt und die Alternativhypothese bestätigt: das Original liefert signifikant bessere Kohärenzwerte als die TDM. Das Konfidenzintervall ist mit Untergrenze 0.3342 und 0.3348 noch deutlich schmaler als das des Originals und liegt mit allen Werten darunter.

Auch hier wird für die inhaltliche Analyse ein Modell aus dem Intervall gewählt: Iteration 4 mit Durchschnittswert 0.3346 und Segmentlänge 500. Allgemein fällt sofort auf, dass viele Begriffe in vielen Topics wiederholt werden. Hier ist das Wort `time` in 16, `hand` in 12 und `eye` in 10 Topics vertreten. Darüber hinaus gibt es noch mehr Fälle in ähnlicher Größenordnung. Die Wiederholungen sind häufiger als beim Original, was bereits auf weniger eindeutige Topics schließen lässt.

In Tabelle 8 sind die Top-3-Topics aufgeführt. Auch auf Topic-Ebene bestätigt sich, was auf der Modellebene zu sehen war: die Wertestreuung ist gering. Nur das erste sticht, relativ gesehen, als besonders hoch heraus und für die weitergehende Topic-Platzierung muss die dritte und höhere Dezimalstellen geprüft werden. Interessanterweise erhält das Topic mit ähnlicher Semantik (beinhaltet Familienmitglieder) den

Kohärenz	Topic-Wörter
0.364	time child poor hear mother feel father sister speak word dm
0.339	poor father heart hand mother word eye time hear turn
0.3388	feel time mind people dodo hand poor hear suppose dear

Tabelle 8: Top 3 Topics der Term-Dokument-Matrix

Kohärenz	Topic-Wörter
0.32734	eye life hand word face head soldier cigarette horse hear
0.32733	eye hand poor heart ship head turn face money word
0.3256	eye child life night grow light lie water hand time

Tabelle 9: Worst 3 Topics der Term-Dokument-Matrix

gleichen Wert wie das Familien-Topic aus Tabelle 5. Dass Palmetto hier ähnliche Kohärenzen errechnet ist aber nicht verwunderlich. Dass das Topic ähnliche Wörter beinhaltet, kann daran liegen, dass diese Wörter im Gesamtkorpus relativ häufig auftreten und damit die Segmente trotz Zufallszusammensetzung häufig miteinander kookurrieren oder es ist tatsächlich eine reine Zufallserscheinung.[56] Die beiden anderen Topics wirken sehr zufällig und es scheint unmöglich hier einen Zusammenhang zwischen den Wörtern zu finden.

Werden die schlechtesten drei Topics (Tabelle 9) betrachtet, bestätigt sich dieser Eindruck weiter. Auch hier ist allein an den Dezimalstellen erkennbar, dass die Unterschiede vernachlässigbar sind. Diese Topics ergeben wie erwartet keinen Sinn mehr und sind durch die häufigen Wiederholungen kaum noch zu unterscheiden.

Schlussendlich belegt auch der semantische Vergleich zwischen dem Original- und TDM-Modell, dass das Original zumindest bei den höher gewerteten Topics eindeutig bessere Ergebnisse liefert und dieses Format keine gute Alternative darstellt, wenngleich noch vage Interpretationen möglich sind.

[56] Hierfür müssten wieder mehrere Texte erstellt und verglichen werden, was aber nicht mehr im Rahmen dieser Arbeit liegt.

5.4.4 Vergleich zwischen Original und segmentweiser Aufhebung der Sequenzinformation

Wie zuvor bestätigt sich auch hier die Textvorverarbeitung durch identische Token- und Segmentlängen (vgl. Tabelle 7).

Im Vergleich zum TDM-Kapitel zeichnet sich in Abbildung 15 deutlich ab, dass die Überschneidungen in den Kohärenzwerten hoch sind. Das Konfidenzintervall ist nicht von beiden Formaten eingezeichnet, da sie so nahe beieinanderliegen, dass sie kaum von einander zu unterscheiden sind. Stattdessen ist nur eines für beide zu sehen. Es erstreckt sich bei der SAS, wie auch beim Original, von gerundet 0.347 bis 0.348 und unterscheiden sich erst ab der vierten Dezimalstelle. Das zeigt, dass die Kohärenzmittelwerte sich so ähnlich sind, dass sie kaum voneinander zu unterscheiden sind.

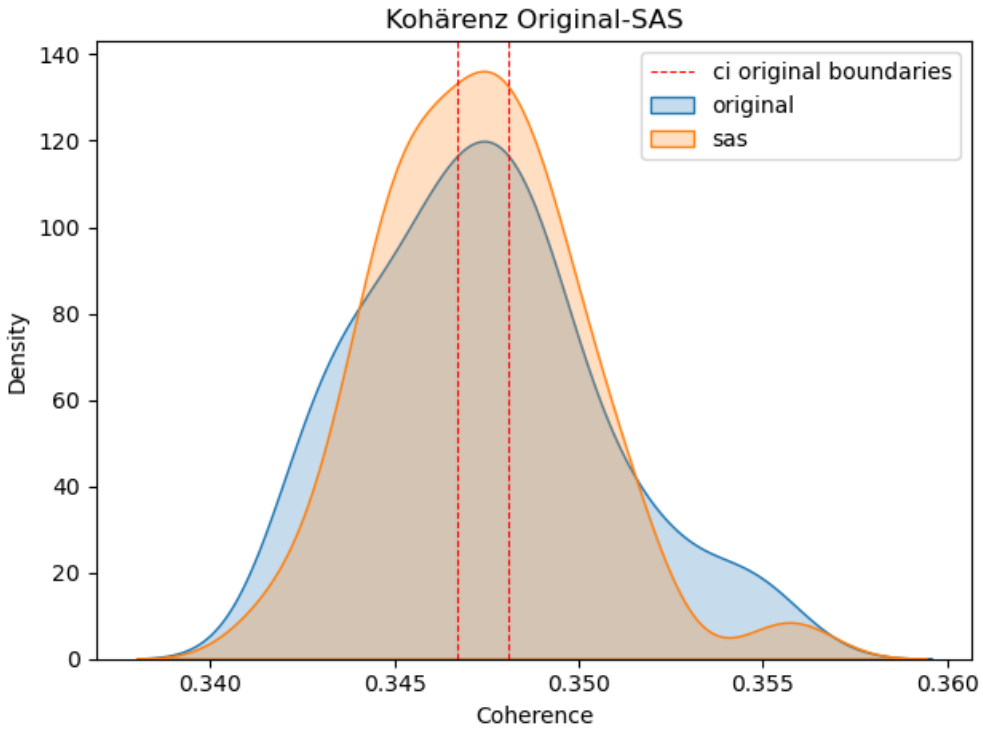


Abbildung 15: KDE-Plots aller SAS- und Original-Modelle mit Konfidenzintervallgrenzen

Der aus dem t-Test errechnete p-Wert ist 0.86 und liegt damit weit über beiden üblichen α-Werten. Damit ist auch nach t-Test die Nullhypothese bestätigt: es gibt keinen signifikanten Unterschied zwischen den Werten vom Original zum SAS.

Für die inhaltliche Betrachtung wird das Topic aus Iteration 19 mit Wert 0.3471 und Segmentlänge 500 herangezogen. Es wiederholen sich Wörter, aber nicht so häufig wie es bei den vorherigen Formaten der Fall war. Das Wort `time` taucht in 9, `feel` in 6, `hand` in 5 und `eye` in lediglich 4 Topics auf. Die Topics sind in dieser Hinsicht deutlicher voneinander abgegrenzt.

Kohärenz	Topic-Wörter
0.391	eye dress face hair white wear beautiful black smile head
0.379	room table sit door house glass drink dinner chair window
0.353	child mother father poor dear heart leave time hear speak

Tabelle 10: Top 3 Topics der segmentweisen Aufhebung von Segmentinformationen

Die Top-Tabelle 10 zeigt andere Wörter und Topics als im Original, was aber durch die Zufälligkeit nicht überraschend ist.

Im höchstbewerteten Topic ist es tatsächlich auch nicht trivial ein Wort zu finden, welches nicht passen würde, scheint jedes Wort das Erscheinungsbild einer bestimmten Person zu beschreiben. Auch beim zweiten Topic passen die Wörter semantisch durchaus zueinander und kann vage z.B. ein Abendessen beschreiben. Das dritte Topic aber kann diese Kohärenz allerdings nicht mehr demonstrieren, bietet zwar mit Familienmitgliedern teilweise ein kohärentes Topic, doch lässt sich ein menschlich interpretierbares Thema nicht trivial benennen.

Bei der Auswahl der schlechtesten Kohärenzen für Tabelle 11 ist bemerkenswert, dass die Werteverteilung relativ gleichmäßig ist und sich die meisten Topics (14 von 20) um den Bereich 0.33-0.34 ansiedeln. Die Interpretation jener Topics fällt, wie auch beim Original deutlich schwerer, aber einige Wörter unterstützen sich auch durchaus gegenseitig, wie im dritten Topic zu sehen `talk people laugh play evening`. Sie wirken nicht vollkommen willkürlich.

Kohärenz	Topic-Wörter
0.340	light night door hand room open face fall stand hear
0.339	life work mind feel human people nature book sort live
0.335	talk people time feel laugh suppose hear sit play evening

Tabelle 11: Worst 3 Topics der segmentweisen Aufhebung von Segmentinformationen

Schlussendlich wurde hiermit gezeigt, dass dieses Format eine gute, wenn nicht sogar gleichwertige Alternative zum Original darstellt.

5.4.5 Vergleich zwischen Original und selektiv reduzierten Informationen über Tokens

Auch hier ist der Textumfang der gleiche wie bei den anderen Formaten.

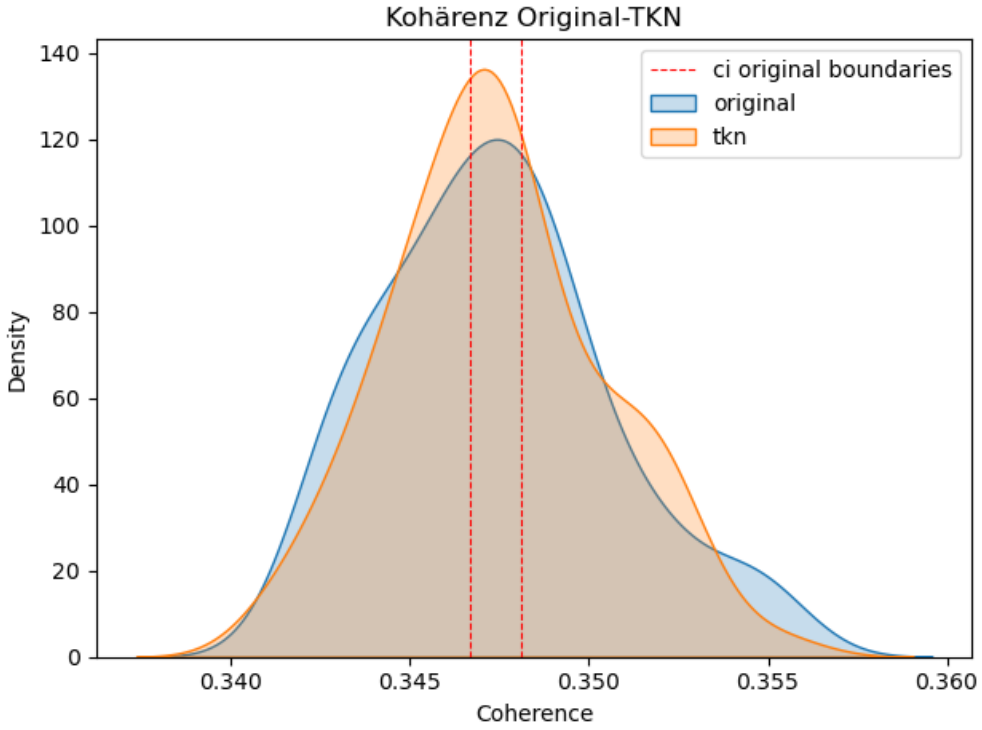


Abbildung 16: KDE-Plots aller TKN- und Original-Modelle mit Konfidenzintervallgrenzen

Abbildung 16 zeigt ein ähnliches Bild wie bereits im Vergleich zwischen Original und den SAS. Die KDE-Plots zeigen starke Überschneidungen und die Intervallgrenzen sind annähernd gleich (Unterschied erst ab der vierten Dezimalstelle), weshalb auch hier nur die des Originals eingezeichnet wurden. Die Durchführung des t-Tests liefert den p-Wert 0.99 zurück und damit sind die Datensätze nach diesem Test nicht signifikant unterschiedlich[57] und TKN ist nicht schlechter für Topic Modeling geeignet.

Für die Betrachtung der Topic-Wörter wird Iteration 8 mit Kohärenzwert 0.3471 und Segmentlänge 2000 gewählt.

[57] Wird der t-Test auf zwei identischen Datensätzen ausgeführt, dann wird der maximale p-Wert 1 erreicht.

Kohärenz	Topic-Wörter
0.43	ship water boat sea river sail hand time deck captain
0.38	child mother poor father dear time hear sister talk aunt
0.36	time eye feel big freckle sit face laugh people talk

Tabelle 12: Top 3 Topics der segmentweisen Aufhebung von Segmentinformationen

Kohärenz	Topic-Wörter
0.328	time hand return head friend dear sit table eye place
0.326	country time friend society party public year family life power
0.325	hand speak heart stand fair word friend fall turn knight

Tabelle 13: Worst 3 Topics der segmentweisen Aufhebung von Segmentinformationen

Auffällig ist wieder die Häufung potenzieller Stoppwörter aus den obigen Kapiteln. Aus den Stichproben erzielt dieses Format den höchsten Wert mit 0.43 im *Seefahrt-Topic*. Nur `hand` und `time` sind hier Abweichler. Das zweite Topic in Tabelle 12 beschreibt*Familienmitglieder* und ist damit auch den oben beschriebenen Formaten ähnlich. Das dritte Topic ist durchwachsen und nicht mehr so aufschlussreich wie die anderen.

Tabelle 13 präsentiert die niedrigsten Werte und sind in einem ähnlichen Wertebereich wie die niedrigsten Topics des Originals.

5.4.6 Vergleich aller Formate auf Topic-Kohärenz-Ebene

Durch die Erzeugung von Mittelwerten, werden pro Format 1600 Kohärenzwerte auf 80 reduziert und damit geht nuancierte Information verloren. Für die Auswertung aller Topic-Kohärenzen (6400 insgesamt) werden sie daher in Abbildung 17 als Box-Plot visualisiert, welche nach ihren Segmentlängen kategorisiert sind, um eine noch detaillierte Darstellung der Daten zu erhalten. Sie wurde auch mit `seaborn` generiert, indem die Kohärenzen zunächst als Dataframe gespeichert werden. Diese ist in drei Spalten unterteilt: Format, Segmentlänge und Kohärenz.[58]

[58] Es ist wichtig darauf zu achten, dass die Datenfelder richtig typisiert sind. Das kann in einer dynamisch typisierten Programmiersprache wie Python schnell zu Verwirrungen führen, wie eigene Erfahrungen zeigen. Es wurden mit einer Funktion Strings ausgelesen und deren Inhalte nicht richtig als Floats (Kohärenzwerte) und Integer (Segmentlänge) typisiert, sodass `seaborn` keine kategorisierten Plots erstellen konnte und Fehler warf.

Für die kategorisierte Darstellung von Box-Plots wird die Funktion aus Code 33 aufgerufen.

```
import seaborn as sns
sns.boxplot(orient="h", palette="pastel", y=df["format"],
                    x=df["coherence"], hue=df["seglen"])
```

Code 33: Erstellung von kategorisierten Box-Plots

Den X- und Y-Parametern werden die jeweiligen Spalten zugeordnet und mit `hue` wird bestimmt, welcher Wert für die Kategorisierung der Unter-Plots gewählt wird. Mit `orient` wird zwischen vertikaler und horizontaler Ausrichtung entschieden und mit `palette` wird eine Farbpalette für die Boxen ausgewählt.

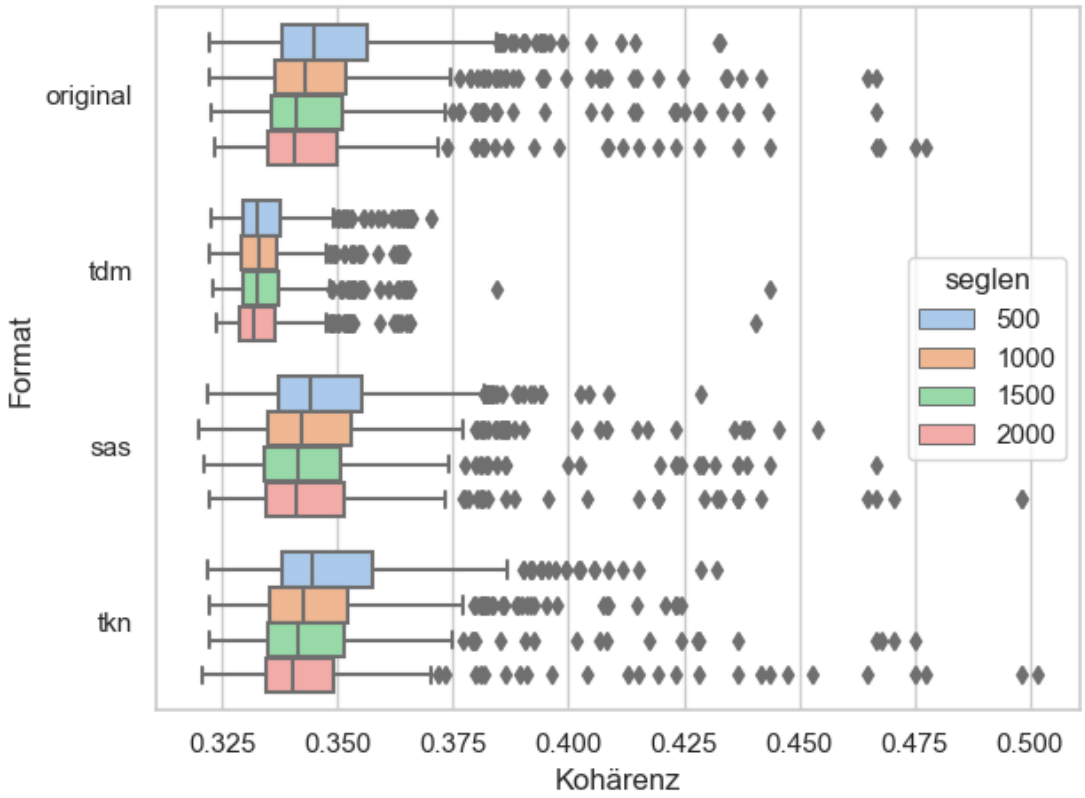


Abbildung 17: Box-Plots aller Formate im Vergleich

Jede Box-Plot-Gruppe repräsentiert ein Format und die erreichten Kohärenzen sind, wie bei den KDE-Plots, auf der X-Achse eingetragen. Eine Box repräsentiert die mittleren 50% der erreichten Werte und die in ihr befindliche vertikale Linie ist der Mittelwert. Die horizontalen Linien (sogenannte *Whiskers*) visualisieren die unteren

und oberen 25% der Daten. Sie repräsentieren auch die Extrema des Datensatzes am jeweiligen Ende mit einer vertikalen Linie. Die Punkte außerhalb des Boxplots sind Ausreißer, die mit 0.7% nur einen äußerst geringen Anteil der Daten darstellen.

Nun kann anhand dieser Grafik visuell bestätigt werden, was in den obigen Kapiteln beschrieben wurde. Wenn sich der Median eines Plots nicht auf gleicher Höhe einer danebenliegenden Box befindet, wird angenommen dass es einen signifikanten Unterschied zwischen den Daten gibt. Das ist ähnlich zur visuellen Auswertung von Konfidenzintervallen, denn die Mehrheit der Daten erzielt andere Werte als in der Vergleichsmenge. Es ist zu sehen, dass nur die Boxen der TDM sich deutlich dadurch von den anderen unterscheiden, dass sie kleinere Kohärenzwerte umspannen und schmaler sind. Ihre Mediane sowie Maxima liegen ebenfalls deutlich unter denen der anderen Modellen. Interessant ist, dass die Minima geringfügig höher sind, denn auch hier zeichnet sich ab, dass die Werte viel dichter beieinanderliegen. Die Boxen der restlichen Modelle hingegen sind sich sehr ähnlich: Mediane liegen pro Segmenteinheit etwa auf der gleichen Höhe und die Boxen haben eine ähnliche Breite.

Bei allen Formaten außer der TDM ist bei wachsender Segmentgröße ein erwartbarer Abwärtstrend zu beobachten, denn ihre Mittelwerte und ersten Quartile sind entweder geringer oder etwa gleich groß, wie die Boxen der SAS mit Segmentlängen 1500 und 2000 zeigen. Das zeigt sich durch eine leicht stufige Verschiebung der Boxen, doch die Werte scheinen zumindest im ausgewählten Längenbereich nicht signifikant verschieden zu sein. Bei der TDM ist dieser Abwärtstrend hier nicht zu sehen. Die Segmentlänge scheint keine signifikanten Einfluss auf die Kohärenz auszuüben. Das erscheint logisch, da die Segmente in der TDM durch die zufällige Neuanordnung keine natürlichen Kookurrenzen aufweisen können und die daraus ermittelten Kohärenzwerte ähnlich sind.

Es fällt auf, dass alle Modelle Ausreißer nur in positive Richtung besitzen, die teilweise mit Werten zwischen 0.4 bis 0.5 deutlich höher sind, als die Mehrheit der Daten. Interessanterweise erreichen die Ausreißer in der höchsten Segmentlänge die größten Kohärenzen. Es zeigt sich, dass die größten Kohärenzen vom Seefahrt-Topic erreicht werden. Der höchste Wert von 0.502 wurde von einem Topic im TKN-Format mit Segmentlänge 2000 erreicht. Es ist Teil der 16. Iteration und ist für einen solch ho-

hen Wert erstaunlich inkonsistent: `ship dodo boat sea water sail deck time captain vessel`. Das Wort `dodo` sticht besonders heraus, weil es aus semantischer Sicht (ausgestorbene Vogelart) nicht gut in diese Wortgruppe passt und `time` ist ein sehr unspezifisches Wort. Betrachtet man dagegen das gleiche Topic mit niedrigerem Wert (0.45) aus Iteration 15, scheint es doch aus menschlicher Sicht deutlich kohärenter zu sein: `ship water boat sea river sail captain deck wind land`. Jedes Wort hat eine starke Verbindung zur Seefahrt. Das zeigt, dass die Ergebnisse aus Palmetto mit der englischen Wikipedia als Referenz kritisch zu hinterfragen sind und die menschliche Auswertung nicht immer mit dem Kohärenzwert korreliert.

Wenig überraschend zeigen t-Tests mit diesen Daten (aggregiert über Segmentgrenzen hinweg) auch vergleichbare Ergebnisse wie bereits oben dargestellt. Im Vergleich mit dem Original ist der p-Wert der TDM sogar noch kleiner mit 110 Nachkommastellen, der der SAS mit 0.91 geringfügig höher und der der TKN mit 0.99 gleich.

5.4.7 Diagnostik-XML

Ergänzend zu den Kohärenzen werden nun zunächst die Diagnostik-XMLs eingelesen und die Werte `document_entropy`, `uniform_dist` und `corpus_dist` in Boxplots visualisiert und ausgewertet.

Abbildung 18 zeigt das Box-Plot, welches aus den Werten von `document_entropy` erstellt wurde. Wie in Kapitel 2.4.3 beschrieben, weist ein hoher Wert darauf hin, dass ein Topic in vielen Dokumenten vertreten ist.

Tabelle 14 zeigt die statischen Eckdaten in Zahlen. Die Minima der TDM, der SAS und den TKN sind nahezu gleich und die TDM weist geringfügig höhere Minima auf. Dass die Entropie der TDM jedoch hauptsächlich etwas geringer ist, als das der anderen Formate, entspricht nicht den Erwartungen, da dies bedeutet, dass die Topics fast so gleichmäßig verteilt sind, wie bei den anderen Formaten. Die Werte für Original und TKN sind auf die zweite Dezimalstelle gerundet identisch und das der SAS ist größer.

Ein wenig überraschend sind die Ergebnisse der t-Tests. Die p-Werte für den Ver-

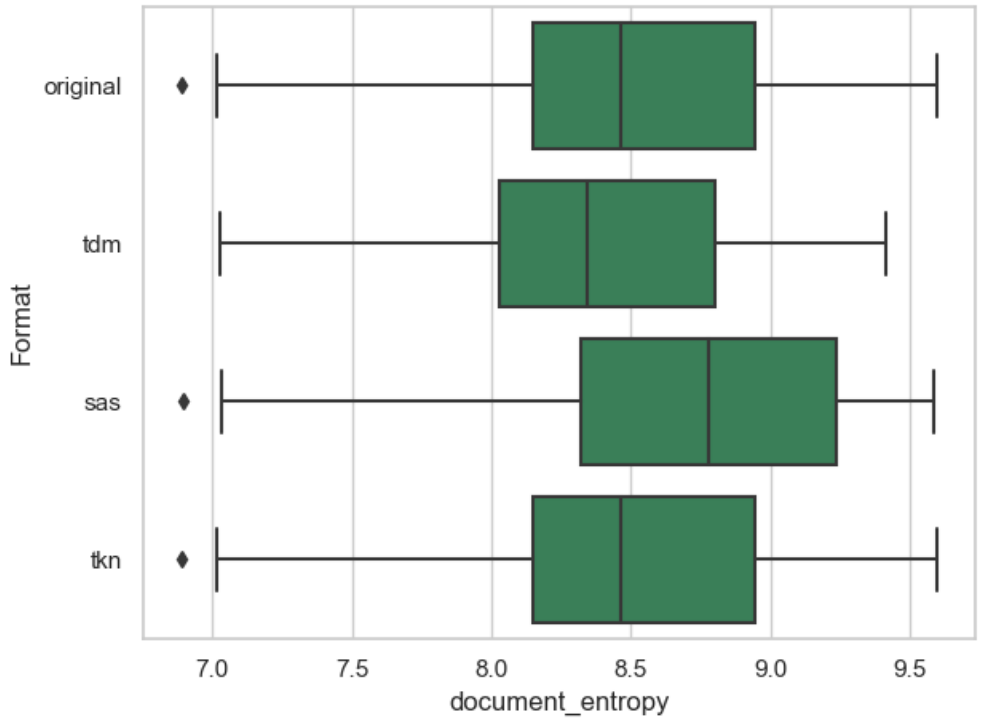


Abbildung 18: Box-Plots zum Attribut `document_entropy` aus den Diagnostik-XMLs

format	min	max	mean	1. quantile	3. quantile
original	6.89	9.60	8.52	8.15	8.94
tdm	7.02	9.41	8.42	8.03	8.80
sas	6.90	9.59	8.73	8.32	9.23
tkn	6.89	9.60	8.52	8.15	8.94

Tabelle 14: Extrema und Durchschnittswerte der Dokumententropien

gleich mit der TDM ist mit 8 führenden Nullen sehr klein und damit signifikant anders, aber die Werte der SAS sind auch signifikant höher, da der p-Wert 19 führende Nullen vorweist. Damit ist die Dokumententropie der SAS höher, als die des Originals. Allerdings scheint sich dieser Wert nicht maßgeblich mit den Kohärenzwerten zu korrelieren. Das TKN-Format liefert im Vergleich den p-Wert 1.0 zurück, womit die Entropie als identisch bestätigt ist.

Als nächstes werden die Werte von `uniform_dist` betrachtet. Sie messen die Distanz der Wortverteilung eines Topics zu einer Uniformverteilung. Je höher die errechnete Distanz, desto spezifischer das Topic. Es wird erwartet, dass sie sich ähnlich zu den Kohärenzen verhalten, denn bei der Betrachtung der Topics fiel bereits auf, dass die Wiederholungen der Topic-Wörter bei der TDM besonders hoch war und damit die

Topics unspezifischer wirken. Abbildungs 19 zeigt die dazugehörigen Box-Plots.

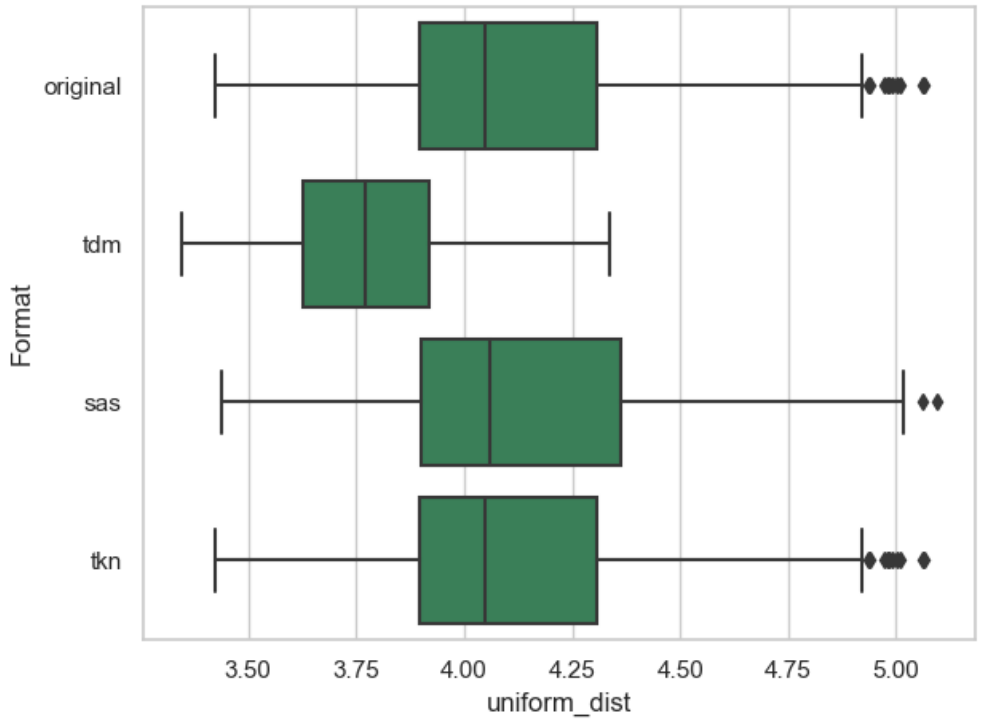


Abbildung 19: Box-Plots zum Attribut `uniform_dist` aus den Diagnostik-XMLs

Hier scheint sich grafisch widerzuspiegeln, was bereits bei den Kohärenzen zu sehen war: die TDM weicht deutlich ab und die Werteverteilung der restlichen Modelle sind sich ähnlich. In Tabelle 15 finden sich die wichtigsten Eckdaten wieder. Der t-Test bestätigt diesen Verdacht.

format	min	max	mean	1. quantile	3. quantile
original	3.42	5.06	4.11	3.90	4.30
tdm	3.35	4.33	3.78	3.63	3.91
sas	3.44	5.09	4.14	3.90	4.36
tkn	3.42	5.06	4.11	3.90	4.31

Tabelle 15: Extrema und Durchschnittswerte der Uniformdistanzen

Es ist zu sehen, dass die Minima sich alle unter 3.5 befinden und das Minimum der TDM ist mit 3.35 lediglich um etwa 2% geringer als das des Originals (3.42). Das Maximum der TDM ist mit Wert 4.33 um 16.86% geringer als das des Originals (5.06). Die anderen Formate liegen etwa gleichauf. Auch mit dem Durchschnittwert 3.78 ist die TDM um 8.73% geringer als das Original (4.11). Daraus lässt

sich schließen, dass die unspezifischsten Topics aller Formate sich nicht stark voneinander unterscheiden, allerdings insgesamt signifikant mehr Topics aller Formate spezifischer sind, als die der TDM. Die spezifischsten Topics allerdings erzielt die SAS, dessen Maximaldistanz mit 4.36 sogar um 1.4% spezifischer sind, als die des Originals (4.30).

Der p-Wert ist im Vergleich zwischen Original und TDM mit 242 führenden Nullen erwartungsgemäß klein. Der p-Wert im Vergleich zwischen Original und SAS beträgt 0.057 und liegt damit über der Signifikanzgrenze von 0.05, aber ist nicht so deutlich größer, wie bei den Kohärenzen. Der p-Wert des TKN-Formats liegt hier wieder bei 1.0 und damit ist der Datensatz identisch.

Als letztes werden die Werte der `corpus_dist` betrachtet. Sie sind ebenfalls errechnete Distanzen. Wie der Name andeutet, wird die Distanz der Wortverteilungen des Topics zur Verteilung des Korpus berechnet. Je größer die Entfernung, desto distinktiver das Topic.

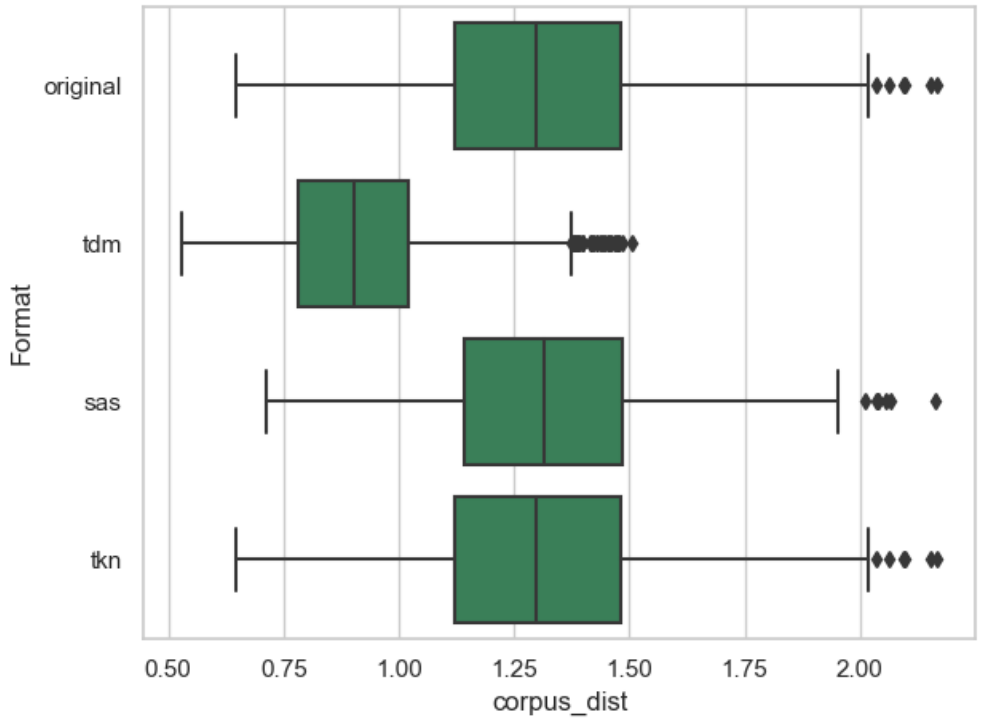


Abbildung 20: Box-Plots zum Attribut `corpus_dist` aus den Diagnostik-XMLs

In Abbildung 20 sind die Boxplots zur Korpusdistanz abgebildet. Die Eckdaten sind in Tabelle 16 aufgelistet.

format	min	max	mean	1. quantile	3. quantile
original	0.65	2.17	1.30	1.12	1.48
tdm	0.53	1.51	0.92	0.78	1.02
sas	0.71	2.16	1.32	1.14	1.48
tkn	0.65	2.16	1.30	1.12	1.48

Tabelle 16: Extrema und Durchschnittswerte der Uniformdistanzen

Die Korpusdistanzwerte des Originals und des TKN-Formats sind auch hier wieder gleich. Die Minimumdistanz der TDM liegt mit 0.53 um 22.64% niedriger als das Minimum des Originals (0.65). Das Minimum der SAS ist mit 0.71 um 8.45% größer. Die Maximaldistanz der TDM ist mit 1.51 sogar um 43.71% geringer als die des Orginals (2.17) und der Durchschnitt ist mit 0.92 um 41.30% geringer (Original: 1.30). Das zeigt, dass die TDM auch in Bezug zum Korpus deutlich weniger distinktive Topics erreicht, während die anderen Formate wieder etwa gleichauf liegen.

Die p-Werte sind nicht überraschend und zeigen auch, dass die Werte der TDM signifikant niedriger (0.0), die des TKN-Formats gleich (1.0) und die der SAS höher (0.14), aber nicht signifikant höher sind.

6 Fazit

Ziel dieser Arbeit war es empirisch zu zeigen, ob und in welchem Maße sich die abgeleiteten Textformate *Term-Dokument-Matrix*, *segmentweise Aufhebung von Sequenzinformationen*, *selektiv reduzierte Informationen über einzelne Tokens* und *N-Gramme* für Topic Modeling eignen. Dafür wurden diese zunächst aus einem englischen Literaturkorpus des 19. und frühen 20. Jahrhundert erstellt und skizziert, wie diese mit Python-Code und spezialisierten Bibliotheken vorverarbeitet wurden.

Es ist deutlich zu erkennen, dass die Formate *segmentweise Aufhebung von Sequenzinformationen* und *selektiv reduzierte Informationen über einzelne Tokens* außerordentlich gute Ergebnisse erzielen, weil die mit ihnen erreichten Kohärenzwerte nicht signifikant schlechter sind, als die des Volltextes. Die Topic-Strukturen, Dokumententropie, Uniform- und Korpusdistanzen sind ebenfalls ähnlich oder gleich.

Das Format *Term-Dokument-Matrix* hingegen ist für Topic Modeling nutzbar, allerdings nur in beschränktem Maße. Die erreichten Kohärenzen sind signifikant niedriger als die des Volltextes und die Topic-Wortkompositionen sind deutlich unspezifischer und damit schlechter interpretierbar. Das zeigt sich nicht nur an den Kohärenzwerten, sondern auch an der Dokumententropie und den Uniform- sowie Korpusdistanzen. Sie kann aber einen groben Überblick über den Inhalt eines Korpus bieten. Trotzdem ist sie nicht für diesen Zweck zu empfehlen, wenn die Möglichkeit besteht, stattdessen die beiden o.g. Textformate zu nutzen.

Das Format *N-Gramme* stellt sich tatsächlich als problematisch heraus und ist nicht für Topic Modeling zu empfehlen. Die in dieser Untersuchung durchgeführte Formatserstellung und die damit einhergehende Reduktion des Textes auf bestenfalls 0.6% des vorverarbeiteten Volltextes, ist als Input für LDA-Algorithmen sinnlos. Als Anschlussforschung böte sich daher an, zu prüfen, ob sich N-Gramme ohne die Einschränkung von Mindesthäufigkeiten besser eignen. Da hierdurch allerdings die Gefahr besteht, dass größere Textpassagen wieder rekonstruiert werden könnten, sollte die Wortreihenfolge, ähnlich wie bei der segmentweisen Aufhebung von Sequenzinformationen, randomisiert werden. Damit könnten verwertbare Ergebnisse erzielt werden. Allerdings stellt sich trotzdem die Frage der Nützlichkeit, wenn die anderen Textformate mit deutlich weniger Aufwand erstellt werden können, aber

trotzdem sehr gute Ergebnisse liefern.

Für die weitere Forschung bietet es sich an, den vorgestellten Workflow zu kopieren, zu verfeinern und die Textformate noch mit anderen Parametern, wie der Erweiterung oder Reduzierung der Wortartenfilter, zu untersuchen. Für Archive, die abgeleitete Textformate für Topic Modeling anbieten möchten, sei ganz klar die segmentweise Aufhebung von Sequenzinformation zu empfehlen. Sie bietet den enormen Vorteil, dass keinerlei Informationen für LDA-Algorithmen verloren gehen und überflüssige Information von Forschenden ggf. selbst entfernt werden können. Allerdings sollte dann darauf geachtet werden, dass die Rekonstruierbarkeit durch eine zu kleine Segmentgröße nicht ermöglicht wird.

Letztendlich bleibt es interessant, ob es eine Zukunft für alternative Textformate in der Forschung gibt und wie sie sich gestalten wird. Die künftige Entwicklung kann mit Spannung beobachtet werden.

Literatur

Bevans, R. (2020). *Confidence intervals explained* [Scribbr]. Verfügbar 6. August 2021 unter https://www.scribbr.com/statistics/confidence-interval/

Blei, D. M. (2012). Topic Modeling and Digital Humanities. *Journal of Digital Humanities*, *2*(1). Verfügbar 28. Januar 2021 unter http://journalofdigitalhumanities.org/2-1/topic-modeling-and-digital-humanities-by-david-m-blei/

Blei, D. M., Ng, A. Y. & Jordan, M. I. (2003). Latent Dirichlet Allocation. *Journal of Machine Learning Research*, *3*, 993–1022. Verfügbar 8. März 2021 unter https://www.jmlr.org/papers/volume3/blei03a/blei03a.pdf

Block, S. (2020). Doing More with Digitization - Commonplace. *Commonplace: the journal of early American life*. Verfügbar 28. Januar 2021 unter http://commonplace.online/article/doing-more-with-digitization/

Burnard, L., Odebrecht, C. & Schöch, C. (03/05/2021). *ELTeC-eng: TEI XML Sources for the English novel part of the ELTeC* (L. Burnard, Hrsg.). Verfügbar 3. Mai 2021 unter https://github.com/COST-ELTeC/ELTeC-eng

Chang, J., Boyd-Graber, J., Wang, C., Gerrish, S. & Blei, D. M. (2009). Reading Tea Leaves: How Humans Interpret Topic Models. *Neural Information Processing Systems*. http://umiacs.umd.edu/~jbg//docs/nips2009-rtl.pdf

Clark, J. (1999). *XML Namespaces*. Verfügbar 6. Mai 2021 unter http://www.jclark.com/xml/xmlns.htm

Cokelaer, T. (n. d.). *FITTER documentation*. Verfügbar 2. August 2021 unter https://fitter.readthedocs.io/en/latest/index.html

Distant Reading for European Literary History. (03/05/2021). *Distant Reading for European Literary History: COST Action CA16204*. Verfügbar 3. Mai 2021 unter https://www.distant-reading.net/

Hunter, J. D. (2007). Matplotlib: A 2D graphics environment. *Computing in science & engineering*, *9*(3), 90–95.

Jockers, M. L. (n. d.). *Expanded stopwords list*. Verfügbar 5. Juli 2021 unter https://www.matthewjockers.net/macroanalysisbook/expanded-stopwords-list/

Konzack, M. (2019). *Parsing TEI XML documents with Python*. Verfügbar 14. April 2021 unter https://komax.github.io/blog/text/python/xml/parsing_tei_xml_python/

Lange, S. & Bender, R. (2001). Was ist ein Signifikanztest? *DMW - Deutsche Medizinische Wochenschrift*, *126*, T 42–T 44.

Lexical Computing CZ s.r.o. (2017). *English Penn Treebank tagset with modifications*. Verfügbar 15. Juni 2021 unter https://www.sketchengine.eu/english-treetagger-pipeline-2/

McCallum, A. K. (2002). MALLET: A Machine Learning for Language Toolkit [http://mallet.cs.umass.edu].

McCallum, A. K. (04/05/2021). *Topic model diagnostics*. Verfügbar 4. Mai 2021 unter http://mallet.cs.umass.edu/diagnostics.php

McKinney, W. (2010). Data Structures for Statistical Computing in Python. *Proceedings of the 9th Python in Science Conference*, 56–61. https://doi.org/10.25080/Majora-92bf1922-00a

Meeks, E. & Weingart, S. (2012). The Digital Humanities Contribution to Topic Modeling. In D. J. Cohen & J. F. Troyano (Hrsg.), *Journal of Digital Humanities* (S. 2–5).

Mimno, D. (2020-02-25). *Using phrases in Mallet topic models*. Verfügbar 21. Juni 2021 unter http://www.mimno.org/articles/phrases/

Prel, J.-B. d., Hommel, G., Röhrig, B. & Blettner, M. (2009). Confidence Interval or P-Value? Part 4 of a Series on Evaluation of Scientific Publications. *Deutsches Aerzteblatt Online*. https://doi.org/10.3238/arztebl.2009.0335

Projekt Zeta Team. (2021). *Projekt – Zeta and Company*. Verfügbar 17. Juni 2021 unter https://zeta-project.eu/de/projekt/

Řehůřek, R. (2021). *Models.ldamulticore – parallelized latent dirichlet allocation*. Verfügbar 9. Juli 2021 unter https://radimrehurek.com/gensim/models/ldamulticore.html

Řehůřek, R. & Sojka, P. (2010). Software Framework for Topic Modelling with Large Corpora [May, http://is.muni.cz/publication/884893/en]. *Proceedings of the LREC 2010 Workshop on New Challenges for NLP Frameworks*, 45–50.

Röder, M. (n. d.). *Palmetto: Palmetto is a quality measuring tool for topics*. Verfügbar 8. Juni 2021 unter https://aksw.org/Projects/Palmetto.html

Röder, M., Both, A. & Hinneburg, A. (2015). Exploring the Space of Topic Coherence Measures [Backup Publisher: International Conference on Web Search and Data Mining Num Pages: 10 event-place: New York, NY]. In X. Cheng,

H. Li, E. Gabrilovich & J. Tang (Hrsg.), *Proceedings of the Eighth ACM International Conference on Web Search and Data Mining , Shanghai, China 31. Januar - 06. Februar 2015* (S. 399–408). ACM. https://doi.org/10.1145/2684822.2685324

Ruchirawat, N. (2020). 6 Tips for Interpretable Topic Models - Towards Data Science. *Towards Data Science*. Verfügbar 13. Mai 2021 unter https://towardsdatascience.com/6-tips-to-optimize-an-nlp-topic-model-for-interpretability-20742f3047e2

Schmid, H. (1994). Probabilistic Part-of-Speech Tagging Using Decision Trees. *Proceedings of International Conference on New Methods in Language Processing*. Verfügbar 7. Juni 2021 unter https://www.cis.lmu.de/~schmid/tools/TreeTagger/data/tree-tagger1.pdf

Schmid, H. (1995). Improvements in Part-of-Speech Tagging with an Application to German. *Proceedings of the ACL SIGDAT-Workshop*. Verfügbar 7. Juni 2021 unter https://www.cis.lmu.de/~schmid/tools/TreeTagger/data/tree-tagger2.pdf

Schmidt, B. M. (2012). Words Alone: Dismantling Topic Models in the Humanities. In D. J. Cohen & J. F. Troyano (Hrsg.), *Journal of Digital Humanities* (S. 48–65).

Schöch, C. (2016). *Topic modeling with MALLET: Hyperparameter optimization* [The dragonfly's gaze]. Verfügbar 28. Juli 2021 unter https://dragonfly.hypotheses.org/1051

Schöch, C. (2017). Topic Modeling Genre: An Exploration of French Classical and Enlightenment Drama. *Digital Humanities Quarterly*, *11*(2).

Schöch, C., Döhl, F., Rettinger, A., Gius, E., Trilcke, P., Leinen, P., Jannidis, F., Hinzmann, M. & Röpke, J. (2020). Abgeleitete Textformate: Text und Data Mining mit urheberrechtlich geschützten Textbeständen. https://doi.org/10.17175/2020_006

Schöch, C., Erjavec, T., Patras, R. & Santos, D. (2021). *Creating the European Literary Text Collection (ELTeC): Challenges and Perspectives*. https://doi.org/10.5281/zenodo.4742420

Seabold, S. & Perktold, J. (2010). statsmodels: Econometric and statistical modeling with python. *9th Python in Science Conference*.

Sieg, C. (2019). Topic Modeling von Fallgeschichten [PII: 153]. *Zeitschrift für Literaturwissenschaft und Linguistik*, *49*(4), 653–671. https://doi.org/10.1007/s41244-019-00153-z

spaCy Dev Team. (n. d. a). *Industrial-strength natural language processing in python*. Verfügbar 23. Juni 2021 unter https://spacy.io/

spaCy Dev Team. (n. d. b). *Language processing pipelines* [Language processing pipelines]. Verfügbar 23. Juni 2021 unter https://spacy.io/usage/processing-pipelines

The pandas development team. (2020). pandas-dev/pandas: Pandas [feb,]. https://doi.org/10.5281/zenodo.3509134

The SciPy community. (2021). *t-Test Dokumentation*. Verfügbar 2. August 2021 unter https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.ttest_ind.html#scipy.stats.ttest_ind

University of Oxford. (2019). *Oxford Text Archive*. Verfügbar 3. Mai 2021 unter https://ota.bodleian.ox.ac.uk/repository/xmlui/

University of Stuttgart. (2021). *Project Textual corpora and tools for their exploration*. Verfügbar 15. Juni 2021 unter https://www.ims.uni-stuttgart.de/en/research/projects/textkorpora-werkzeuge/

vDHd2021 - Experiments Team. (2021). *Workshop „Kontrastive Analyse literarischer Texte mit Zeta". Einführung in die Implementierung und Evaluation von Distinktivitätsmaßen*. Verfügbar 17. Juni 2021 unter https://vdhd2021.hypotheses.org/185

Virtanen, P., Gommers, R., Oliphant, T. E., Haberland, M., Reddy, T., Cournapeau, D., Burovski, E., Peterson, P., Weckesser, W., Bright, J., van der Walt, S. J., Brett, M., Wilson, J., Millman, K. J., Mayorov, N., Nelson, A. R. J., Jones, E., Kern, R., Larson, E., … SciPy 1.0 Contributors. (2020). SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*, *17*, 261–272. https://doi.org/10.1038/s41592-019-0686-2

Waskom, M. L. (2021). seaborn: statistical data visualization. *Journal of Open Source Software*, *6*(60), 3021. https://doi.org/10.21105/joss.03021

Weingart, S. (2011). *Topic Modeling and Network Analysis*. Verfügbar 4. Mai 2021 unter http://www.scottbot.net/HIAL/index.html@p=221.html

Weitin, T. & Herget, K. (2017). Falkentopics [PII: 49]. *Zeitschrift für Literaturwissenschaft und Linguistik*, *47*(1), 29–48. https://doi.org/10.1007/s41244-017-0049-3

Xu, J. (2018). Topic Modeling with LSA, PLSA, LDA & lda2Vec - NanoNets - Medium. *Medium*, *2018*. Verfügbar 18. Mai 2021 unter https://medium.com/nanonets/topic-modeling-with-lsa-psla-lda-and-lda2vec-555ff65b0b05

Anhang

Tabelle 17: Penn-Tagset (Quelle: Lexical Computing CZ s.r.o., 2017)

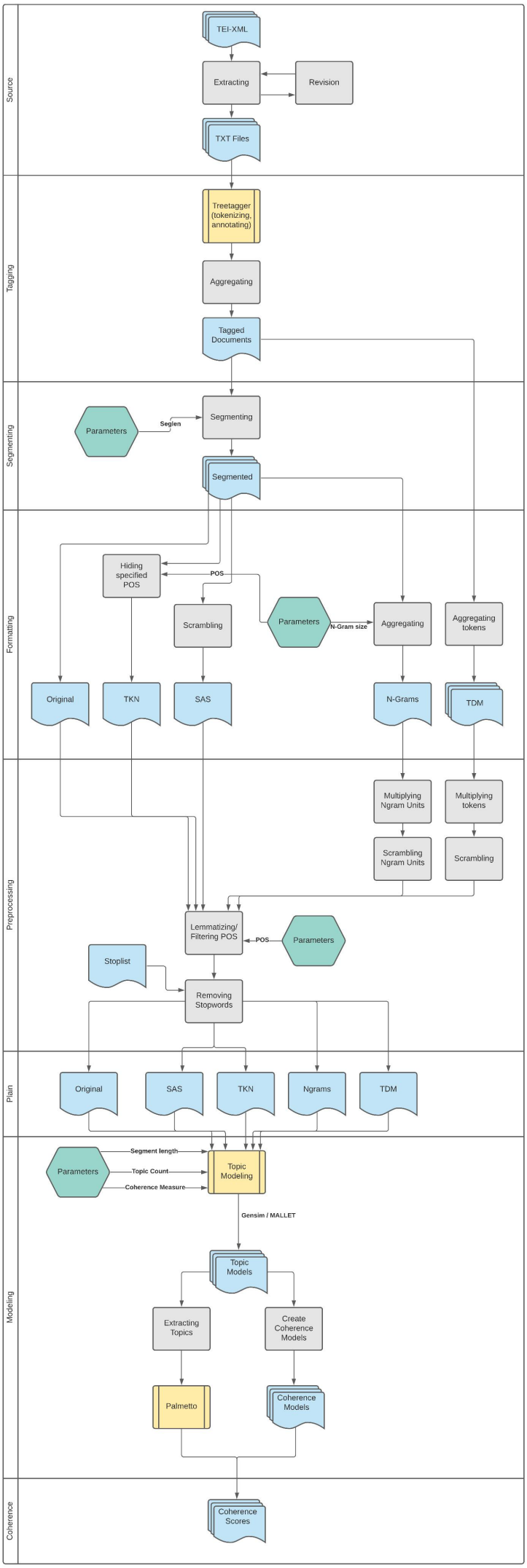
POS Tag	Description
CC	coordinating conjunction
CD	cardinal number
CDZ	possesive pronoun
DT	determiner
EX	existential there
FW	foreign word
IN	preposition, subordinating conjunction
IN/that	that as subordinator
JJ	adjective
JJR	adjective, comparative
JJS	adjective, superlative
LS	list marker
MD	modal
NN	noun, singular or mass
NNS	noun plural
NNSZ	possessive noun plural
NNZ	possessive noun, singular or mass
NP	proper noun, singular
NPS	proper noun, plural
NPSZ	possesive proper noun, plural
NPZ	possesive noun, singular
PDT	predeterminer
PP	personal pronoun
PPZ	possessive pronoun
RB	adverb
RBR	adverb, comparative
RBS	adverb, superlative
RP	particle
SENT	Sentence-break punctuation
SYM	Symbol
TO	infinitive ‘to’
UH	interjection
VB	verb be, base form
VBD	verb be, past tense
VBG	verb be, gerund/present participle

Fortsetzung auf nächster Seite

Tabelle 17 – Fortsetzung aus vorheriger Seite

POS Tag	Description
VBN	verb be, past participle
VBP	verb be, present, non-3d person
VBZ	verb be, 3rd person sing. present
VH	verb have, base form
VHD	verb have, past tense
VHG	verb have, gerund/present participle
VHN	verb have, past participle
VHP	verb have, sing. present, non-3d
VHZ	verb have, 3rd person sing. present
VV	verb, base form
VVD	verb, past tense
VVG	verb, gerund/present participle
VVN	verb, past participle
VVP	verb, present, not 3rd person
VVZ	verb, 3rd person sing. present
WDT	wh-determiner
WP	wh-pronoun
WPZ	possessive wh-pronoun
WRB	wh-abverb
Z	possessive ending

Vollständige Abbildung des Workflows


Source
TEI-XML
Extracting
Revision
TXT Files
Tagging
Treetagger (tokenizing, annotating)
Aggregating
Tagged Documents
Segmenting
Parameters
SegLen
Segmenting
Segmented
Formatting
Hiding specified POS
POS
Scrambling
Parameters
N-Gram size
Aggregating
Aggregating tokens
Original
TKN
SAS
N-Grams
TDM
Preprocessing
Multiplying Ngram Units
Multiplying tokens
Scrambling Ngram Units
Scrambling
Lemmatizing/ Filtering POS
POS
Parameters
Stoplist
Removing Stopwords
Plain
Original
SAS
TKN
Ngrams
TDM
Modeling
Parameters
Segment length
Topic Count
Coherence Measure
Topic Modeling
Gensim / MALLET
Topic Models
Extracting Topics
Create Coherence Models
Palmetto
Coherence Models
Coherence
Coherence Scores

Eidesstattliche Erklärung

Ich versichere, die von mir vorgelegte Arbeit selbstständig verfasst zu haben. Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder nicht veröffentlichten Arbeiten anderer entnommen sind, habe ich als entnommen kenntlich gemacht. Sämtliche Quellen und Hilfsmittel, die ich für die Arbeit benutzt habe, sind angegeben. Die Arbeit hat mit gleichem Inhalt bzw. in wesentlichen Teilen noch keiner anderen Prüfungsbehörde vorgelegen.

$Unterschrift$: $Ort, Datum$: