Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

Projektarbeit

Exploration of News

vorgelegt von

Tatyana Galitskaya

Sara Yüksel

Alexander Spikofsky

MIN Fakultät

Fachbereich: Scientific Computing

Studiengang: B.Sc. Wirtschaftsinformatik

Betreuer: Dr. Julian Kunkel

Tatyana Galitskaya , Matrikelnr. 6613032, tatyana.galitskaya@gmx.de

Sara Yüksel, Matrikelnr. 6797672, sara.y@web.de

Alexander Spikofsky, Matrikelnr. 6790864, alexander.spikofsky@gmail.com

Inhaltsverzeichnis

1. Einleitung

Die Aufgabenstellung unseres Big-Data-Projektes „Exploration of News“ ist es, US-amerikanische, britische und australische Newsartikel zu sammeln, relevante Daten zu identifizieren und zu extrahieren, diese zwischen den Ländern und zwischen den Papern zu vergleichen und hingehend ihrer Ähnlichkeit zu untersuchen. Ziel der Arbeit ist es, Erkenntnisse bezüglich der Gemeinsamkeiten und Unterschiede der Artikel – vor allem im Hinblick auf die Herkunftsländer – abzuleiten.

Sowohl zum Sammeln als auch zur Analyse und Visualisierung aller Daten wird Python als Programmiersprache eingesetzt. Ein uns zu Verfügung gestellter Crawler übernimmt dabei das Zusammentragen von großen Mengen an zu untersuchenden Artikeln. Ist ein aussagekräftiges Volumen erreicht, beginnt die Bearbeitung und Auswertung der Artikel. Die Analyse wird kleinschrittig vorgenommen und im Laufe des Projektes nach Möglichkeit erweitert und an die Daten angepasst. Alle relevanten Ergebnisse werden in verschiedenen Grafiken visualisiert, um daraus ableitbare Schlüsse erkennbar zu machen.

2. Tools

Im Kapitel „Tools“ beschreiben wir die theoretischen Grundlagen und generellen Funktionsweisen der von uns eingesetzten Tools im Rahmen unseres Projektes und in welchen Bereichen diese Verwendung finden (für Quellen siehe Quellenverzeichnis unter *Bibliotheken und Dokumentationen*).

Python

Python ist eine dynamische Programmiersprache, die aufgrund ihrer Einfachheit und der leicht verständlichen Syntax eine leicht zu erlernende Sprache darstellt. Aufgrund ihrer vielen zur Verfügung stehenden Standardbibliotheken (auf welche wir im Folgenden noch genauer eingehen werden) und Module in den Bereichen Datenanalyse und semantische Textanalyse eignet sie sich besonders gut für die Untersuchung von textuellen Nachrichten und stellt eine Alternative zur Programmiersprache R da. Ein weiterer großer Vorteil ist, dass Python plattformunabhängig läuft, was bedeutet, dass es sowohl unter Linux als auch unter verschiedenen Unix Systemen ohne große Anpassungen des Programmcodes funktioniert.

Anaconda

Anaconda ist eine Open Source Distribution für Programmiersprachen wie Python und R. Die Software beinhaltet einen Python-Interpreter. Aufgrund dessen ist es egal, auf welchem System sie ausgeführt wird. Zusätzlich beinhaltet Anaconda ein webbasiertes Frontend für Jupyter Notebook.

Das Hauptaugenmerk bei Anaconda liegt auf der Verarbeitung von großen Datenmengen und Datenanalysen.

Jupyter Notebook

Jupyter Notebook ist ein webbasierter Kommandozeileninterpreter für die Programmiersprache Python. Innerhalb dieses Interpreters ist das Entwickeln und Ausführen von Python-Programmen und -Code möglich.

Web Crawler

Wie der Name Web Crawler vermuten lässt handelt es sich um ein Computerprogramm, welches immer wiederkehrende Aufgaben durchführt. Ein Web Crawler durchsucht das World Wide Web und analysiert und speichert Daten nach Kriterien die vorher definiert worden sind. In unserem speziellen Fall werden die von uns vorher definierten RSS-Feeds durchsucht und an einem Ort gespeichert.

BeautifulSoup

BeautifulSoup ist eine frei zur Verfügung stehende Bibliothek in Python mit der sich HTML und XML Dateien zerlegen lassen. Dabei wird die Datei in die Formate konvertiert, welche für die nächsten Schritte benötigt werden, sodass diese weiterverarbeitet werden können. In unserem Fall wird die Datenstruktur der von uns gecrawlten HTML-Seiten in einen Reintext gewandelt, welcher nur noch den Text-Body beinhaltet.

Numpy

Numpy ist wie BeautifulSoup eine frei zur Verfügung stehende Bibliothek in Python. Dabei stellt Numpy hauptsächlich eine Implementierung von Funktionen bereit, die für numerische und wissenschaftliche Berechnungen benötigt wird. Des Weiteren ermöglicht die Bibliothek eine leichtere Handhabung und Arbeitsweise mit Matrizen, Vektoren und mehrdimensionalen Arrays. Im Vergleich zu Standard Python Funktionen, ist Numpy um eine ganze Ecke schneller in der Berechnung und Arbeit mit diesen.

Pandas

Pandas ist ebenfalls eine Bibliothek in Python. Sie bietet hauptsächlich Datenstrukturen und Operationen zur Bearbeitung von Tabellen an. Die drei wichtigsten Objekte dieser Bibliothek sind Series, Dataframes und Panels. Im Folgenden Verlauf unseres Projektes benutzen wir vor allem die Dataframes, welche zweidimensionale Tabellen darstellen.

NLTK für Natural Language Processing

Das Natural Language Toolkit ist eine Bibliothek, die hauptsächlich zu Textmining-Zwecken und zur Textanalyse benutzt wird. Das Modul beinhaltet bereits kleinere Visualisierungsmöglichkeiten um beispielsweise erste Baumstrukturen auf analysierten Text anzuwenden. Zusätzlich unterstützt die Bibliothek nützliche Funktionen wie die Lemmatisierung, Part of Speech Analysemethoden oder Tokenization auf Texten.

Scikit Learn

Scikit Learn ist ein Framework, welches Funktionen zum maschinellen Lernen enthält und Python damit erweitert. In dieser Bibliothek sind zum Beispiel Algorithmen für das Clustering, die Klassifikation und die Regression enthalten.

Matplotllib.pyplot

Matplotllib erweitert die Numpy Bibliothek um grafische Darstellungsformen in 2D sowie in 3D und anderen Formaten. Die Kombination aus beiden Bibliotheken ist die am weitesten verbreitete Bibliothekskombination, um Daten in Python grafisch darzustellen. Ein weiterer Vorteil ist, dass in matplotllib objektorientiert programmiert werden kann.

Seaborn

Seaborn ist eine Visualisierungsbibliothek, welche auf matplotllib aufsetzt und diese um weitere Funktionen erweitert und ergänzt. Die Bibliothek beinhaltet ein high level Interface mit dessen Hilfe sich ansprechende statistische Grafiken erstellen lassen.

3. Realisierung

Dieses Kapitel behandelt die praktische Umsetzung der Arbeit. Im Unterkapitel „Design“ erläutern wir den Entwurf der einzelnen Projektbereiche und wie die Arbeitsschritte geplant waren. In „Implementierung“ wird die tatsächliche Realisierung in Python-Code beschrieben und die Ergebnisse der Analysen inklusive der Visualisierungen werden dargestellt. Dazu werden abgeleitete Erkenntnisse präsentiert. Im Unterkapitel „Leistungsanalyse“ befassen wir uns mit der Performance des Codes und eingesetzten Verbesserungsmaßnahmen.

3.1 Design

Bei der Vorbereitung des Projektes war die erste Frage, welche Online-Nachrichtenagenturen wir in unsere RSS-Feed-Liste aufnehmen, um an das Textmaterial heranzukommen. Dabei ist die Vergleichbarkeit der Artikel wichtig. Wir beschränkten uns deshalb auf Nachrichtenseiten in englischer Sprache und innerhalb dieser nur auf die internationalen Rubriken, da wir so eine hohe Konsistenz der Daten erzielten. Zu Anfang waren auch die englischsprachigen Rubriken deutscher Online-Nachrichten eingeplant, zum Beispiel von „Spiegel Online“. Die Idee wurde verworfen, da wir in dieser Kategorie kaum RSS-Feeds fanden und um darüber hinaus die Einheitlichkeit zu bewahren, fiel die Entscheidung auf Länder, in denen Englisch die Amtssprache ist.

Dem Crawler, welchen wir von einem Kommilitonen zur Verfügung gestellt bekamen, wurde unsere RSS-Feed-Liste eingespeist, sodass dieser die von uns ausgewählten Webseiten über das ganze Semester nach Artikeln durchsucht und sie als CSV-Datei speichert. Insgesamt wurden 179.890 Artikel in den Zeiträumen vom 10.11.2017 bis 09.01.2018 und 17.01.2018 bis 13.03.2018 gesammelt (die Pause im Januar ergab sich durch den Ausfall des Clusters). Die folgende Tabelle zeigt alle genutzten Nachrichten-Webseiten und die Anzahl der gecrawlten Artikel:

USA		Großbritannien		Australien	
ABC News	13.806	Daily Express	6.167	ABC Australia	4.237
BBC US	1.525	Daily Mail	30.541	Canberra Times	64
CBS	7.062	The Guardian	10.015	News Australia	2.535
CNN	1.420	The Independent	6.882	The Advertiser	4.393
Fox News	18.496			The Daily Telegraph AU	2.499
NPR	4.669			The Mercury	28.400
New York Times	4.698			The West Australian	4.909
Reuters	9.067				
Washington Post	18.505				
*(mehrere Rubriken)					
Total:	79.248 Artikel	Total:	53.605 Artikel	Total:	47.037 Artikel

Der HTML-Code soll mit Hilfe von BeautifulSoup mit dem reinen Text ersetzt werden. Für diese Aufgabe wurde eine Konvertierungs-Methode geschrieben, welcher die CSV-Dateien übergeben werden. Aufgrund der Größe der gecrawlten Dateien von mehreren Gigabyte, sollte die Extraktion des Reintextes über den Cluster laufen. Die konvertierten Dateien dagegen sind größentechnisch mit einigen Megabyte handhabbar. Um die CSV-Dateien in Python zu bearbeiten, organisieren wir sie in

Dataframes, einer Pandas Datenstruktur. Das soll einen möglichst einfachen Umgang mit den Tabellen ermöglichen.

Das Pre-Processing der Texte besteht unter anderem aus dem Entfernen aller Satzzeichen. Des Weiteren werden Füllwörter mit Hilfe vordefinierter Listen herausgefiltert. So wird sichergestellt, dass ausschließlich textuell relevante Wörter untersucht werden. Außerdem sollten Stemming und Lemmatizing ausprobiert werden, um deklinierte und konjugierte Wörter zu ihrem Ursprung zurückzuführen. Dies soll sicherstellen, dass bei der Analyse das Vorkommen von zum Beispiel „spricht“ und „sprachen“ nicht als zwei verschiedene Wörter gewertet wird. Dabei reduziert Stemming Wörter auf ihre Wurzel, also auf die kleinste mögliche Einheit des Wortes, welche meist alleinstehend keine Bedeutung mehr hat. Im vorangegangenen Beispiel wäre dies „sprech“. Lemmatizing dagegen gibt die Grundform der Wörter zurück, also „sprechen“. Unter anderem aus dieser Beobachtung heraus entschieden wir uns letztendlich gegen den Einsatz von Stemming.

Das Pre-Processing sollte beginnen, bevor ein aussagekräftiges Datenvolumen gecrawlt wurde. Es wurde eine beispielhafte, bereits konvertierte CSV-Datei genutzt, um den Code zu testen. Der Code musste später nur noch auf die größeren Artikelmengen angewandt werden.

Die Analyse und Visualisierung der Ergebnisse fand statt, nachdem eine ausreichende Menge an Daten gesammelt wurde. Die Recherche unterschiedlicher Bibliotheken in Bereichen wie Natural Language Processing, Text Mining und Machine Learning sollte die Basis für die Implementierung darstellen. Die Konzepte, die wir ausprobieren wollten, waren in erster Linie Bag of Words, TF-IDF, KMeans-Clustering und MeanShift-Clustering. Diese wurden letztendlich aber alle entweder komplett verworfen oder neu implementiert. Im Nachhinein war das größte Problem in der Planungsphase, dass wir noch keine genauen Vorstellungen davon hatten, welche Methoden zu unseren Zielen führen würden, und deshalb sehr viele Bibliotheken recherchiert und ausprobiert haben, die in Hinsicht auf unser Projekt und unseren Wissensstand die falsche Wahl waren. Ganz zu Anfang war als Hauptaugenmerk des Projektes die automatische Themenidentifikation geplant, also die Implementation von Methoden, die durch maschinelles Lernen das Thema eines Textes herausfiltern. Erst nach einiger Einarbeitung in die Materie erkannten wir, wie hochgestochen dieses Ziel war. Aufgrund der Tatsache, dass wir in diesem Projekt sowohl mit der Programmiersprache Python als auch mit Big Data Text-Analysen zum ersten Mal in Kontakt traten, wollten wir alle vermeintlich relevanten, komplexen Methoden schnell erlernen, damit uns später kein Wissen fehlte. So fiel unser Fokus zu früh auf zu fortgeschrittene Methoden.

Nachdem diese Konzepte nicht weiterführten und verworfen wurden, begann das kleinschrittige, strukturiertere Vorgehen, wobei zunächst Metadaten betrachtet werden sollten, die interessante Eigenschaften aufzeigen. Auf deren Ergebnissen aufbauend sollten, wenn möglich und sinnvoll, Vergleiche zwischen den Texten stattfinden. Eine Liste mit geplanten Analysen und dazugehörigen Grafiken wurde zu diesem Zweck erstellt:

Metadatenanalyse:

- Anzahl der Artikel pro Land und pro Paper (Bar-Charts, x-Achse: Länderkürzel bzw. Zeitungsname)
- Gesamt-Textlänge pro Land (Bar-Chart, x-Achse: Länderkürzel)
- Textlänge pro Artikel für je ein Land (drei Bar-Charts, y-Achse: Häufigkeit)
- Durchschnittliche Textlänge pro Land (Bar-Chart, außerdem als Linie in der jeweiligen Grafik „Textlänge pro Artikel“)
- Vergleich der Textlänge vor und nach dem Bereinigen pro Land und pro Paper (Bar-Charts mit je zwei unterschiedlichen Datensätzen, y-Achse: Häufigkeit)
- Anzahl von Unique Words pro Artikel für je ein Land (drei Bar-Charts, y-Achse: Häufigkeit)
- Durchschnittliche Anzahl von Unique Words (Bar-Chart, außerdem als Linie in der jeweiligen Grafik „Anzahl von Unique Words pro Artikel“)
- Durchschnittliche Wortlänge pro Artikel für je ein Land (drei Bar-Charts, y-Achse: Häufigkeit)
- Durchschnittliche Wortlänge pro Land (Bar-Chart, außerdem als Linie in der jeweiligen Grafik „Wortlänge pro Artikel“)

Ähnlichkeitsanalyse:

- Part of Speech im Durchschnitt pro Land und pro Paper (Tabellen mit häufigsten grammatikalischen Formen)
- Bag of Words
- TF-IDF
- Top-Wörter nach Häufigkeit (BOW) pro Land (Bar-Chart)
- Top-Wörter nach Relevanz (TF-IDF-Wert) pro Land (Bar-Chart)
- Distanz-Metriken und passende Visualisierungsmöglichkeiten recherchieren

Alle weiteren letztlich genutzten Methoden, wie Multidimensional Scaling und das Ward-Verfahren, ergaben sich während des Programmierens und Recherchierens als sinnvoll und waren vorher nicht geplant gewesen.

3.2 Implementierung

Im folgenden Kapitel werden die Ergebnisse der Auswertungen inklusive der Visualisierungen besprochen und Erkenntnisse abgeleitet. Wir gehen grob auf die Implementierung des Codes ein, genauere Beschreibungen dazu sind in den ZIP-Dateien im Code zu finden.

3.2.1 Crawler

Mit Hilfe von einem Crawler werden Artikel aus verschiedenen Online-Zeitungen heruntergeladen, um sie später analysieren zu können. An dieser Stelle wurde ein bereits implementierter Crawler von unserem Kommilitonen Max Lübbering benutzt (Quelle: https://github.com/le1nux/crawly/).

Dieser Crawler kann die von Anfang an festgelegten RSS-Feeds abrufen und überprüfen, ob neue RSS-Items dazu gekommen sind, die noch nicht bekannt sind. In diesem Fall werden dann aus den dazu gekommenen Items Artikel heruntergeladen, auf die diese RSS-Items referenzieren. Der Prozess geschieht in von dem User vordefinierten Zeitintervallen und mit vordefinierter Häufigkeit. In der folgenden Abbildung sind die zu analysierenden RSS-Feeds aufgelistet:

```
outlet,outlet_url,country,feed_url,schedule
"Daily Express","http://www.express.co.uk",UK,http://feeds.feedburner.com/daily-express-world-news,1
"Daily Mail","http://www.dailymail.co.uk",UK,http://www.dailymail.co.uk/news/index.rss,1
"The Guardian","https://www.theguardian.com/international",UK,https://www.theguardian.com/world/rss,1
"The Independent","http://www.independent.co.uk",UK,http://www.independent.co.uk/news/world/rss,1
"The Advertiser","http://www.theadvertiser.com/",AU,http://rssfeeds.theadvertiser.com/lafeyettela/home,1
"The Daily Telegraph AU","http://www.dailytelegraph.com.au/",AU,http://www.dailytelegraph.com.au/news/world/rss,1
"Canberra Times","http://www.canberratimes.com.au/",AU,http://www.canberratimes.com.au/rssheadlines/world/article/rss.xml,1
"The Mercury","http://www.themercury.com.au/",AU,http://www.themercury.com.au/rss,1
"ABC Australia","http://www.abc.net.au/",AU,http://www.abc.net.au/news/feed/52278/rss.xml,1
"The west australian","https://thewest.com.au/",AU,https://thewest.com.au/news/world/rss,1
"News Asutralia","http://www.news.com.au/",AU,http://www.news.com.au/content-feeds/latest-news-world/,1
"NPR","http://www.npr.org/",US,http://www.npr.org/rss/rss.php?id=1001,1
"Reuters US","http://www.reuters.com/",US,http://feeds.reuters.com/Reuters/domesticNews,1
"Reuters World","http://www.reuters.com/",US,http://feeds.reuters.com/Reuters/worldNews,1
"Reuters Politics","http://www.reuters.com/",US,http://feeds.reuters.com/Reuters/PoliticsNews,1
"BBC US","http://www.bbc.com/news",US,http://feeds.bbci.co.uk/news/world/us_and_canada/rss.xml?edition=int,1
"New York Times US","https://www.nytimes.com/",US,http://rss.nytimes.com/services/xml/rss/nyt/US.xml,1
"New York Times World","https://www.nytimes.com/",US,http://rss.nytimes.com/services/xml/rss/nyt/World.xml,1
"CNN US","http://edition.cnn.com/",US,http://rss.cnn.com/rss/edition_us.rss,1
"CNN World","http://edition.cnn.com/",US,http://rss.cnn.com/rss/edition_world.rss,1
"Washington Post US","https://www.washingtonpost.com/",US,http://feeds.washingtonpost.com/rss/national,1
"Washington Post World","https://www.washingtonpost.com/",US,http://feeds.washingtonpost.com/rss/world,1
"Washington Post Politics","https://www.washingtonpost.com/",US,http://feeds.washingtonpost.com/rss/politics,1
"FOX News US","http://www.foxnews.com/",US,http://feeds.foxnews.com/foxnews/national,1
"FOX News World","http://www.foxnews.com/",US,http://feeds.foxnews.com/foxnews/world,1
"FOX News Politics","http://www.foxnews.com/",US,http://feeds.foxnews.com/foxnews/politics,1
"CBS US","https://www.cbsnews.com/",US,https://www.cbsnews.com/latest/rss/us,1
"CBS World","https://www.cbsnews.com/",US,https://www.cbsnews.com/latest/rss/world,1
"CBS Politics","https://www.cbsnews.com/",US,https://www.cbsnews.com/latest/rss/politics,1
"ABC News US","http://abcnews.go.com/",US,http://abcnews.go.com/abcnews/usheadlines,1
"ABC News World","http://abcnews.go.com/",US,http://abcnews.go.com/abcnews/worldnewsheadlines,1
"ABC News International","http://abcnews.go.com/",US,http://abcnews.go.com/abcnews/internationalheadlines,1
"ABC News Politics","http://abcnews.go.com/",US,http://abcnews.go.com/abcnews/politicsheadlines,1
```

Abb.: feed_list.csv in der ZIP-Datei

In unserem Fall wurde eine Standardeinstellung benutzt, indem jeder Artikel sieben Mal im 10-minutigen Intervall heruntergeladen wurde. Damit können eventuelle Änderungen an Artikeln sichtbar und nachvollziehbar gemacht werden. Allerdings wurden diese Änderungen bei der Analyse nicht weiter berücksichtigt, da beim Preprocessing-Teil alle Duplikate rausgelöscht wurden.

Jeder Artikel wird als HTML-Seite in einer CSV-Datei gespeichert. Ein Beispiel einer solchen Datei ist in der folgenden Abbildung dargestellt:

Dabei ist der erste Eintrag in jeder Zeile ein Zeitstempel, der zweite eine URL zum RSS-Feed von einer Zeitung und der dritte eine URL zum Artikel aus dieser Zeitung. Auf der letzten Position wird ein HTML-Code aus der in URL angegebenen Webseite gespeichert. Dabei ist zu beachten, dass jeweils ein Artikel pro Zeile in einer CSV-Datei abgespeichert wird. Insgesamt gab es eine CSV-Datei pro Tag.

Der Crawler lief über den gesamten Projektzeitraum auf dem Cluster, wo auch die heruntergeladenen Artikel gespeichert wurden. Am Ende wurden Artikel vom 10.11.2017 bis 09.01.2018 und vom 17.01.2018 bis 13.03.2018 analysiert (dazwischen ist ein Netzwerkteil aus dem Cluster kaputt gegangen, so dass der Crawler nicht funktioniert hat).

Funktionsweise vom Crawler

Alle Anfragen von Crawler (RSS-Feeds und Artikel-Downloads) werden in requests.csv gespeichert. Beim Start werden zwei "Warmup"-Läufe gestartet (-> ~6min). Dort werden alle Items/Artikel heruntergeladen und als bekannt markiert. RSS-Feeds enthalten immer alle aktuellen Artikel (z.B. die letzten 100 veröffentlichten oder alle Artikel der letzten 48 Stunden). Der Crawler soll allerdings nur die neuen, gerade veröffentlichten Artikel herunterladen. Deswegen werden alle Artikel der RSS-Feeds zu Anfang als bekannt markiert. Nach den zwei Warmup-Läufen werden in regelmäßigen Intervallen die RSS-Feeds heruntergeladen und deren Items in crawled_articles.csv gespeichert. Bei jeder Iteration werden alle Items der Feeds gespeichert. Ein Item taucht daher mehrmals in der Datei auf. Wenn in einem Iterationsschritt ein komplett neuer Artikel in einem RSS-Feed gefunden wird, wird der Artikel (über Link im RSS-Item) sieben Mal in 10-minutigen Abständen heruntergeladen und als HTML-Seite in website_downloads.csv gespeichert.

3.2.2 Reintextextraktion mit BeautifulSoup

Da die „gecrawlten“ Artikel mit dem kompletten HTML-Code gespeichert werden, muss aus diesen zuerst der Artikel-Text extrahiert werden, um den Reintext später analysieren zu können. Dafür wurde ein Converter (s. Converter.py in der ZIP-Datei) implementiert. Dieser parst HTML-Code mit der BeautifulSoup-Bibliothek und extrahiert daraus Artikeltexte anhand der bestimmten HTML-Tags. Dabei mussten zuerst für jede Online-Zeitung diese Tags gesucht werden, unter denen Artikeltexte zu finden sind. In den meisten Fällen waren das <p>-Tags innerhalb eines -Tags mit bestimmten Attributen (z.B. „article_body“ oder „articleContent“).
Anschließend wurden die extrahierten Artikeltexte in einer CSV-Datei gespeichert. Diese ist identisch zu den ursprünglichen CSV-Dateien mit den „gecrawlten“ Artikeln aufgebaut, wobei der HTML-Code mit dem jeweiligen Reintext ersetzt wurde.

Die gecrawlten CSV-Dateien sind folgendermaßen aufgebaut:

1. Zeile: "Timetag","RSS","URL","html-Seite 1"
2. Zeile: "Timetag","RSS","URL","html-Seite 2"
usw.

Die "extrahierten" CSV-Dateien sind folgendermaßen aufgebaut:

1. Zeile: "Timetag","RSS","URL","Artikeltext 1"
2. Zeile: "Timetag","RSS","URL","Artikeltext 2"
usw.

Die Implementation von dem Converter war sehr zeitaufwendig, da viele unterschiedliche Zeitungen zur Analyse genommen wurden, für die jeweils eine passende Kombination aus HTML-Tags und -Attributen gefunden werden musste, um die richtigen Parts aus einer Webseite extrahieren zu können. Darüber hinaus sind hier Performance-Probleme aufgetreten, so dass eine Extraktion pro CSV-Datei bis zu 5 Minuten Zeit in Anspruch genommen hat.
Es ist auch zu erwähnen, dass der gesamte Extraktionsprozess aufgrund der Speicherplatzmängel auf dem Cluster durchgeführt wurde.
Größenvergleich:

CSV-Datei mit gecrawlten Artikeln:

Durchschnittliche Größe – 1,7 GB

Gesamt – ca. 210 GB

CSV-Datei mit extrahierten Artikeln:

Durchschnittliche Größe – 20 MB

Gesamt – ca. 2,55 GB

3.2.3 Preprocessing der Texte

In dieser Phase haben wir erstmal versucht, durch Rumprobieren und viel experimentelles Coden Verständnis für Listen, Dataframes, Strings und ihre Methoden zu entwickeln, nebenbei begann das Preprocessing.

Beim Preprocessing sollen Artikel für die eigentliche Textanalyse vorbereitet werden, indem alle Stopwörter (z.B. „the“ oder „and“), die keine semantische Bedeutung besitzen und bei der Textanalyse zur Fälschung der Ergebnisse führen, und Satzzeichen rausgelöscht werden. Außerdem sollen unterschiedliche Wortformen zu einer Grundform umgewandelt werden (z.B. „splitting“ oder „splitted“ werden in „split“ umgewandelt), um das Vocabulary bei der Textanalyse möglichst klein zu halten und die Wortvielfalt besser einschätzen zu können.

Die folgende Abbildung zeigt den ersten funktionierenden Versuch (s. Versuchscode.py):

```
##### Delete Stop Words & Punctuation #####
from nltk.corpus import stopwords
from nltk.tokenize import word_tokenize
from nltk.stem import PorterStemmer
import string

stop_words = set(stopwords.words("english"))
# raw_text ist jetzt eine Liste aus strings, jeder string ein Artikel:
raw_text = list((data_frame["Article"].values.astype("U")))
# raw_text_liste ist eine Liste aus Listen mit string-Elementen,jede innere Liste ist ein Artikel
raw_text_liste = [[art] for art in raw_text]

k=0
ps = PorterStemmer()
for i in raw_text_liste:
    raw_str = "".join(i)
    raw_words = word_tokenize(raw_str) # split to words
    # Stop words and punctuation
    nonpunc = [char for char in raw_words if char not in string.punctuation]
    nonpunc = " ".join(nonpunc)
    raw_words = word_tokenize(nonpunc)
    filtered_words = [w for w in raw_words if w.lower() not in stop_words]
    #Stemming
    x=0
    for w in filtered_words:
        filtered_words[x] = ps.stem(w)
        x = x+1

    raw_text_liste[k] = filtered_words
    k = k+1
    filtered_text = raw_text_liste

z=0
for l in filtered_text:
    filtered_text[z] = " ".join(l)
    z=z+1
filtered_text  # Liste aus clean strings, jeder string ein clean Artikel
```

Für Stopwörter wird ein stopwords-Package von nltk und für Satzzeichen wird string.punctuation benutzt. Außerdem wird hier ein PorterStemmer für Wortformenumwandlung angewendet, um

möglichst wenige unterschiedliche Wortformen zu haben. Wie es bereits in dem Abschnitt 3.1 erklärt wurde, ist ein PorterStemmer aus unserer Sicht jedoch ineffizient, da nicht alle Wortformen erkannt und umgewandelt werden. Darüber hinaus werden Wörter zu einem Stamm und nicht zu einer Grundform zurückgeführt, so dass die nachträgliche semantische Wortableitung manchmal sehr schwer fällt (z. B. wird das Wort „party“ in „parti“ umgewandelt). Außerdem hat der oben gezeigte Code nicht alle Satzzeichen rausgelöscht (die Unicode-Sonderzeichen wurden nicht als Satzzeichen erkannt):

```
["johannesburg - latest new leader south africa 's rule parti time local 12:50 a.m new leader south africa 's rule parti coun
tri 's like next presid say african nation congress must halt corrupt `` within rank '' cyril ramaphosa make first address an
c leader parti face weakest point sinc took power end apartheid system white minor rule 1994 parti face grow public frustrat
multipl alleg corrupt presid jacob zuma ramaphosa say parti member must avoid motiv person gain ___ 12:30 a.m new leader sout
h africa 's rule parti countri 's like next presid make first speech tri reunit african nation congress weakest point sinc ta
ke power end apartheid cyril ramaphosa nelson mandela 's prefer successor proclaim `` victori doomsay '' observ said parti co
uld split amid grow frustrat scandal-pron presid jacob zuma ramaphosa say thousand rule parti deleg seen africa 's oldest lib
er movement `` worst ... divid '' deputi presid face challeng reviv economi countri 's trust anc ahead 2019 elect say `` aliv
lead stay ''",
 "unit nation - presid donald trump threaten wednesday cut u.s. fund countri support resolut critic decis recogn jerusalem is
rael 's capit `` 'll save lot n't care '' said allud u.s. aid presid strongli support u. ambassador nikki haley said tuesday
unit state `` take name '' countri vote favor gener assembl resolut thursday declar jerusalem 's statu chang direct israeli-p
alestinian negoti `` nation take money vote us take hundr million dollar even billion dollar vote us '' trump told report cab
inet meet washington haley sit nearbi `` 're watch vote let vote us '' letter 180 193 u.n. member state even tougher tweet tu
esday haley hint possibl u.s. retali trump 's comment made clear recipi u.s. assist billion dollar could stake haley 's threa
t drew sharp critic palestinian turkish foreign minist flew new york gener assembl vote accus u.s. intimid nihad awad nation
execut director council american-islam relat tweet trump 's comment `` govern use leadership UN bully/blackmail nation stand
```

Bei dem zweiten Versuch wurde der PorterStemmer durch WordNetLemmatizer ersetzt, der verschiedene Wortformen nicht als Wortstamm, sondern in der Grundform darstellt. Außerdem werden alle Unicode-Sonderzeichen extra rausgelöscht:

```
####### Preprocessing - 1 Version ######
##### Delete Stop Words & Punctuation #####

def longPreprocessing(big_dataframe):
    lemmatizer = WordNetLemmatizer()
    stop_words = set(stopwords.words("english"))

    # Iteriert über alle Artikeln und übergibt Zeilenindex von einem Artikel (art_index) und Artikel selbst (art)
    for article_index,article in big_dataframe["Article"].head(n=100).astype("U").iteritems():

        # Checkt jedes Zeichen, ob es ein Satzzeichen ist
        nopunc = [char for char in article if char not in string.punctuation]

        # Alle Zeichen wieder zurückjoinen
        nopunc = "".join(nopunc)

        # Löscht alle Stopwörter
        filtered_words = [word for word in nopunc.split() if word.lower() not in stop_words] # filtered_words ist eine Liste aus

        x=0
        for word in filtered_words:
            filtered_words[x] = lemmatizer.lemmatize(word)
            x = x+1

        filtered_text = " ".join(filtered_words)

        # Löscht alle Unicode-Zeichen
        tbl = dict.fromkeys(i for i in range(sys.maxunicode)
                            if unicodedata.category(chr(i)).startswith("P"))

        big_dataframe.set_value(article_index, "Article", filtered_text.translate(tbl))
    return big_dataframe
```

Jedoch konnten nicht alle Wortformen mit Hilfe von dem Lemmatizer (Standardeinstellung) erkannt und umgewandelt werden, da die Part of Speech für jedes Wort falsch bzw. mit einem anderen, für den Lemmatizer unbekannten POS-Tag markiert wurden. Deswegen haben wir Part of Speech und

Lemmatizing selbst implementiert. Hier ist eine Endversion von unserem Preprocessing (s. Preprocessing.py)

```
### Preprocessing - WORKING VERSION ###
from IPython.core.debugger import set_trace

def shortPreprocessing(b_dataframe):

    stop_words = set(stopwords.words("english"))

    # Iteriert über alle Artikeln und übergibt Zeilenindex von einem Artikel (art_index) und Artikel selbst (art)
    for article_index,article in b_dataframe["Article"].astype("U").iteritems():

        # Löscht alle Unicode-Zeichen
        decodedString = (article.encode("ascii", "ignore")).decode("utf-8")

        #set_trace()

        # Checkt jedes Zeichen, ob es ein Satzzeichen ist
        nopunc = [char for char in decodedString if char not in string.punctuation]

        # Alle Zeichen wieder zurückjoinen
        nopunc = "".join(nopunc)

        filtered_words = [lem(word) for word in nopunc.split() if word.lower() not in stop_words]

        filtered_text = " ".join(filtered_words)

        b_dataframe.set_value(article_index, "Article", filtered_text)
    return b_dataframe

### Part of Speech implementation ###
def get_pos( word ):
    w_synsets = wordnet.synsets(word)

    pos_counts = Counter()
    pos_counts["n"] = len(  [ item for item in w_synsets if item.pos()=="n"]  )
    pos_counts["v"] = len(  [ item for item in w_synsets if item.pos()=="v"]  )
    pos_counts["a"] = len(  [ item for item in w_synsets if item.pos()=="a"]  )
    pos_counts["r"] = len(  [ item for item in w_synsets if item.pos()=="r"]  )

    most_common_pos_list = pos_counts.most_common(3)
    return most_common_pos_list[0][0]

### Lemmatizer implementation ###
cache = {}
wnl = WordNetLemmatizer()

def lem(w):
    if w in cache:
        return cache[w]
    else:
        cache[w] = wnl.lemmatize(w, get_pos(w))
        return wnl.lemmatize(w, get_pos(w))
```

Zwischendurch traten auch Performance-Probleme auf, da eine große Datenmenge umgewandelt werden sollte. Eine detaillierte Beschreibung ist im Abschnitt 3.3 zu finden.

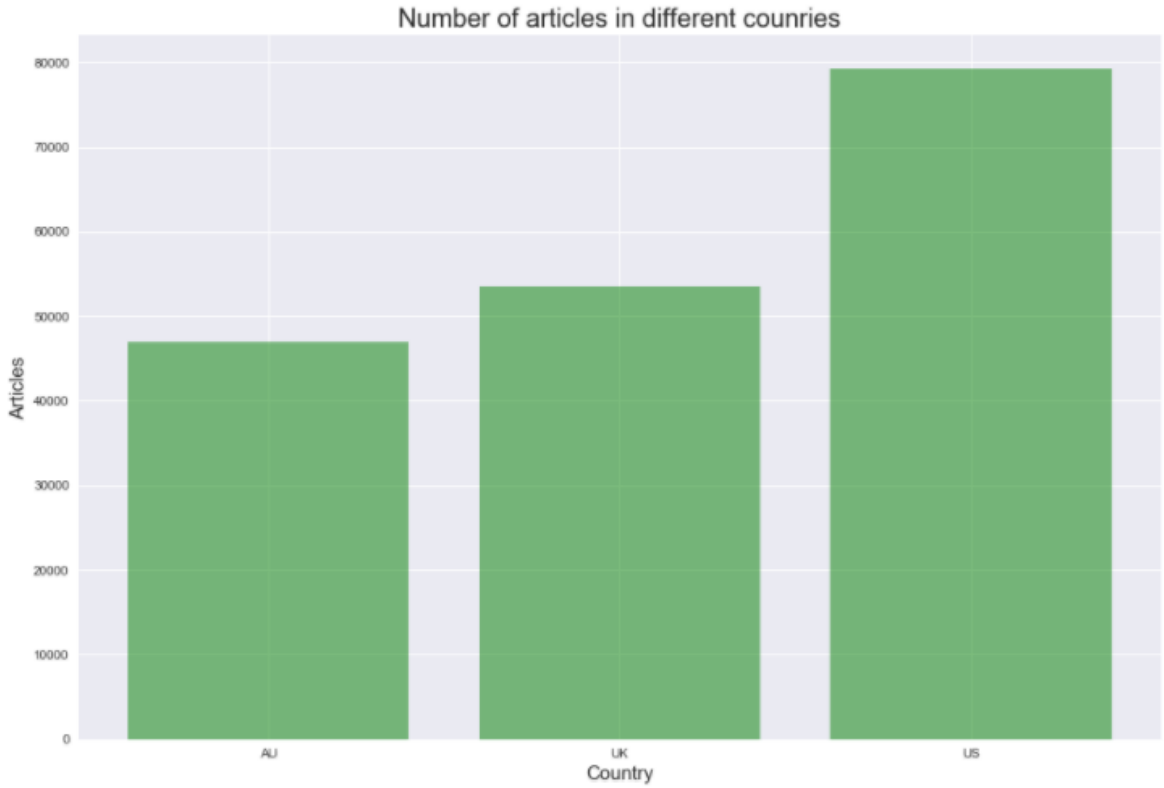
3.2.4 Visualisierungen

Bei der Visualisierung von entstandenen Ergebnissen wurden Diagramme von matplotlib.pyplot und das Seaborn-Design benutzt. Es ist auch zu erwähnen, dass die Koordinatensysteme (Max-Values) und „Bins“-Parameter bei der Erstellung von Diagrammen jedes Mal manuell angepasst werden mussten, da die Anzahl von Artikeln in verschiedenen Ländern/Zeitungen sehr unterschiedlich war und sich verschieden auf die Lesbarkeit der Grafik ausgewirkt hat.

3.2.5 Metadatenanalyse

Die Analyse umfasst im ersten Teil eine Auswertung und Visualisierung der Metadaten. Die Implementierung erfolgte hauptsächlich mit fundamentalen Methoden von Datentypen wie Integer und String, außerdem wurde häufig List Comprehension angewandt. Beispiele sind count, len, sum, split, append und set. Die Visualisierungen entstanden mit den Funktionen bar, hist und plot der Bibliothek matplotlib.pyplot. Insgesamt gab es bei der Implementierung in diesem Abschnitt keine nennenswerten Probleme.

Anzahl der Artikel pro Land



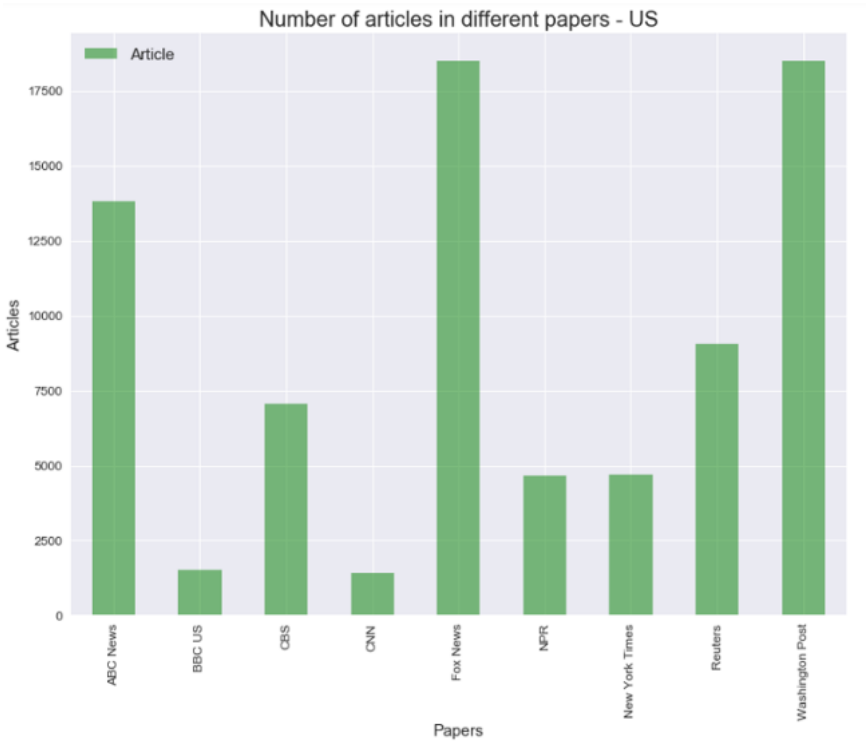
Die erste Grafik zeigt, dass am meisten US-amerikanische Artikel gecrawlt wurden (79.248 Artikel). Damit haben sie 25.643 Artikel bzw. 47,8% mehr als Großbritannien und 32.211 Artikel bzw. 68,5% mehr als Australien. Der Unterschied zwischen Großbritannien (53.605 Artikel) und Australien (47.037 Artikel) fällt mit 6.568 Artikeln oder 14% wesentlich kleiner aus. Die gesamten 179.890 Artikel verteilen sich folgendermaßen: 44% US, 29,8% UK und 26,2% AU.

Dies ist damit zu erklären, dass auch in der RSS-Feed-Liste die USA die meisten Paper hatte. Hinzu kommt, dass unter diesen Zeitungen fast immer mehrere relevante Rubriken zur Verfügung standen, die wir auch alle genutzt haben (zum Beispiel die Rubriken „World“ und „International“). Der große Abstand der USA muss bei der Auswertung in einigen der folgenden Analysen berücksichtigt werden. Bei der Implementierung nutzten wir hier die count-Funktion für Listen, um alle Artikel zu zählen.

Anzahl der Artikel pro Paper

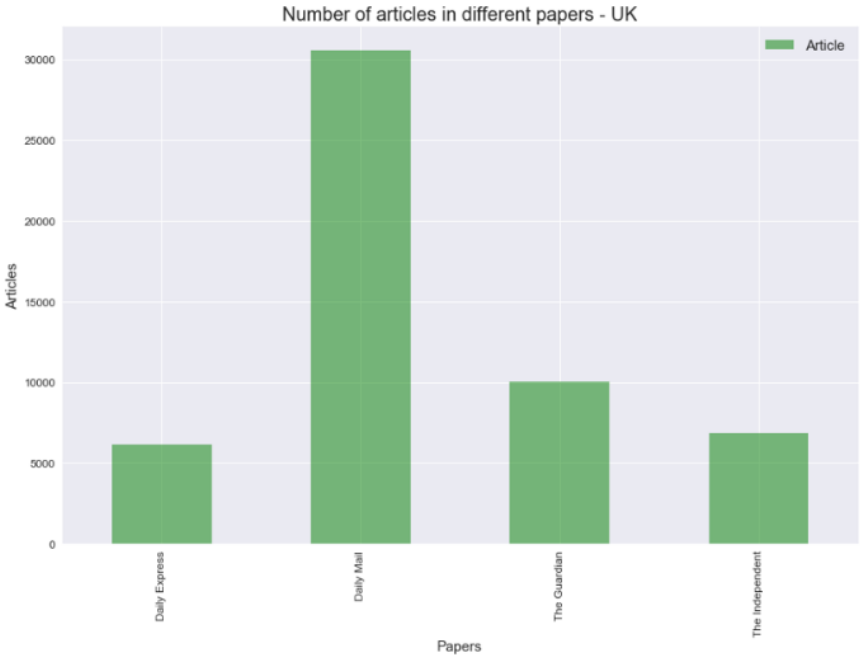
USA:

Paper	Article
ABC News	13806
BBC US	1525
CBS	7062
CNN	1420
Fox News	18496
NPR	4669
New York Times	4698
Reuters	9067
Washington Post	18505



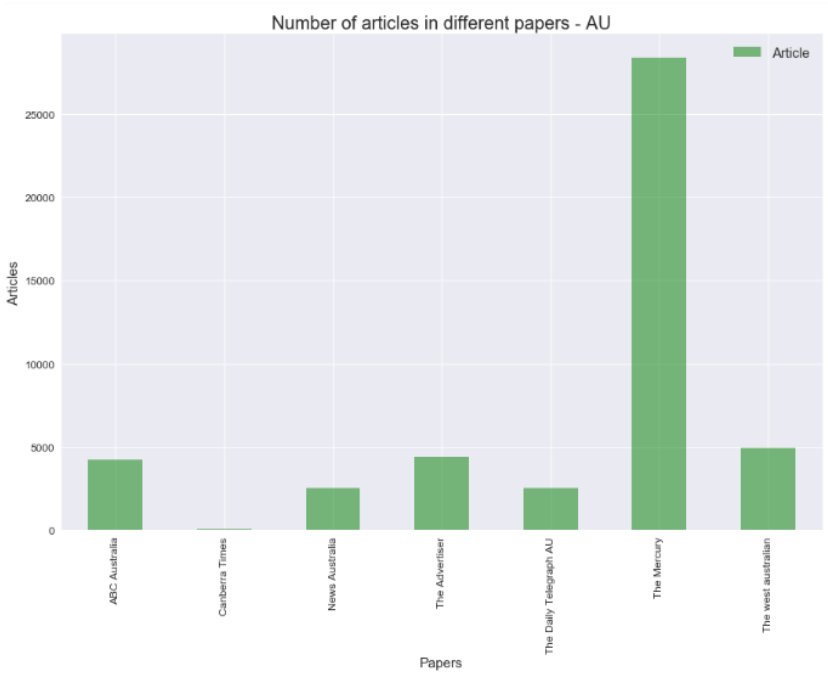
Großbritannien:

Paper	Article
Daily Express	6167
Daily Mail	30541
The Guardian	10015
The Independent	6882



Australien:

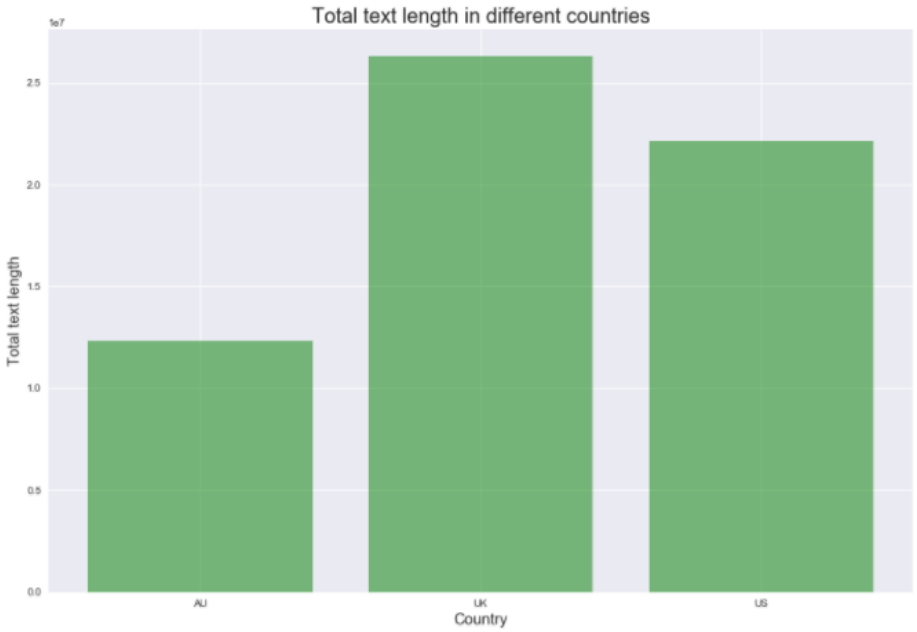
Paper	Article
ABC Australia	4237
Canberra Times	64
News Australia	2535
The Advertiser	4393
The Daily Telegraph AU	2499
The Mercury	28400
The west australian	4909



Hier wird die Verteilung der gecrawlten Artikel innerhalb der Paper verdeutlicht. In den USA scheinen ABC News, FOX News und die Washington Post vermehrt über internationale Themen zu schreiben. In Großbritannien ist es die Daily Mail und in Australien The Mercury, die solche Ausreißer darstellen. BBC US ist nur eine amerikanische Außenstelle der ansonsten britischen Newsseite, wodurch eventuell ihr geringerer Output zu erklären ist. Nicht deutlich wird, wieso CNN, die hinsichtlich ihrer Relevanz in der US-Nachrichten-Landschaft mit den anderen Webseiten auf einer Stufe steht, so wenige Artikel produziert hat. Das Schlusslicht stellt die australische Canberra Times dar, die über alle Länder hinweg am wenigsten Output produziert hat. Da das Paper nur eine lokale Zeitung ist, ist es logisch, dass sie seltener über internationale Themen berichten, zudem produzierte sie eventuell viele Video-Nachrichten, aus denen kein Reintext extrahiert werden konnte.

Gesamt-Textlänge pro Land

in Millionen

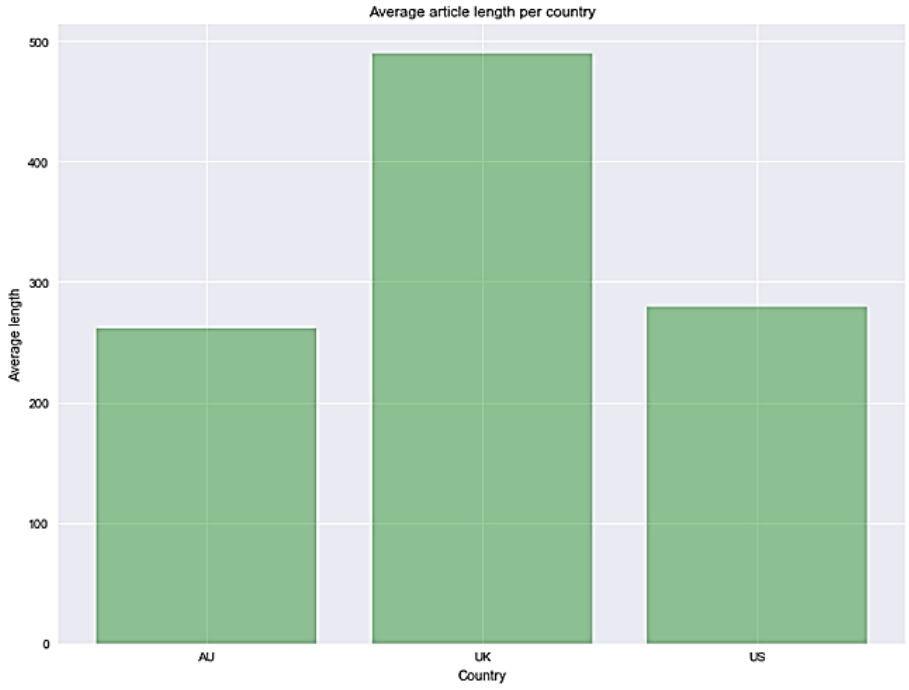


AU: 12.327.069

UK: 26.297.698

US: 22.143.257

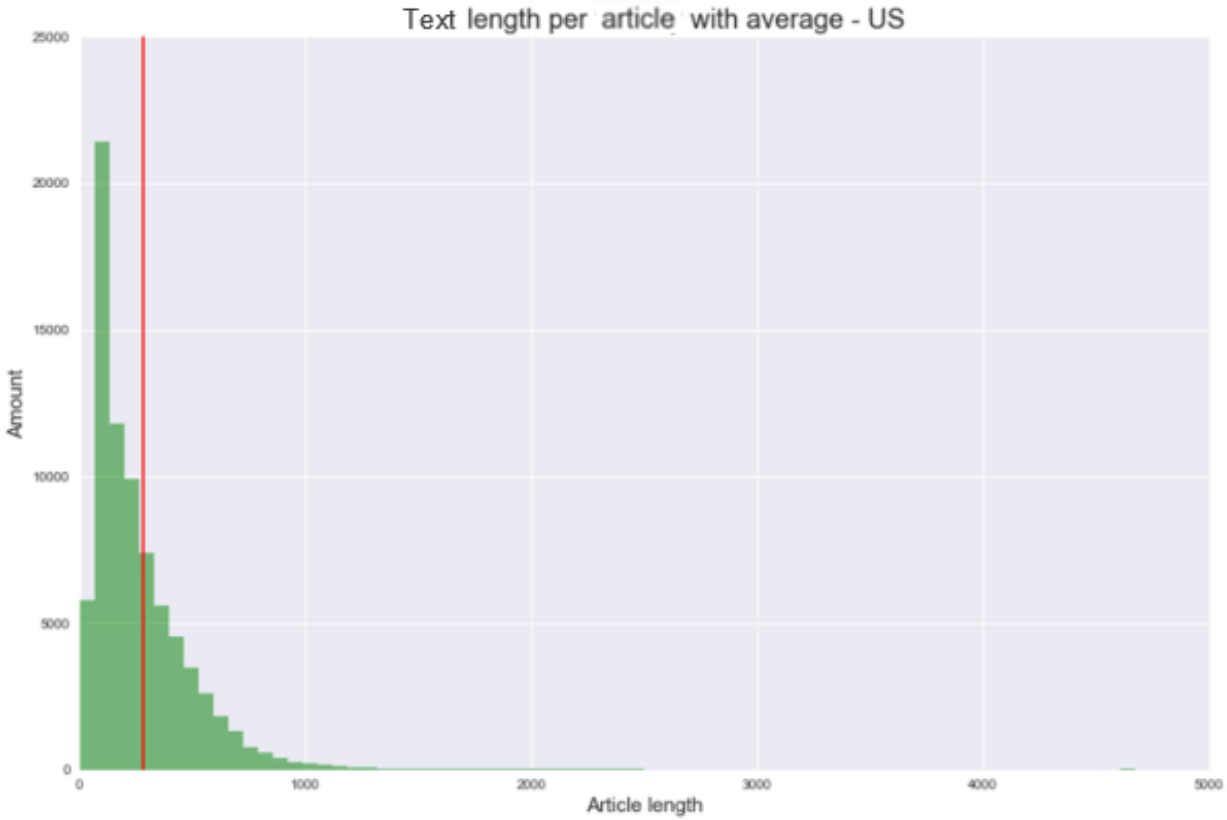
Durchschnittliche Textlänge pro Land



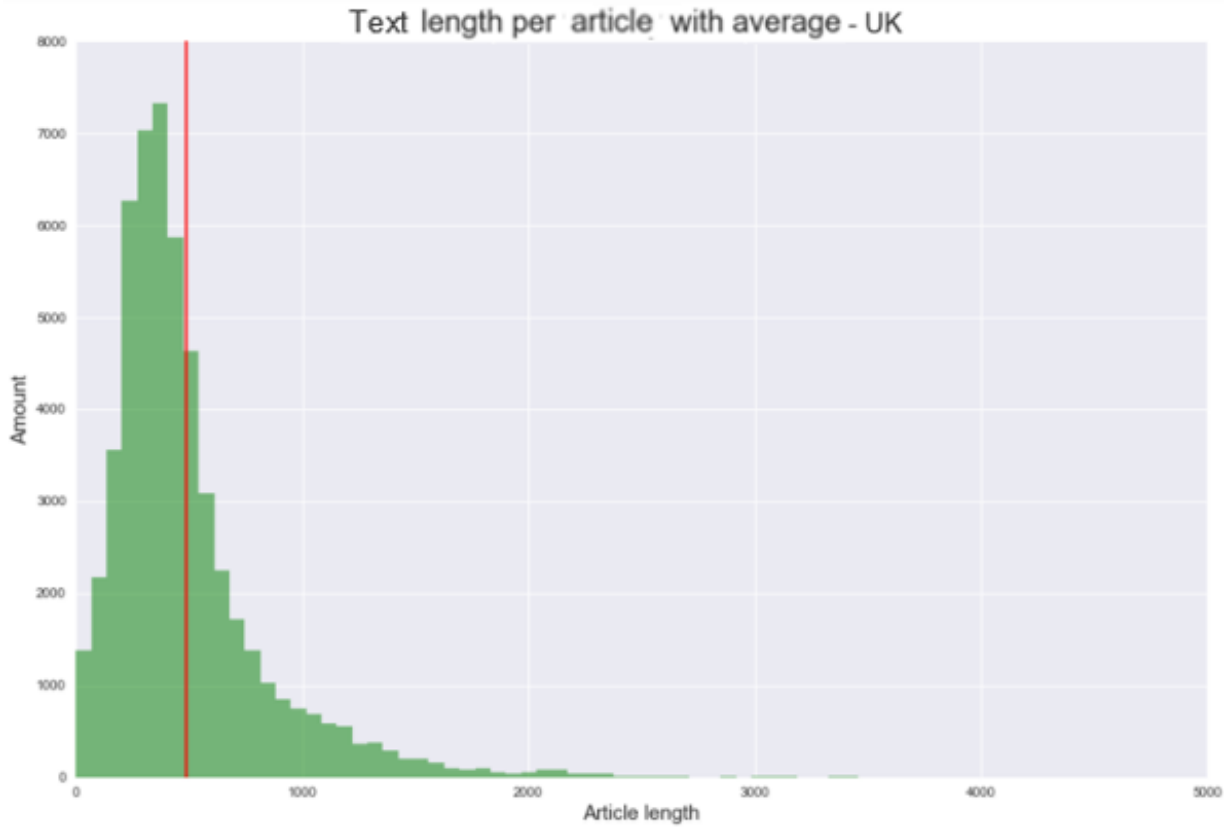
Die gesamte und die durchschnittliche Textlänge pro Land zeigen, dass Großbritannien die längsten Texte produziert. Vor allem bezüglich der Gesamtlänge ist dies erstaunlich, da die USA insgesamt viel mehr Artikel hat. Außerdem fällt auf, dass die USA eine fast doppelt so hohe Gesamtlänge aufweist als Australien, sich in der Durchschnittslänge aber nur gering von ihnen unterscheiden. Die nächsten Grafiken könnten die Gründe dafür aufzeigen.

Textlänge pro Artikel mit Landesdurchschnitt

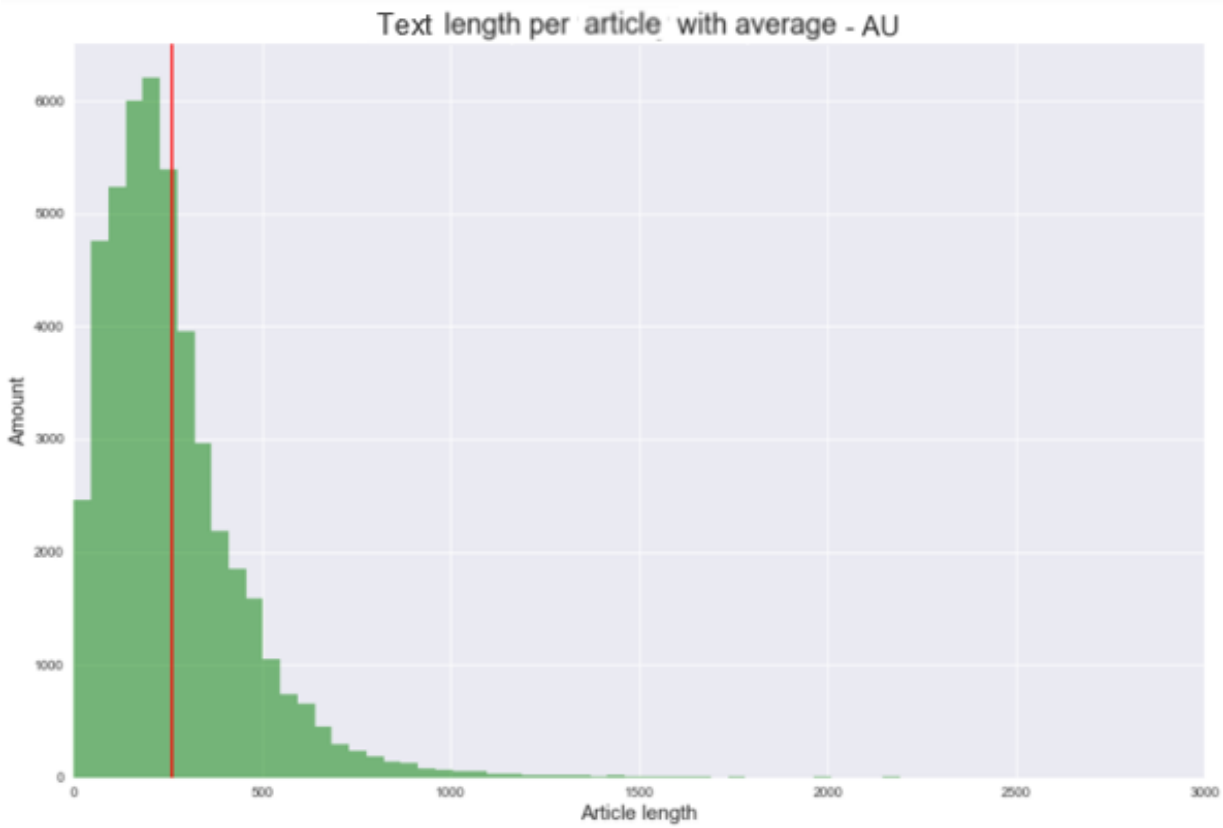
USA:



Großbritannien:



Australien:



In diesen Grafiken müssen die unterschiedlichen Dimensionen der Achsen berücksichtigt werden.

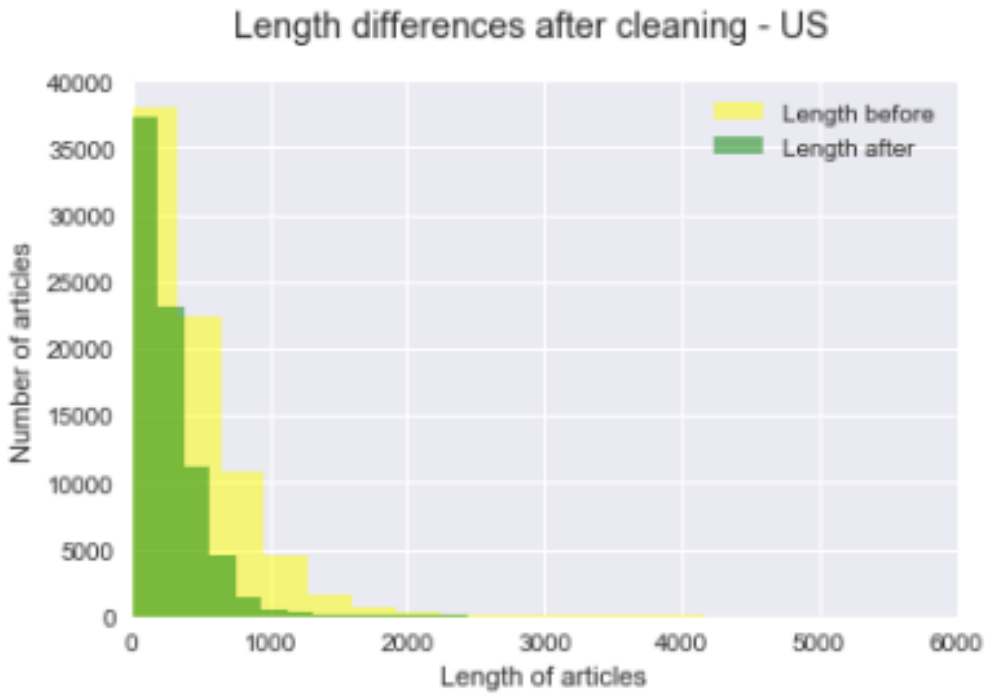
In den USA und Australien ist der deutliche Großteil der Artikel unter 500 Wörter lang (in Großbritannien ist dies zwar auch der Fall, aber in einem weit weniger extremen Verhältnis). Die USA hat zudem ca. 22.000 Artikel, die um die 100 – 200 Wörter lang sind. Dies erklärt, wieso ihr Durchschnitt ähnlich ausfällt. Die USA hat außerdem extremere Ausreißer mit über 4.500 Wörtern Länge, Australien dagegen nur bis fast 2.500 Wörtern. Diese Tatsache gepaart mit ihrer extrem hohen Anzahl an kürzeren Artikeln erklären wiederum ihre höhere Gesamtlänge im Vergleich mit Australien. Die hohe Gesamt- und Durchschnittslänge in Großbritannien ist ebenfalls sichtbar. Großbritannien hat, verglichen mit den anderen Ländern, die meisten Artikel mit über 500 Wörtern Länge und auch deutlich die meisten mit über 1000 Wörtern.

Vergleich der Textlänge vor und nach dem Bereinigen pro Land und pro Paper

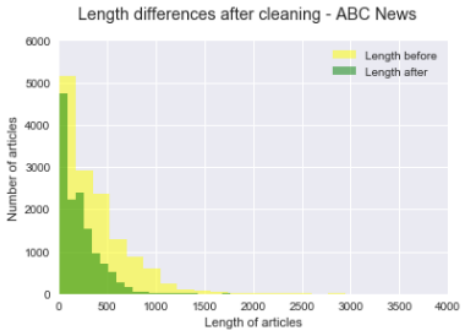
Hier wird visualisiert, in welchem Verhältnis sich die Textlänge der Artikel nach dem Bereinigen geändert hat. Es wird sichtbar, dass vermehrt kürzere Artikel vorhanden sind als vor dem Bereinigen (dies trifft für jedes Land zu).

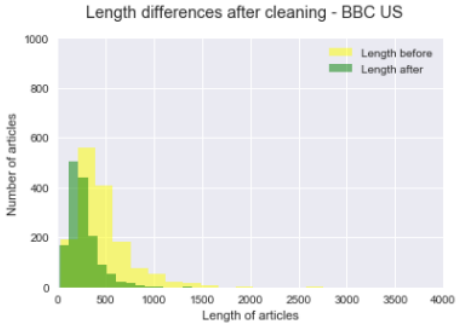
Die folgenden Grafiken zeigen alle, dass dasselbe auch für jedes Paper gilt. Diese Ergebnisse sind erwartungsgemäß ausgefallen, da der Code zum Bereinigen Füllwörter und Satzzeichen rauslöscht und die Texte so verkürzt.

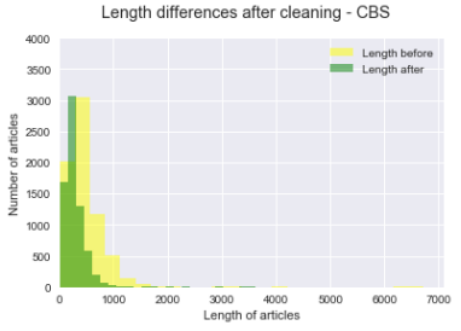
USA:

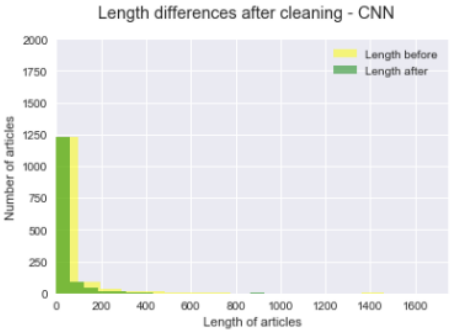


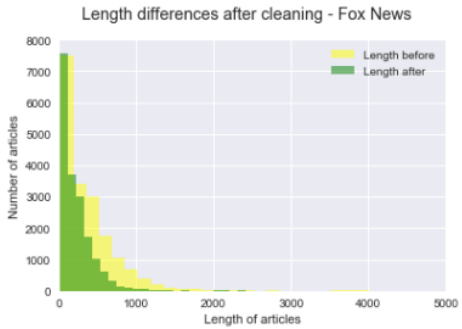
USA :

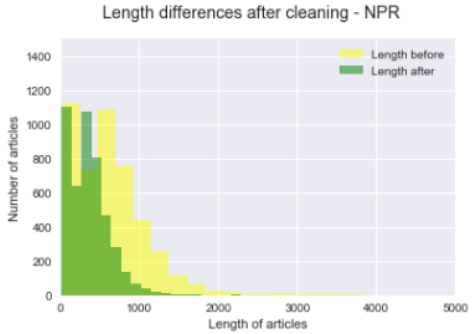

Length differences after cleaning - ABC News
Length before
Length after
Number of articles
Length of articles

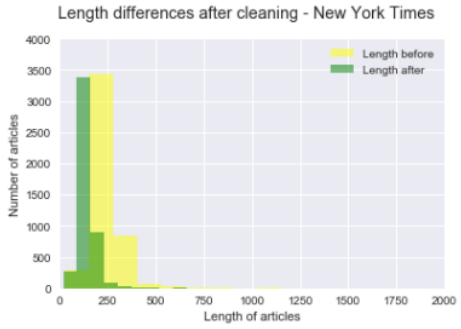

Length differences after cleaning - BBC US
Length before
Length after
Number of articles
Length of articles

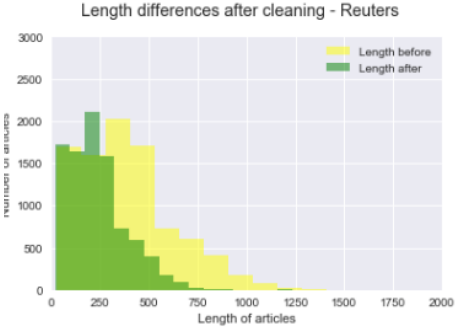

Length differences after cleaning - CBS
Length before
Length after
Number of articles
Length of articles

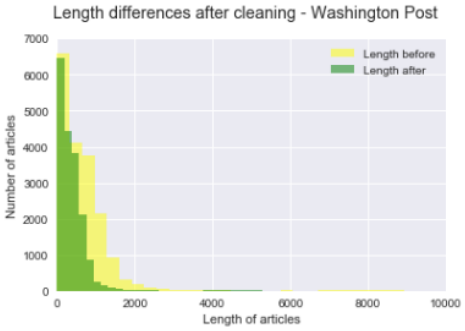

Length differences after cleaning - CNN
Length before
Length after
Number of articles
Length of articles


Length differences after cleaning - Fox News
Length before
Length after
Number of articles
Length of articles

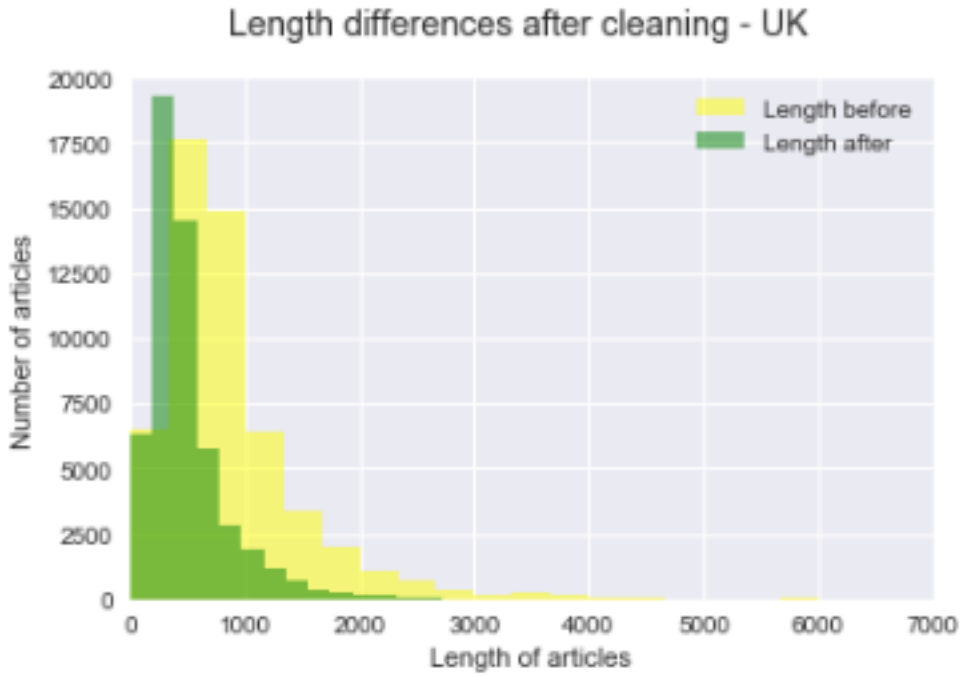

Length differences after cleaning - NPR
Length before
Length after
Number of articles
Length of articles

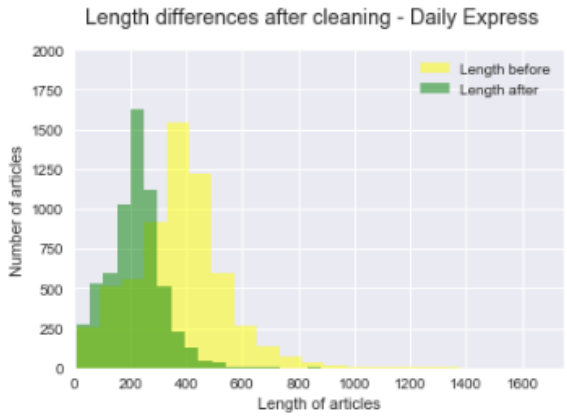

Length differences after cleaning - New York Times
Length before
Length after
Number of articles
Length of articles

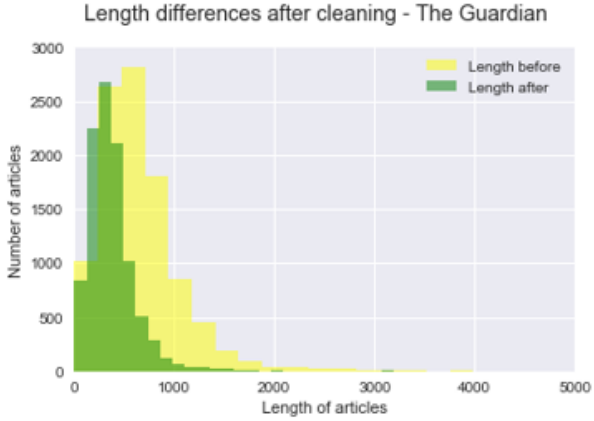

Length differences after cleaning - Reuters
Length before
Length after
Length of articles

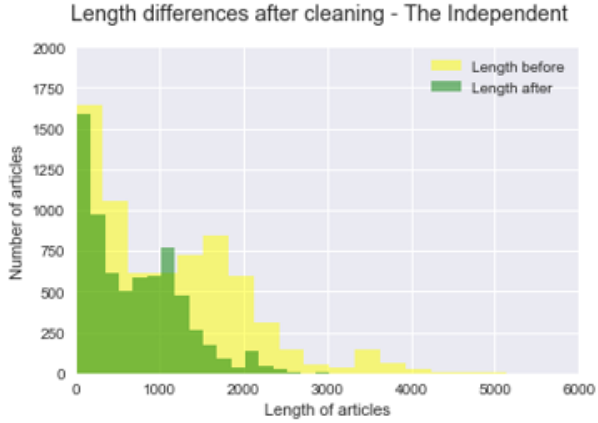

Length differences after cleaning - Washington Post
Length before
Length after
Number of articles
Length of articles

Großbritannien:

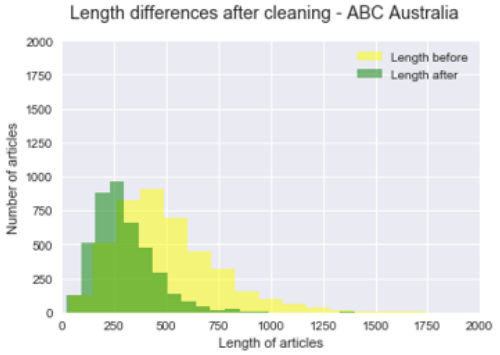


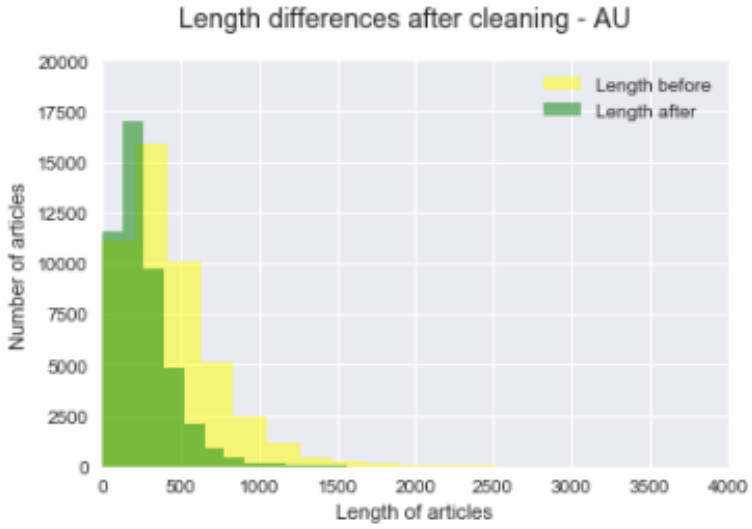


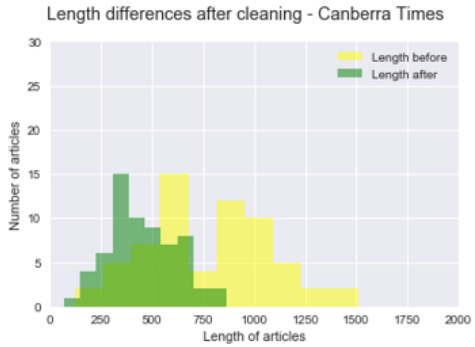


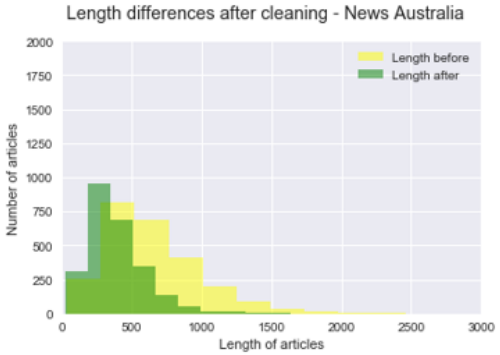


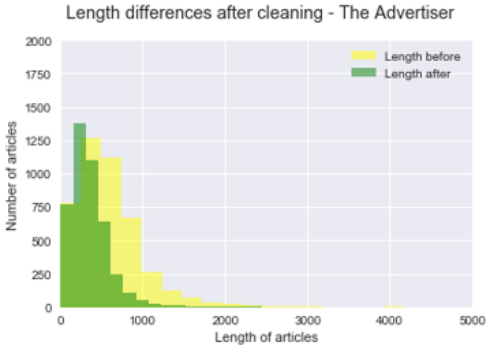
Australien:

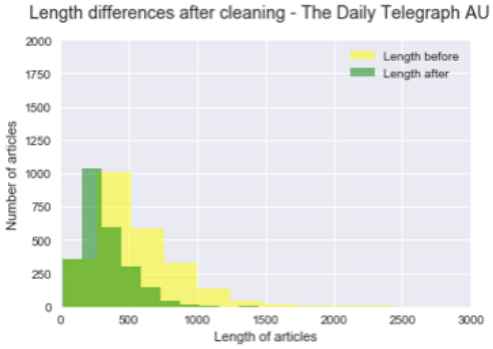

Length differences after cleaning - ABC Australia
Length before
Length after
Number of articles
Length of articles

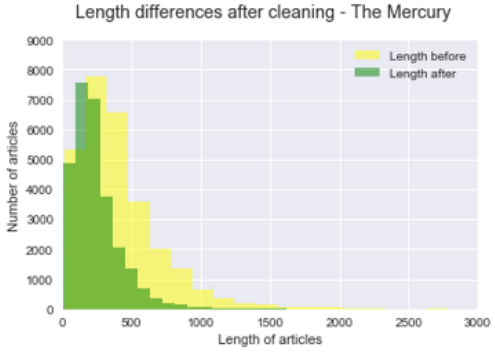

Length differences after cleaning - AU
Length before
Length after
Number of articles
Length of articles

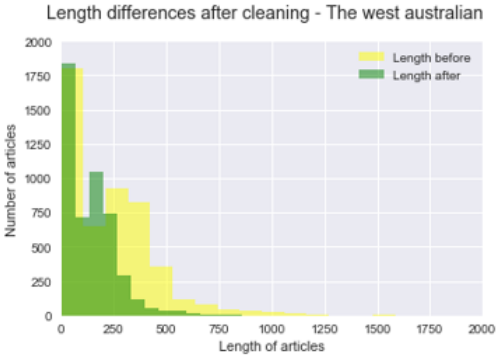

Length differences after cleaning - Canberra Times
Length before
Length after
Number of articles
Length of articles


Length differences after cleaning - News Australia
Length before
Length after
Number of articles
Length of articles

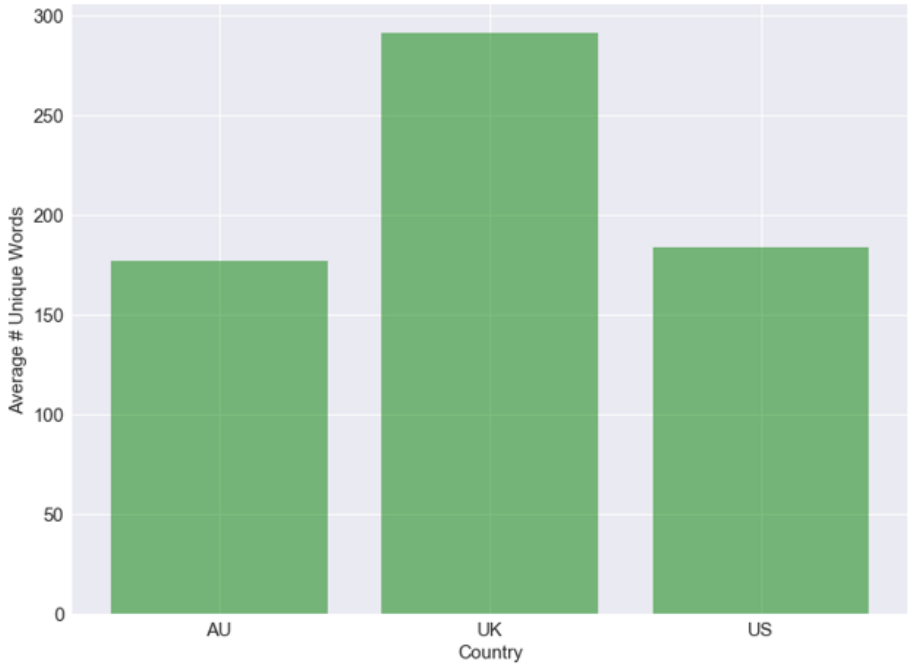

Length differences after cleaning - The Advertiser
Length before
Length after
Number of articles
Length of articles


Length differences after cleaning - The Daily Telegraph AU
Length before
Length after
Number of articles
Length of articles


Length differences after cleaning - The Mercury
Length before
Length after
Number of articles
Length of articles


Length differences after cleaning - The west australian
Length before
Length after
Number of articles
Length of articles

Durchschnittliche Anzahl von Unique Words

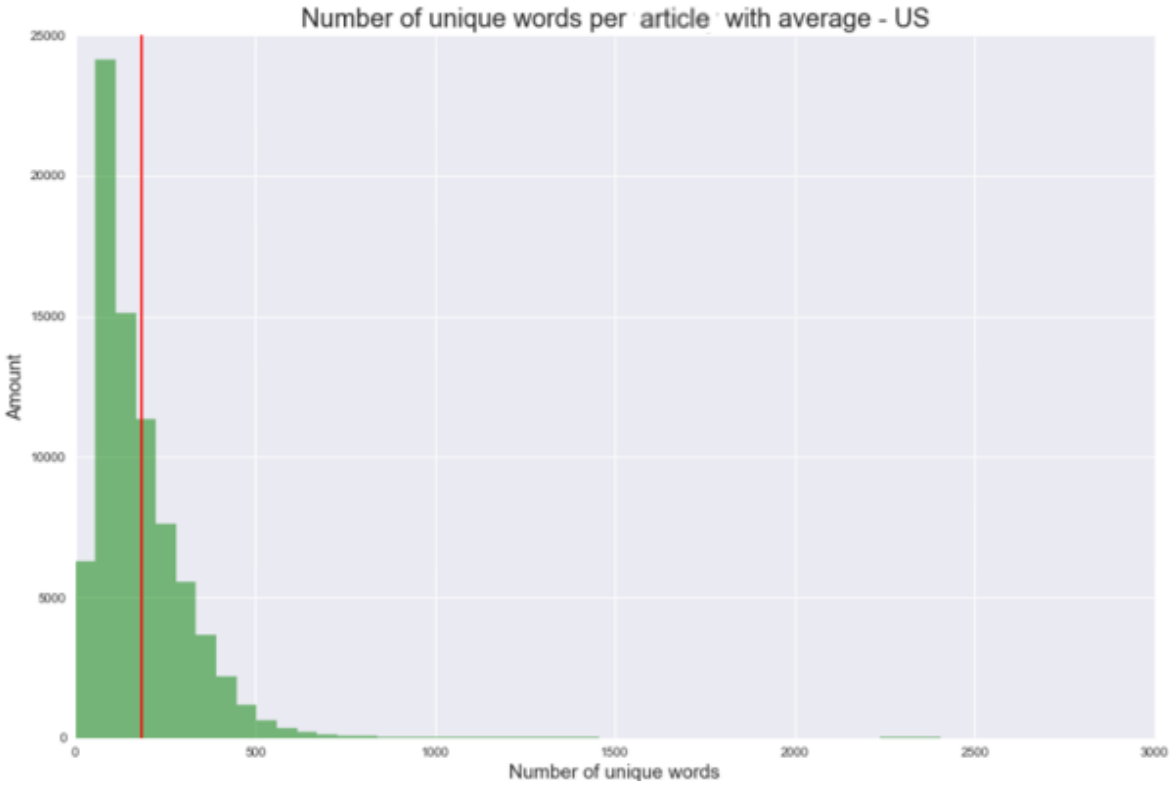


Unique Words beschreiben nicht Wörter, die im Text nur einmalig vorkommen, sondern zählen, wie viele verschiedene Wörter es gibt. Bei der Implementierung wurden die Wörter in ein set gepackt und dann die Länge des sets ausgegeben. Da Mengen in sets keine Duplikate haben, kam so die Anzahl der unterschiedlichen Wörter heraus.

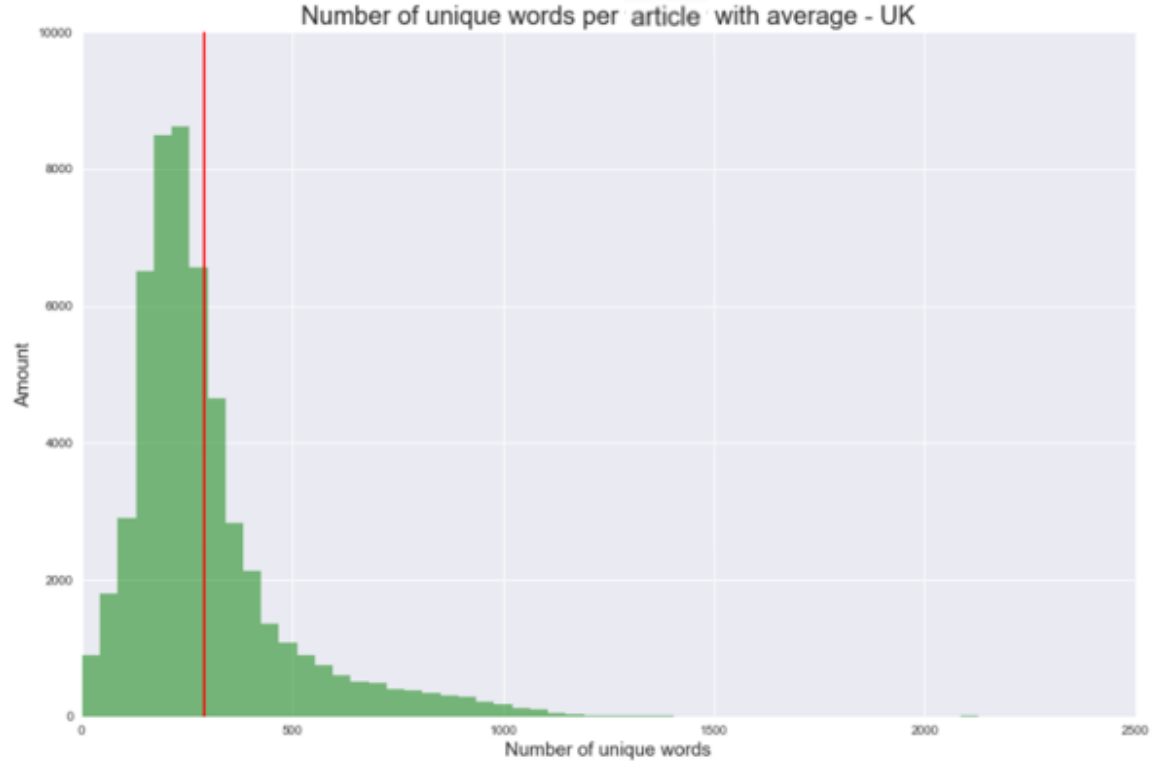
Großbritannien benutzt deutlich mehr Unique Words, was logisch erscheint, da ihre Texte durchschnittlich länger sind. Interessant wäre es, Texte der gleichen Länge zwischen den Ländern auf ihre Unique Words zu untersuchen, um herauszufinden, ob im britischen Dialekt das Vokabular tatsächlich vielfältiger gestaltet ist, oder ob der Ausreißer nur an den längeren Texten liegt. Dies konnten wir aus zeitlichen Gründen im Rahmen dieses Projektes nicht realisieren.

Anzahl von Unique Words pro Artikel mit Landesdurchschnitt

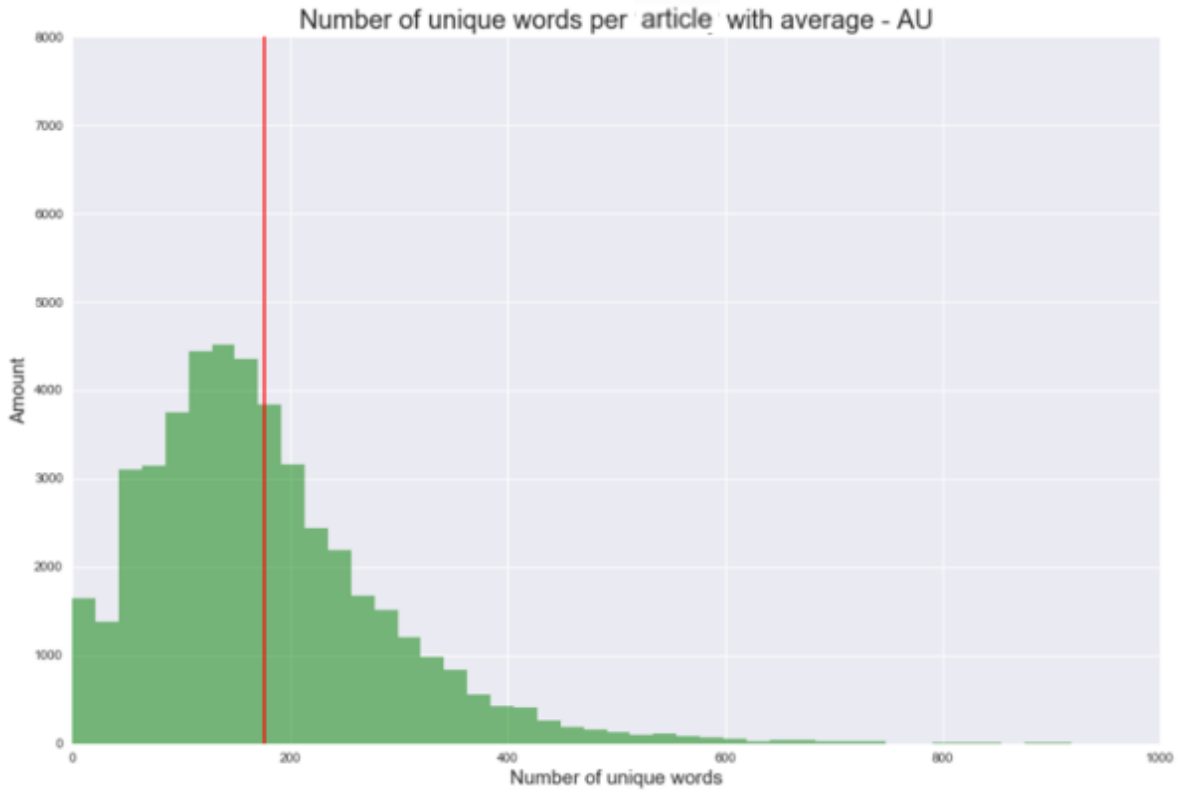
USA:



Großbritannien:



Australien:

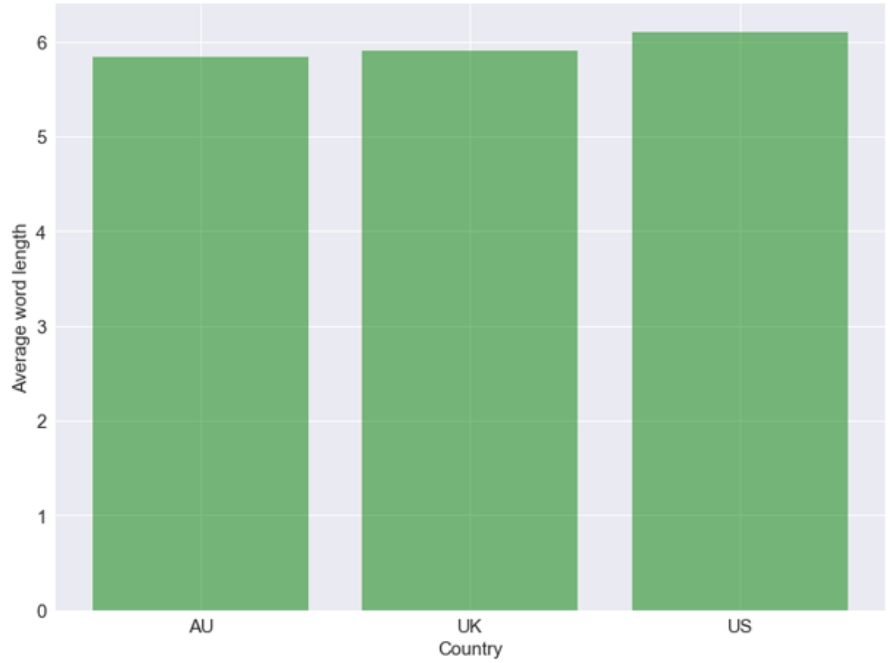


Auch hier müssen die unterschiedlichen Dimensionen der Achsen beachtet werden.

Zwischen den Ländern ist es wieder Großbritannien, die im Bereich von 500 bis 1000 Unique Words die meisten Artikel haben. Dies spiegelt sich in ihrem höheren Durchschnitt.

Vergleicht man diese Grafiken mit „Textlänge pro Artikel mit Landesdurchschnitt“ wird deutlich, dass die Anzahl der Unique Words von der Textlänge abhängig zu sein scheint. Je länger die Texte waren, desto mehr Unique Words sind auch vorhanden. Dafür spricht auch, dass die USA zum Beispiel extrem viele Artikel mit wenigen Unique Words haben, und auch sehr viele Artikel mit kurzem Text.

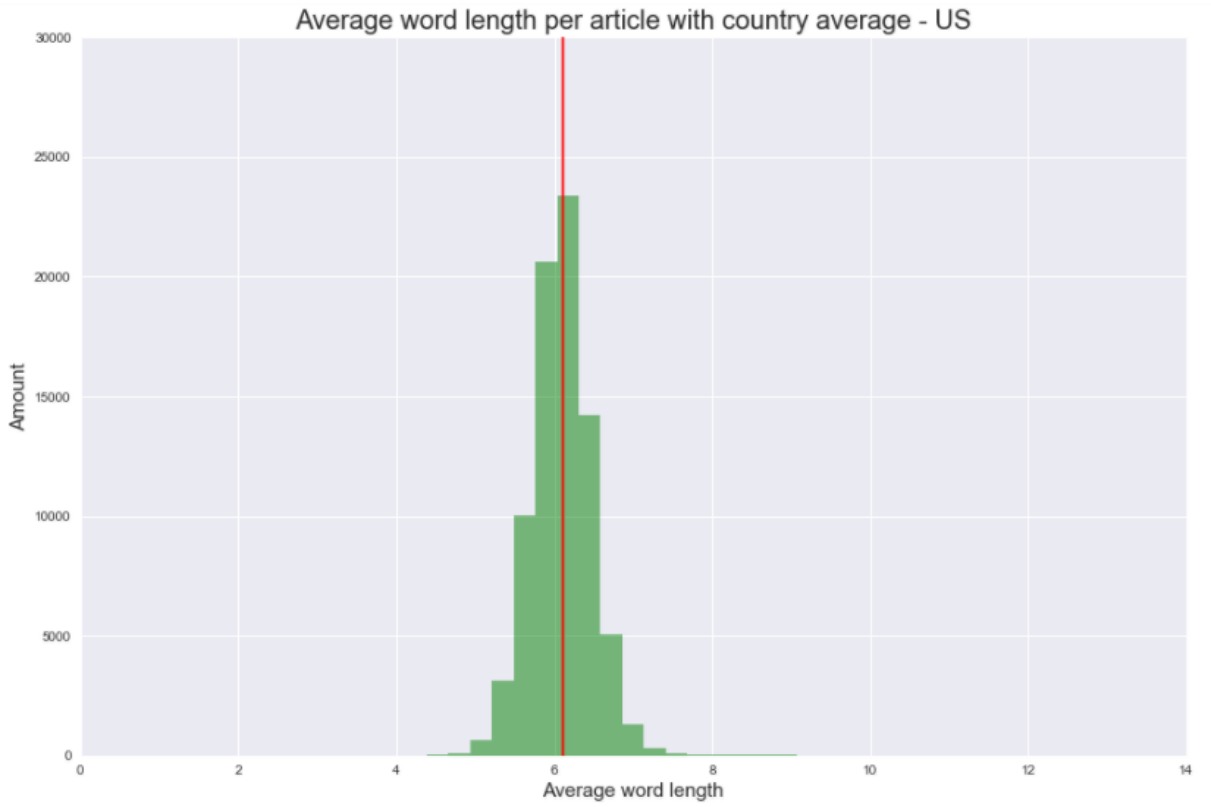
Durchschnittliche Wortlänge pro Land



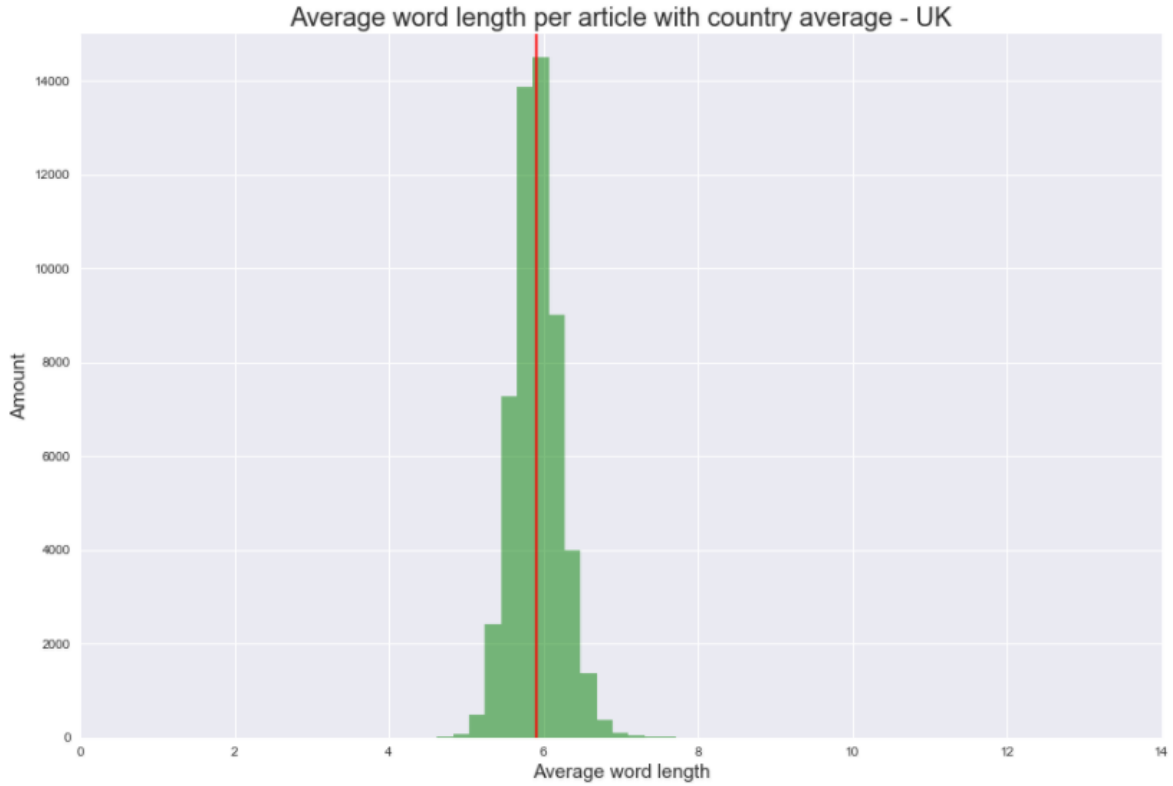
Hier war es interessant zu schauen, ob die verschiedenen englischen Dialekte in der Länge der Wörter zum Ausdruck kommen. Offensichtlich ist das nicht so.

Durchschnittliche Wortlänge pro Artikel mit Landesdurchschnitt

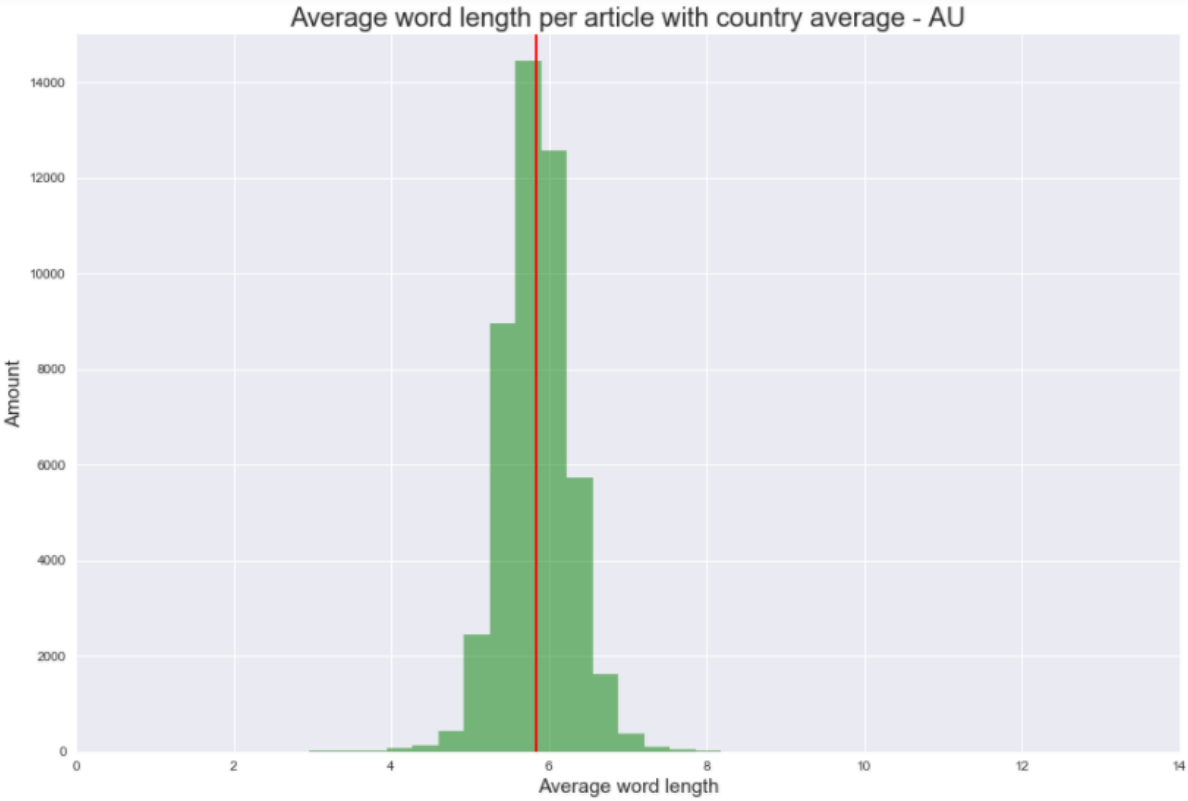
USA:



Großbritannien:



Australien:



Auch anhand der Wortlänge pro Artikel sieht man, dass die meisten Artikel durchgehend fünf bis sieben Buchstaben im Durchschnitt haben. Die Wortlänge innerhalb dieser Analysen zeigt keine relevanten Unterschiede.

3.2.6 Ähnlichkeitsanalyse

In diesem Abschnitt beschäftigen wir uns mit allen Modellen und Methoden, die wir in Bezug auf die inhaltliche Ähnlichkeit von Artikeln zwischen Ländern und zwischen Papern angewendet haben.

Part of Speech

Unter dem Begriff Part of Speech, im folgenden POS genannt, ist die Kategorisierung von Wörtern nach ähnlichen grammatikalischen Strukturen gemeint. Dabei versucht man Wörter nach grammatikalischen Einheiten zu klassifizieren. Durch die Analyse bekommt man Wortarten herausgefiltert, die ein ähnliches Verhalten aufzeigen und die Satzstruktur beschreiben.

Abb.: http://partofspeech.org/

In Python ist die Part of Speech-Analyse eine der wichtigsten Textanalyse-Methoden, um Texte zu klassifizieren. Mit dem Import der NLTK-Bibliothek wird eine Vielzahl von Taggern implementiert, die sich hervorragend eignen, um Wörter einzuordnen.

Zu Beginn unserer Analyse werden die einzelnen Sätze in Tokens aufgeteilt. Hierfür haben wir die Funktion word_tokenize aus der NLTK Bibliothek benutzt. Ein Beispiel hierfür ist der Satz „Less 48 hour President Trump return threeday“.

```
import nltk

text = nltk.word_tokenize("Less 48 hour President Trump return threeday")
print(text)
```

```
['Less', '48', 'hour', 'President', 'Trump', 'return', 'threeday']
```

Anschließend werden die einzelnen Tokens mit sogenannten pos-Tags ihrer Wortart zugehörig klassifiziert. Dies übernimmt eine Funktion namens pos_tag aus der NLTK-Bibliothek. Sofern man dieses auf unseren Beispielsatz anwendet, sieht das Ergebnis wie folgt aus:

```
nltk.pos_tag(text)
```

```
[('Less', 'JJR'),
 ('48', 'CD'),
 ('hour', 'NN'),
 ('President', 'NNP'),
 ('Trump', 'NNP'),
 ('return', 'NN'),
 ('threeday', 'NN')]
```

Beschreibung der POS Tags:

NN - Noun, singular or mass
NNP - Proper noun, singular
CD - Cardinal number
JJR - Adjective, comparative

Quelle:
https://www.ling.upenn.edu/courses/Fall_2003/ling001/penn_treebank_pos.html

Da uns im Laufe unseres Projektes nicht die einzelnen Wörter mit ihrer zugehörigen Wortart interessierten, sondern die Häufigkeit ihres Auftretens, haben wir alle erhaltenen pos-Tags pro Paper und pro Land zusammen gezählt und auf der Summe basierend versucht, eine Auswertung vorzunehmen. Dabei sind immer wieder Probleme aufgetreten, da die Datenmengen zu groß waren. Um dieses Problem zu lösen, haben wir erst jeweils einzelne Dataframes entsprechend einzelner Länder und Paper erzeugt und sie dann durch die POS-Analyse laufen lassen. Da wir im Laufe der Zeit gemerkt haben, dass die Wortarten ungefähr ab der 10. Position kaum mehr nennenswert vorkommen, haben wir unsere Analyse auf die durchschnittliche Anzahl der Top 10 Wortarten eingegrenzt. Ein weiteres Problem, welches vor dem Durchführen unserer POS-Analyse gelöst werden musste, war das Bereinigen der Texte. Anfangs war es uns noch nicht möglich alle Satzzeichen zu hundert Prozent aus den Ursprungstexten zu entfernen (siehe Preprocessing), sodass unser Ergebnis bezüglich der POS-Analyse verfälscht war. Nachdem wir das Problem mit dem Bereinigen der Texte gelöst hatten, konnten wir aber mit der POS-Analyse beginnen.

Beschreibung der POS Tags:

NN - Noun, singular or mass
NNP - Proper noun, singular
JJ - Adjective
VBP - Verb, non-3rd person singular present
RB - Adverb
CD - Cardinal number
VB - Verb, base form
IN - Preposition or subordinating conjunction
NNS - Noun, plural
VBD - Verb, past tense

https://www.ling.upenn.edu/courses/Fall_2003/ling001/penn_treebank_pos.html

Top 10 Wortarten im Durschnitt pro Land

USA:

	NN	NNP	JJ	VBP	RB	CD	VB	IN	NNS	VBD
Number	106.094905	66.671272	37.693216	18.21333	11.283452	9.990233	8.643385	4.363277	3.52289	3.451229

Großbritannien:

	NN	NNP	JJ	VBP	RB	CD	VB	IN	VBD	NNS
Number	188.423804	117.53676	62.125772	30.083462	21.155079	20.926294	15.883145	8.345154	5.684395	5.620987

Australien:

	NN	NNP	JJ	VBP	CD	RB	VB	IN	VBD	NNS
Number	94.01452	64.778132	34.849672	16.30223	13.476901	11.484406	7.865638	4.219572	3.529243	3.346599

Top 10 Wortarten im Durschnitt pro Paper

USA:

ABC News (US)

	NN	NNP	JJ	VBP	RB	CD	VB	IN	NNS	VBD
Number	89.987469	52.63132	31.850717	16.027162	9.306389	8.740765	6.913154	3.693394	3.031146	2.788643

BBS (US)

	NN	NNP	JJ	VBP	RB	CD	VB	IN	NNS	VBD
Number	103.438689	62.55082	36.277377	18.536393	11.26623	9.560656	8.798033	4.034754	3.49377	3.15082

CBS (US)

	NN	NNP	JJ	VBP	RB	CD	VB	IN	NNS	VBD
Number	110.563721	75.915463	38.120504	21.236194	12.313792	11.604645	9.403568	4.618663	4.018267	3.666525

CNN (US)

	NN	NNP	JJ	VBP	CD	RB	VB	IN	NNS	VBD
Number	16.164085	15.587324	5.521127	2.825352	1.940845	1.519718	1.334507	0.646479	0.640141	0.63169

Fox News (US)

	NN	NNP	JJ	VBP	CD	RB	VB	IN	NNS	VBD
Number	88.864836	64.398681	30.254542	15.49854	9.021897	8.985456	6.691447	3.55266	3.005785	2.895329

NPR (US)

	NN	NNP	JJ	VBP	RB	CD	VB	IN	NNS	VBD
Number	138.033412	78.540158	49.535018	23.551938	15.651531	11.994646	11.981581	5.784108	5.004712	3.937674

New York Times (US)

	NN	NNP	JJ	VBP	RB	CD	VB	IN	VBD	NNS
Number	55.765219	32.598978	20.174542	8.125585	5.536611	4.647297	4.335036	2.384845	1.666028	1.55364

Reuters (US)

	NN	NNP	JJ	VBP	CD	RB	VB	IN	VBD	NNS
Number	87.657329	61.989081	31.039815	15.19003	8.059005	7.821882	6.677622	3.453182	2.681482	2.612441

Washington Post (US)

	NN	NNP	JJ	VBP	RB	CD	VB	IN	VBD	NNS
Number	154.501378	88.098946	56.629343	25.253823	17.465766	13.724237	13.357957	6.477817	5.367792	5.013348

Großbritannien:

Daily Express (UK)

	NN	NNP	JJ	VBP	RB	CD	VB	IN	VBD	NNS
Number	81.645533	53.075726	31.465056	14.166045	8.88179	7.732771	6.659802	3.67245	2.773796	2.64229

Daily Mail (UK)

	NN	NNP	JJ	VBP	CD	RB	VB	IN	VBD	NNS
Number	210.935071	118.761501	66.503061	34.263842	25.72532	25.556662	19.229724	9.311025	6.043614	5.764677

The Guardian (UK)

	NN	NNP	JJ	VBP	RB	CD	VB	IN	NNS	VBD
Number	153.323415	67.929206	57.602097	24.448527	16.609885	14.075986	11.927708	6.16026	5.307938	4.287868

The Independent (UK)

	NNP	NN	JJ	VBP	CD	RB	VB	IN	VBD	NNS
Number	242.05667	235.287416	76.7585	33.995641	21.420808	19.234234	15.052892	11.425603	8.730747	8.108108

Australien:

The Mercury (AU)

	NN	NNP	JJ	VBP	CD	RB	VB	IN	VBD	NNS
Number	87.815563	61.166127	32.81081	15.147958	12.205211	11.225704	7.568768	4.014824	3.422887	3.000599

The Daily Telegraph AU (AU)

	NN	NNP	JJ	VBP	RB	CD	VB	IN	VBD	NNS
Number	122.877551	72.431373	45.813125	21.237695	14.533413	11.228491	10.139656	5.256102	4.244898	4.185274

News Australia (AU)

	NN	NNP	JJ	VBP	RB	CD	VB	IN	VBD	NNS
Number	142.067061	107.470217	51.997239	24.189744	15.631164	13.800789	10.881657	6.47574	5.216963	4.712032

The Advertiser (AU)

	NN	NNP	JJ	CD	VBP	RB	VB	IN	VBD	NNS
Number	122.238334	98.668564	43.862053	29.437742	20.867744	14.732529	10.253358	5.159117	4.671523	4.404052

ABC Australia (AU)

	NN	NNP	JJ	VBP	CD	RB	VB	NNS	IN	VBD
Number	109.046023	65.182204	40.701204	20.008025	14.333963	12.14515	8.525608	5.029974	4.921643	3.662497

The West Australian (AU)

	NN	NNP	JJ	VBP	CD	RB	VB	IN	NNS	VBD
Number	51.133021	28.443879	18.693217	8.932777	6.777348	5.729069	4.110409	2.23243	1.781422	1.74231

Anhand unserer Analyse lässt sich erkennen, dass sowohl innerhalb der verschiedenen Länder, wie auch innerhalb der verschiedenen Paper die am häufigsten vorkommenden Wortarten Nomen, Verben, Nummern und Adjektive sind. Daraus lässt sich schlussfolgern, dass die Schreibstile recht ähnlich sind und sich nur durch ihre durchschnittliche Häufigkeit unterscheiden. Dies ist allerdings

darauf zurück zu führen, dass die Artikel in einigen Paper länger sind als in anderen. Generell lässt sich schlussfolgern, dass der Einsatz von verschiedenen Wortarten global gesehen recht ähnlich ist und es keine Unterschiede zwischen den Dialekten gibt. Zum Einsatz der POS-Analyse aus technischer und funktioneller Sicht lässt sich sagen, dass die vorimplementierten Funktionen alle Wortarten richtig erkennen und zuordnen und nur die Dataframes auf den einzelnen Fall abgestimmt werden müssen. Allerdings hat uns die individuelle Anpassung, aufgrund der bereits erwähnten Probleme, relativ viel Zeit gekostet.

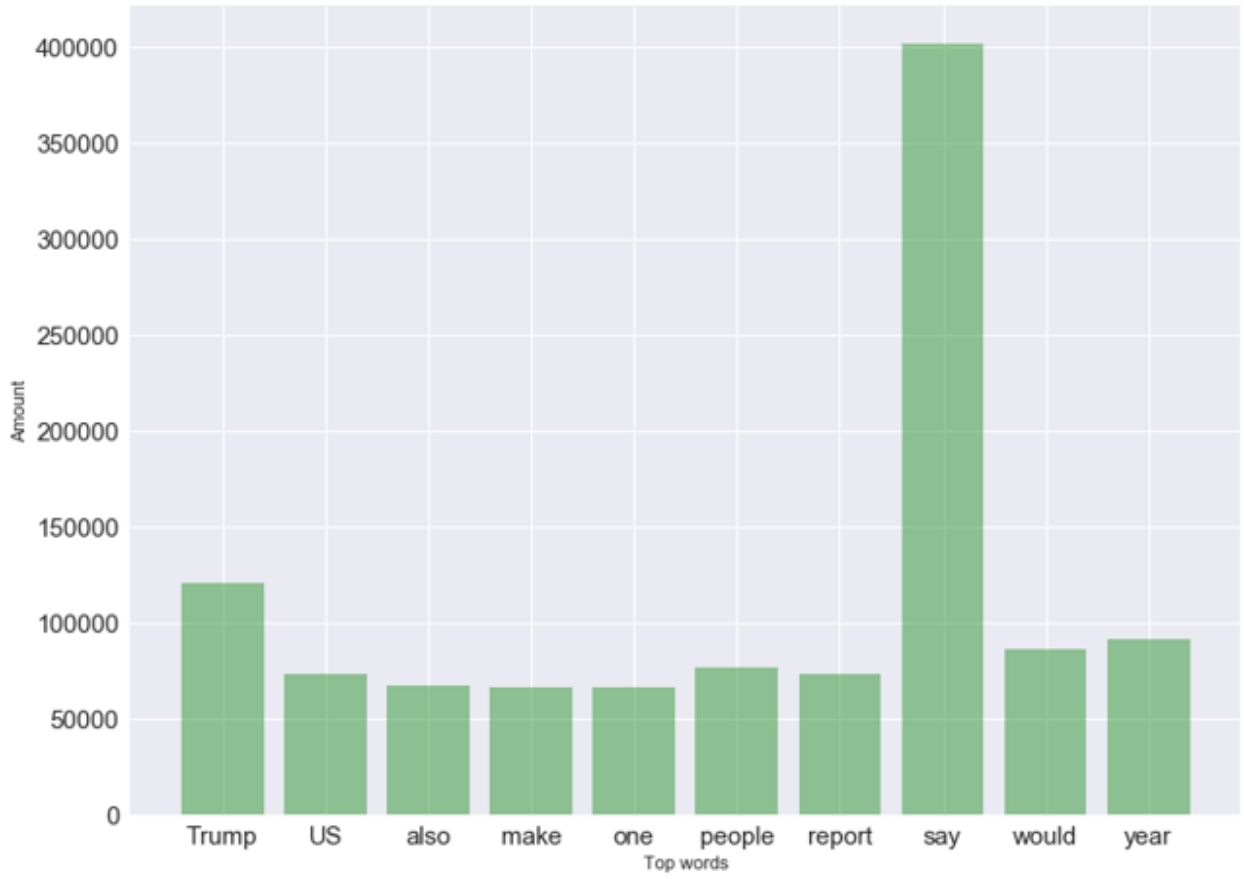
Bag of Words

Bag of Words, im folgenden BOW genannt, ist ein Analysemodell, welches bei der Verarbeitung natürlicher Sprache verwendet wird. Das BOW ist dabei eine reine Darstellungsform von Text, welches aufzeigt wie häufig ein Wort in einem Dokument vorkommt. Das Modell hat seinen Namen, da die Struktur und Reihenfolge des analysierten Dokuments nicht berücksichtig wird. Es sagt nur aus, ob ein bekanntes Wort in einem Dokument vorkommt, egal an welcher Stelle. Weitere Einsatzgebiete in Bezug auf das Thema Machine Learning ist die Umwandlung von Text in eine nummerische Darstellung in Form eines Vektors. Dies wird Vector Space Model genannt und geschieht unter anderem auf der Basis des BOW. Eine BOW-Matrix ist in der Regel so aufgebaut, dass die einzelnen Dokumente die Zeilen darstellen (und je einen Vektor bilden) und die Wörter die Spalten. Der Inhalt der Matrix bzw. der einzelnen Vektoren ist die Anzahl, wie oft das Wort im Dokument vorkommt (Quelle: http://mlwiki.org/index.php/Vector_Space_Models).

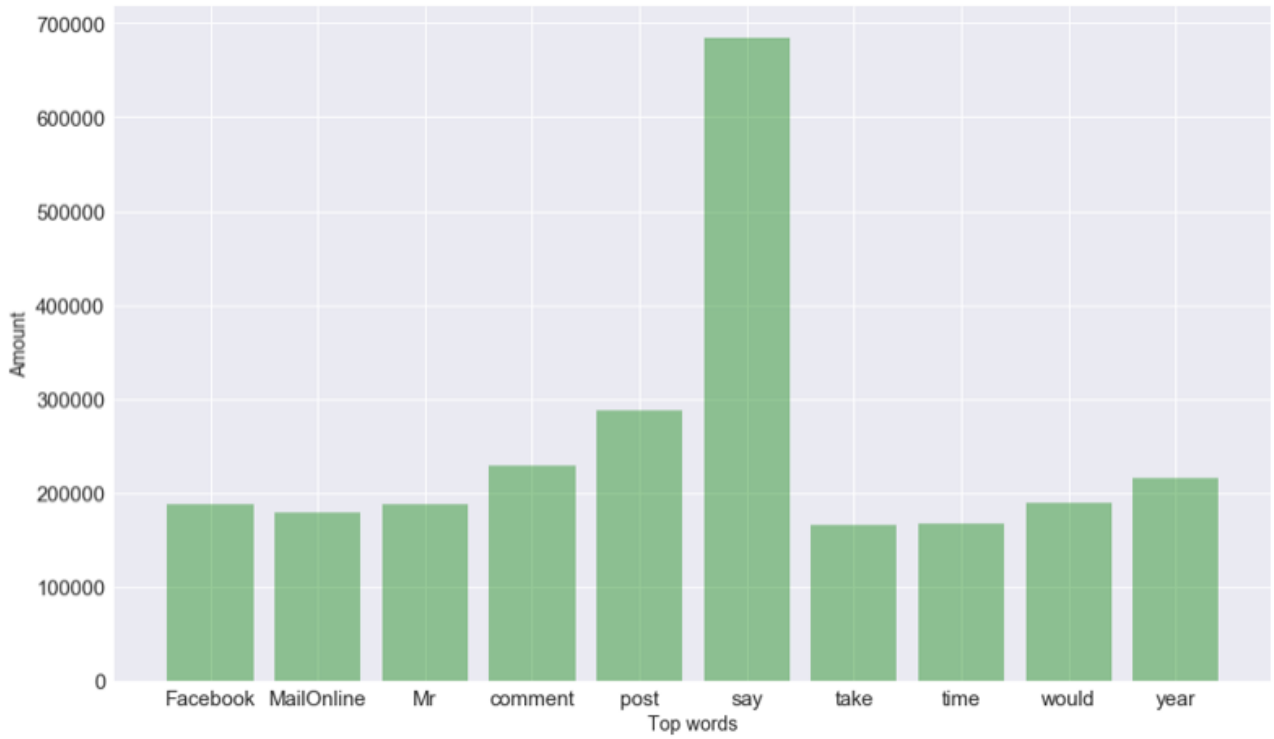
In unserem Fall haben wir BOW anfangs mit Funktionen aus der Bibliothek sklearn.feature_extraction.text implementiert. Allerdings hatten wir hier das Problem, dass wir das BOW-Modell zum einen nicht selber implementierten, sondern nur vorgefertigte Funktionen benutzten und dadurch ein geringerer Lerneffekt bestand. Zum anderen konnten wir die Wörter nicht eindeutig identifizieren, da die Ausgabe über Indizes und generierte Werte dargestellt wurde. So haben wir beispielsweise für einen Artikel die BOW erstellt und konnten die Wörter nicht mehr identifizieren, da „the“ den numerischen Wert 5780 bekommen hat. Die Wörter mussten zur Identifikation einzeln abgefragt werden. Aufgrund dieser Limitierung haben wir uns dazu entschlossen, BOW mit einer selbst geschriebenen Funktion zu implementieren. Hierfür mussten wir nur die Bibliothek collections importieren, die wichtige Klassen wie Counter und Funktionen wie most_common beinhaltet. Mit der Klasse Counter war es uns möglich, die Wörter zu zählen und durch die Funktion most_common nur die Top 10 Wörter auszugeben. Wir haben uns für die Top 10 Wörter entschieden, da ein Abbilden aller Wörter für ein Land innerhalb eines Diagrammes nicht möglich ist.

Top 10 Wörter pro Land

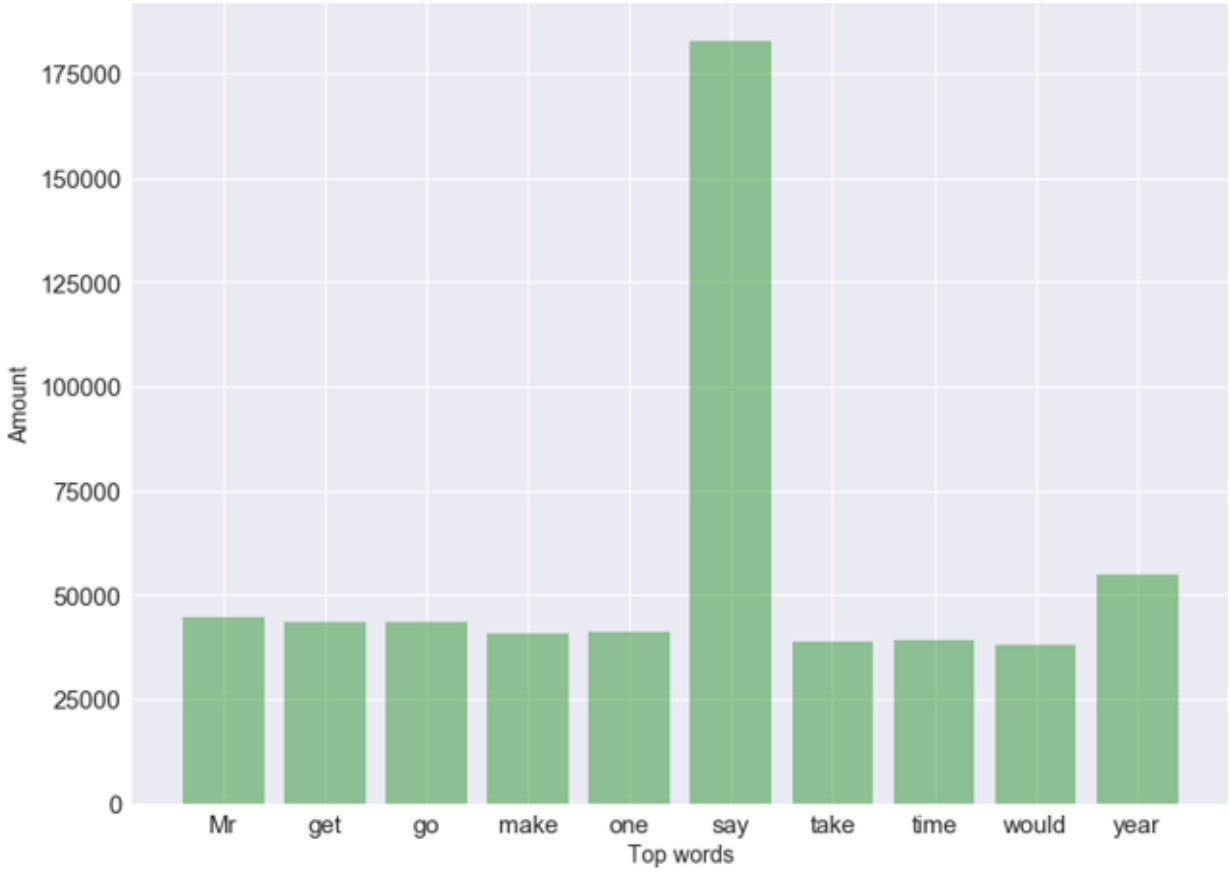
USA:



Großbritannien:



Australien:



Aufgrund der Top 10 BOW-Analyse lassen sich Rückschlüsse auf die aktuellen Themen in den jeweiligen Ländern ziehen. Ein hohes Aufkommen eines Wortes, wie beispielsweise in den amerikanischen Medien das Wort „Trump“, lässt vermuten, dass vieles zu diesem Thema geschrieben wird. Allerdings kann es wie im Fall von Australien auch relativ nichtssagend sein, da hier eher unspezifische Wörter vorkommen. Ein weiteres Problem ist, dass Wörter wie „say“ in jedem Artikel sehr häufig vertreten sind und somit ein hohes Ranking haben, für unsere Analyse aber keine Aussagekraft besitzen. Dies liegt daran, dass im BOW alle Wörter zusammengezählt werden, ohne eine Gewichtung dieser vorzunehmen. Aufgrund dessen macht es mehr Sinn, sich diesem Thema mit dem TF-IDF-Modell zu nähern, bei welchem eine Wortgewichtung vorgenommen wird. Wörter wie „say“ werden dadurch als „weniger wert“ eingestuft und „wichtige“ Wörter kommen an die Spitze. So lassen sich Themen eventuell besser identifizieren.

TF-IDF

Eine weitere Möglichkeit, Text in eine numerische Form umzuwandeln, ist mit dem TF-IDF. Es steht für Term Frequency – Inverse Document Frequency und anders als BOW, welches nur die Anzahl eines Wortes in Dokumenten wiedergibt, berechnet TF-IDF die Relevanz des Wortes für ein Dokument

innerhalb einer Sammlung von Dokumenten. So wird Wörtern, die selten vorkommen, eine höhere Gewichtung zuteil, während häufig vorkommende Wörter eher an Relevanz verlieren. So können sehr verbreitete Wörter wie z.B. „say“ unberücksichtigt bleiben. TF-IDF berechnet sich aus dem Produkt von TF und IDF (Quelle: https://nlp.stanford.edu/IR-book/pdf/06vect.pdf)

Term Frequency: Der erste Teil des Produktes berechnet, wie auch BOW, die Anzahl eines Wortes in einem Dokument. Hier gibt es zusätzlich verschiedene Normalisierungsmethoden, um z.B. in besonders langen Texten eine Verzerrung des Ergebnisses zu verhindern. Wir nutzen die Anpassung der Wortfrequenz an die Länge des Dokumentes, indem wir die absolute Frequenz durch die Länge des Dokumentes dividieren.

Inverse Document Frequency: Hierfür wird die Anzahl aller Dokumente durch die Anzahl der Dokumente, in denen das Wort vorkommt, geteilt. Mit dem Logarithmus dieses Quotienten wird das Ergebnis für IDF berechnet.

Zum Schluss werden die beiden Werte multipliziert. Mathematisch ergibt sich wie folgt:

$$\mathrm{tf}(t,d) = \frac{f_{t,d}}{|\{f_{t',d} : t' \in d\}|}$$

$$\mathrm{idf}(t,D) = \log \frac{N}{|\{d \in D : t \in d\}|}$$

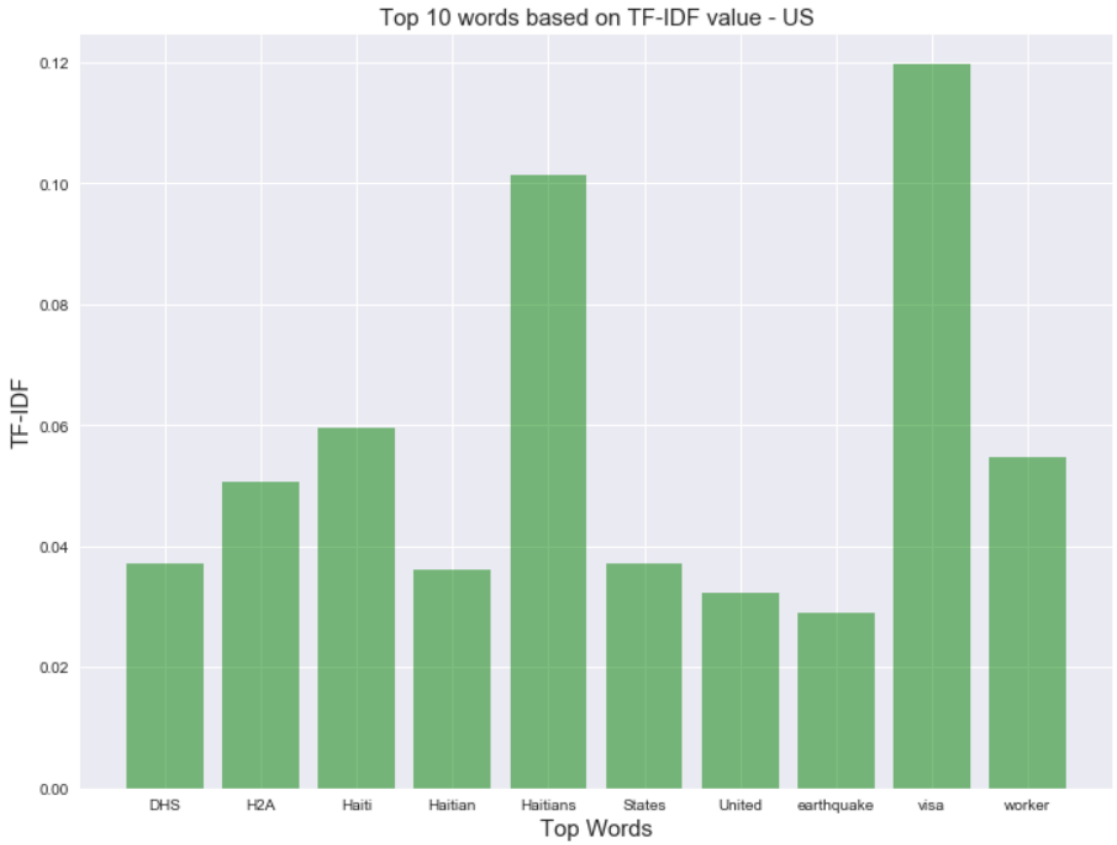
$$\mathrm{tfidf}(t,d,D) = \mathrm{tf}(t,d) \cdot \mathrm{idf}(t,D)$$

Abb.: https://en.wikipedia.org/wiki/Tf-idf

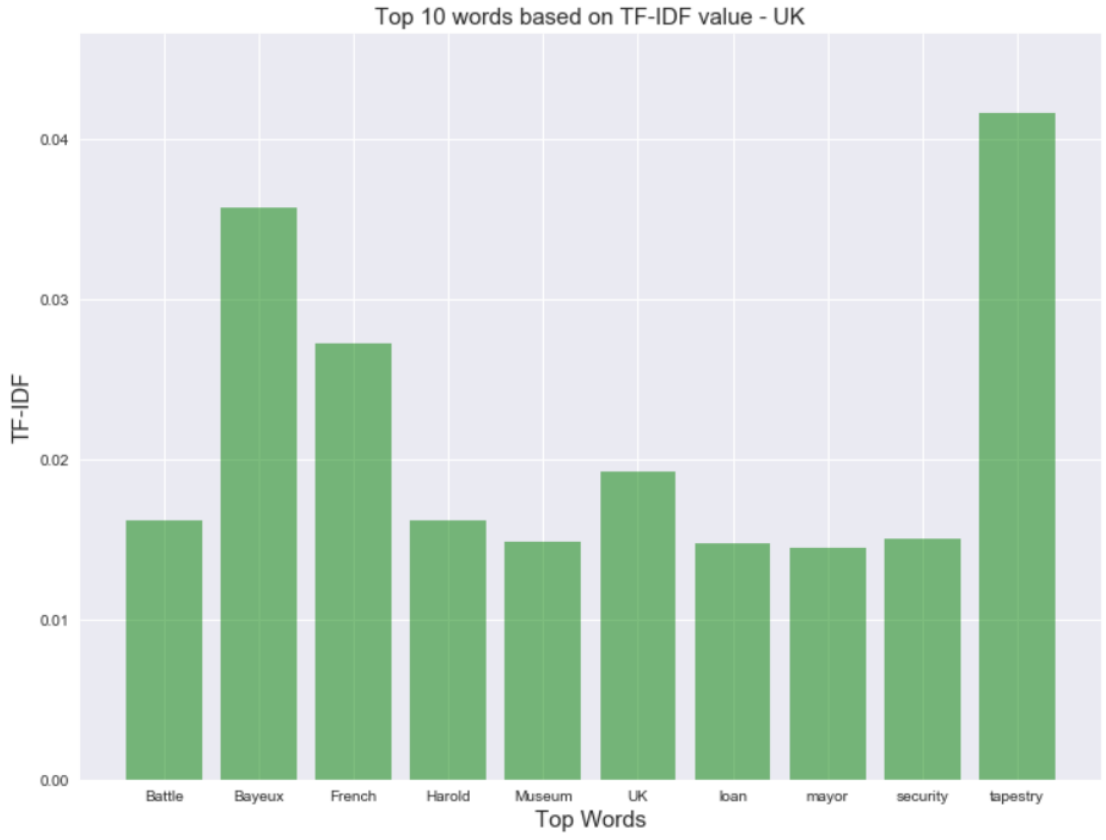
Die Top 10 Wörter pro Land bei der Nutzung von TF-IDF (statt BOW) verändern sich damit hin zu Wörtern, die nicht unbedingt besonders oft benutzt werden, die aber in Bezug auf den gesamten Textkorpus eine höhere Relevanz haben.

Top 10 Wörter pro Land

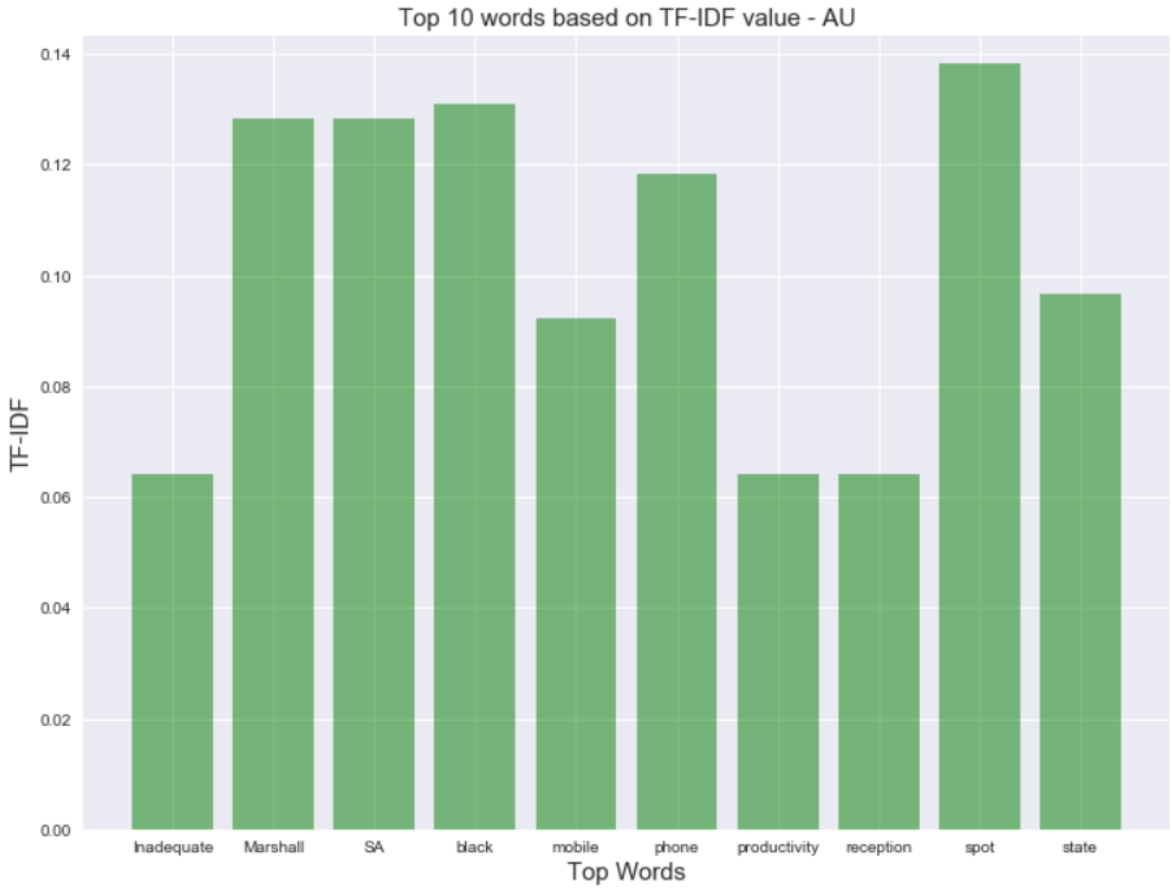
USA:



Großbritannien:



Australien:



Zu den Top Wörtern mit TF-IDF: Wir mussten uns aufgrund eines Memory Errors auf die Auswertung der ersten 100 Artikel beschränken. Zu Methoden für die Verbesserung der Performance siehe Kapitel 3.3 Leistungsanalyse.

Es wäre außerdem noch interessant gewesen, die Methoden, die nachfolgend auf dem BOW basieren, auch auf der Basis von TF-IDF auszuführen. Dazu gehört das Vektorisieren der TF-IDF-Werte, die anschließende Umwandlung in eine Distanzmatrix und der Einsatz dieser Matrix in Multidimensional Scaling und im Ward-Verfahren. Sowohl aufgrund des Memory Errors als auch wegen fehlender Zeit konnten wir diese Aufgaben nicht realisieren.

Erwähnenswert ist zudem, dass wir in der Anfangsphase des Projektes bereits TF-IDF auf andere Weise zu implementieren versucht hatten. Dazu nutzten wir wieder die Bibliothek „sklearn.feature_extraction.text“, diesmal mit den Klassen TfidfTransformer und TfidfVectorizer. Diese Herangehensweise führte zu genau denselben Problemen wie bereits beim BOW-Modell und wurde deshalb erneuert, wobei wir diesmal exakt die mathematische Formel für TF-IDF implementierten.

Gensim

Während der Anfangsphase haben wir die Python-Bibliothek gensim ausprobiert. Gensim beinhaltet Funktionen für die Latente Semantische Analyse (LSA) und das Latente Semantische Indexing (LSI),

welche semantische Zusammenhänge zwischen Dokumenten erkennt. Bei einer Wortsuche werden auch Dokumente gefunden, die nicht das spezifische Wort beinhalten (wie es bei einer reinen Keywortsuche der Fall ist), sondern in den Themenbereich des Wortes fallen. So wird die Qualität der Ergebnisse erhöht (Quelle: http://lsa.colorado.edu/papers/dp1.LSAintro.pdf).
Gensim erstellt im ersten Schritt ein Bag of Words Modell, wandelt dieses dann in TF-IDF-Werte um und diese wiederum in LSI-Werte. Während dieser letzten Transformation wird durch die Übergabe einer Topic-Anzahl versucht, semantische Themen in den Dokumenten automatisch zu identifizieren. Jetzt kann eine semantische Suche vorgenommen werden. Dazu wird Suchwort übergeben, welches ebenfalls in LSI umgewandelt wird. Dieses wird mit dem gesamten LSI-Corpus abgeglichen. Sortiert man die Ergebnisse dann absteigend, erhält man die Indices der semantisch ähnlichsten Artikel zu dem Suchwort.
Auch hier war uns, da wir uns noch relativ früh in der Recherche befanden, nicht klar, dass diese Methoden gelabelte Trainingsdaten und vorher bekannte Topics benötigen. Deshalb war auch unklar, wie mit den Ergebnissen weitergearbeitet werden kann, wie sie visualisiert werden können, etc.

KMeans und MeanShift Clustering

Diese Clustering-Methoden setzten wir in der Anfangsphase ein, bevor wir mit der strukturierteren Arbeitsphase (Metadatenanalyse etc.) begannen. Sie funktionierten zwar in dem Sinne, dass sie ein grafisches Ergebnis lieferten, führten aber nicht weiter, da sie aufgrund fehlenden Vorwissens falsch implementiert wurden und das Ergebnis keinen logischen Sinn ergab. Eigentlich sind die Clustering-Methoden für ungelabelte Daten vorgesehen und arbeiten mit Trainings- und Testdaten. Diese Aufteilung haben wir nicht vorgenommen. Bei KMeans wird die abzubildende Anzahl an Clustern manuell übergeben, so versucht der Algorithmus, die Daten zu gruppieren. MeanShift berechnet die Anzahl der Cluster selbst, um die Daten zu gruppieren und wird in der Praxis eher im Bereich Bildverarbeitung angewandt (Quelle: http://scikit-learn.org/stable/modules/clustering.html). Die Methoden passten weder zu unserem Wissensstand noch zu unserem Vorhaben, die Ähnlichkeit zwischen den Artikeln auszugeben. Der Code hierzu (und für alle weiteren gescheiterten oder nicht verwendeten Methoden) ist innerhalb der ZIP-Datei unter „Versuchscode.py“ zu finden.

NearestNeighbors

Der KNearestNeighbors-Algorithmus ist eine Methode zur Abschätzung von Wahrscheinlichkeitsdichtefunktionen. Dabei wird die einfache Idee herangezogen, unbekannte Werte vorherzusagen, indem sie mit bekannten Werten verglichen werden. Es handelt sich also um eine maschinelle Lernmethode. In den häufigsten Fällen wird die euklidische Distanz verwendet. Es können aber auch andere Entfernungsmetriken zur Berechnung herangezogen werden (Quelle: http://scikit-

learn.org/stable/modules/neighbors.html). In unserem Fall verwenden wir den Algorithmus anders, nämlich um ähnliche Artikel zu einem manuell übergebenen Artikel zu finden.

```
## K-Nearest Neighbors ##
# The NearestNeighbors function can be fit on this sparse matrix and will
# determine the distance between articles based on their TF-IDF values.
# We specify that we want to return the 10 nearest neighbors for each article.

from sklearn.neighbors import NearestNeighbors
# tfidf_all
nbrs = NearestNeighbors(n_neighbors=10).fit(tfidf_matrix)

def get_closest_neighs(name):
    #row = wiki_data.index.get_loc(name)
    distances, indices = nbrs.kneighbors(tfidf_matrix.getrow(name))
    names_similar = pd.Series(indices.flatten()).map(data_frame.reset_index()['Article'])
    result = pd.DataFrame({'distance':distances.flatten(), 'Article':names_similar})
    return result
```

```
print(get_closest_neighs(1))
```

```
                                              Article  distance
0  UNITED NATIONS -  President Donald Trump threa...  0.000000
1  UNITED NATIONS -  President Donald Trump threa...  0.000000
2   President Donald Trump threatened Wednesday t...  0.031921
3  UNITED NATIONS — President Donald Trump threat...  0.246804
4        Updated           December 21, 2017 10:5...  1.198827
5        Updated           December 21, 2017 10:5...  1.198827
6  WASHINGTON — Senior national security official...  1.303869
7  WASHINGTON — Senior national security official...  1.303869
8  **Want FOX News Halftime Report in your inbox ...  1.310683
9  **Want FOX News Halftime Report in your inbox ...  1.310683
```

Diese Methode basiert in unserem Code (s. Versuchscode.py) auf der alten TF-IDF-Implementation und führte letztendlich nicht zu weiterführenden Ergebnissen.

Vectorizing

Bei der Vektorisierung geht es darum, den Text eines Dokumentes in einen Vektor umzuwandeln, damit man auf diesem erzeugten Vektor mathematische Berechnungen durchführen kann. Solche Berechnungen wären auf Textdateien nicht möglich (Quelle: https://de.dariah.eu/tatom/working_with_text.html). In unserem Fall benötigen wir diese erzeugten Vektoren um die Distanzen von Artikeln zueinander zu berechnen.

Bei der Implementierung haben wir die Bibliotheken numpy und sklearn verwendet. Sklearn beinhaltet eine Funktion namens fit_transform, die alle Artikel in einem Dataframe in je einen Vektor umwandelt. Allerdings stoßen wir gleich zu Beginn unserer Implementierung auf einen Memory Error, da das Dataframe zu groß ist. Um dieses Problem zu lösen haben wir anfangs probiert, das Dataframe nicht mit der Klasse CountVectorizer umzuwandeln, sondern mit dem HashingVectorizer, welcher weniger Speicherplatz benötigt. Allerdings bekamen wir auch hiermit immer wieder Memory Errors. Ein weiterer Ansatz war es, das Dataframe per Chunking in kleinere Teile zu zerlegen und anschließend wieder zusammenzusetzen (siehe Leistungsanalyse). Da die einzelnen Chunks auf verschiedene

Vokabulare verweisen passiert es, dass inkompatible Matrizen mit einander zusammen geführt werden, was zu weiteren Problemen führt, da die Matrizen dann nicht mit einander vergleichbar sind. Ein beispielhafter Auszug aus dem Vokabular zeigt die Unique Words der analysierten und vektorisierten Matrizen. Diese unterscheiden sich pro Chunk jedoch.

```
'apprentice', 'approach', 'appropriate', 'approval', 'approve', 'approvingly', 'approximately', 'april', 'aps', 'arab', 'arab
ia', 'archive', 'ardehali', 'area', 'arehttpstco59g6x2f7fd', 'arena', 'arent', 'argentina', 'arguably', 'argue', 'argument',
'arise', 'aristocrat', 'arizona', 'arles', 'arm', 'armed', 'armoured', 'army', 'around', 'arrange', 'arrangement', 'arrest',
'arrival', 'arrive', 'arrow', 'arsenal', 'art', 'arthur', 'article', 'artificial', 'artist', 'artwork', 'asbritain', 'ash',
'ashifa', 'aside', 'asim', 'ask', 'asked', 'asleep', 'ass', 'assail', 'assailant', 'assault', 'assemble', 'assert', 'assess',
'assignment', 'assist', 'assistant', 'associate', 'associated', 'association', 'associations', 'assume', 'assumption', 'assur
ance', 'astonishingly', 'astro', 'astronaut', 'asylum', 'athena', 'athlete', 'atmosphere', 'atmospheres', 'atop', 'attach',
```

Eine Frage, die wir diesbezüglich nicht klären konnten ist: Wie kann man alle Chunks mit der Klasse CountVectorizer vektorisieren, so das am Ende nur eine Vektorenmatrix rauskommt? Aufgrund dessen und damit wir weiter arbeiten konnten haben wir nachfolgende Ähnlichkeitsanalysen auf Vektoren durchgeführt, die maximal 1000 Artikel beinhalten, da dies die Höchstzahl war, die zu keinem Memory Error geführt hat.

Eine Funktion die wir auf den erzeugten Vektoren getestet haben, ist die Suche nach speziellen Wörtern in einem Artikel. So ist es möglich innerhalb der Vektoren nach Wörtern zu suchen, die wir interessant finden und uns deren Häufigkeit in dem Artikel ausgeben zu lassen. Dies ist vor allem dann sinnvoll, wenn man überprüfen möchte, ob aktuell noch über ein bestimmtes Thema geschrieben wird.

Euklidische Distanz und Cosinus Distanz

Nachdem die Artikel in Vektoren umgewandelt sind, können diese miteinander über eine Distanz-Metrik verglichen werden und daraus kann ihre Ähnlichkeit zueinander abgeleitet werden.

In der Bibliothek sklearn haben wir Funktionen gefunden, die sowohl die Cosinus- als auch die euklidische Distanz von Vektoren berechnen können. Da wir bereits auf diversen anderen Webseiten von diesen gängigen Distanzen gelesen haben, entschieden wir uns, beide zu verwenden und für spätere Berechnungen heran zu ziehen (Quelle: https://de.dariah.eu/tatom/working_with_text.html). Dabei berechnen sich die beiden Distanzen wie folgt.

Euklidischer Abstand:

$$d(p,q) = \|q-p\|_2 = \sqrt{(q_1-p_1)^2+\cdots+(q_n-p_n)^2} = \sqrt{\sum_{i=1}^{n}(q_i-p_i)^2}$$

Quelle: https://de.wikipedia.org/wiki/Euklidischer_Abstand

Cosinus-Ähnlichkeit:

$$\text{Kosinus-Ähnlichkeit} = \cos(\theta) = \frac{\mathbf{a}\cdot\mathbf{b}}{\|\mathbf{a}\|_2\|\mathbf{b}\|_2} = \frac{\sum_{i=1}^{n} a_i \cdot b_i}{\sqrt{\sum_{i=1}^{n}(a_i)^2}\cdot\sqrt{\sum_{i=1}^{n}(b_i)^2}}$$

Quelle: https://de.wikipedia.org/wiki/Kosinus-Ähnlichkeit

Für die Implementierung und Berechnung der Distanzen haben wir unsere Vektoren mit den Funktionen euclidean_distances und cosine_similarity berechnet. Die cosine_similarity bzw. Cosinus-Ähnlichkeit ziehen wir von 1 ab, um nicht die Nähe, sondern wie auch beim euklidischen Abstand, die Distanz zu bekommen. Je größer also die Zahl, desto unterschiedlicher sollen die Artikel sein. Hierbei wird eine Matrix erstellt, in der jeder Artikel mit jedem verglichen wird. Daher haben wir in der diagonalen Achse jeweils eine Null-Zeile (wenn nur ein einziges Dataframe übergeben wird), da hier identische Artikel miteinander verglichen werden. Der euklidische Abstand beginnt bei null (identisch) und hat kein oberes Limit. Die Cosinus-Distanz bewegt sich im Bereich von null (identisch) bis eins (keine Gemeinsamkeiten).

Hier folgen zwei Beispiel-Matrizen, die die Ähnlichkeit der untereinander verglichenen Artikel darstellen.

Euklidischer Abstand:

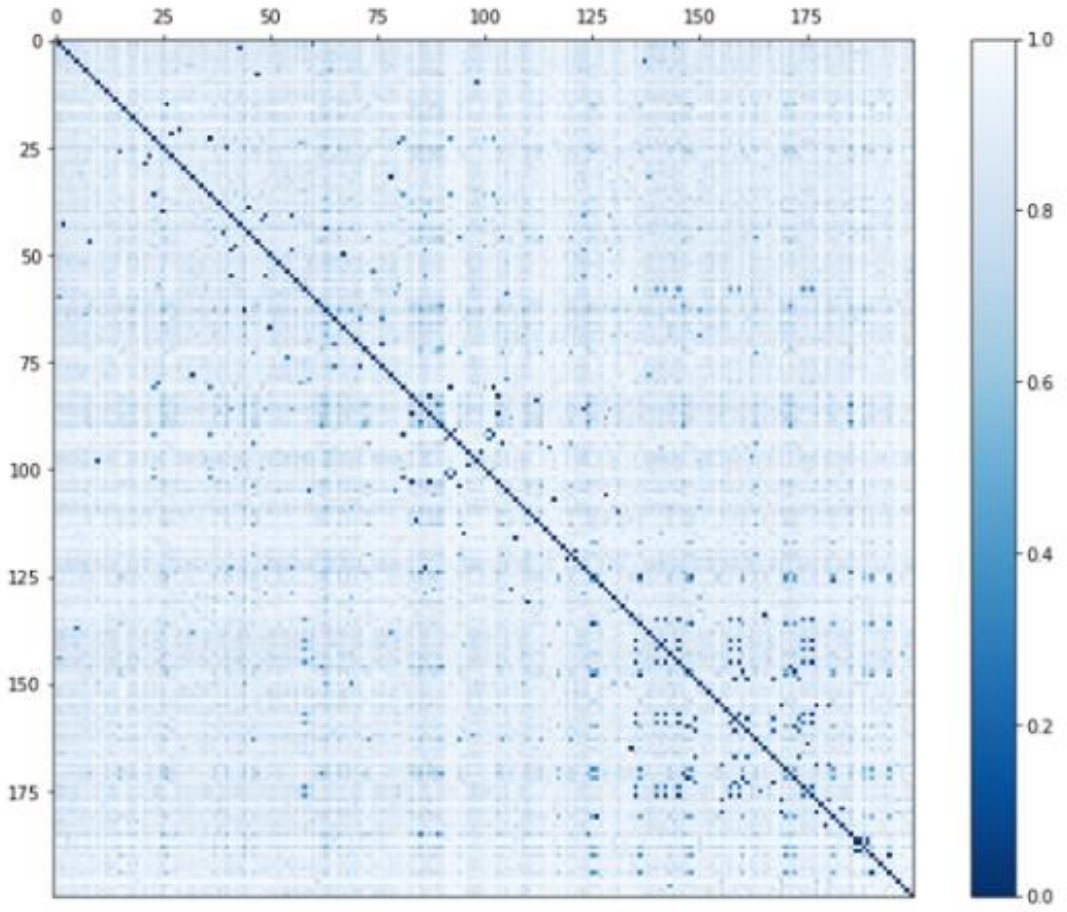
```
array([[  0.   ,  30.984,  34.337, ...,  18.52 ,  20.174,  20.224],
       [ 30.984,   0.   ,  35.93 , ...,  27.258,  27.946,  27.221],
       [ 34.337,  35.93 ,   0.   , ...,  31.016,  31.591,  31.591],
       ...,
       [ 18.52 ,  27.258,  31.016, ...,   0.   ,  10.77 ,  11.402],
       [ 20.174,  27.946,  31.591, ...,  10.77 ,   0.   ,  14.142],
       [ 20.224,  27.221,  31.591, ...,  11.402,  14.142,   0.   ]])
```

Cosinus-Distanz:

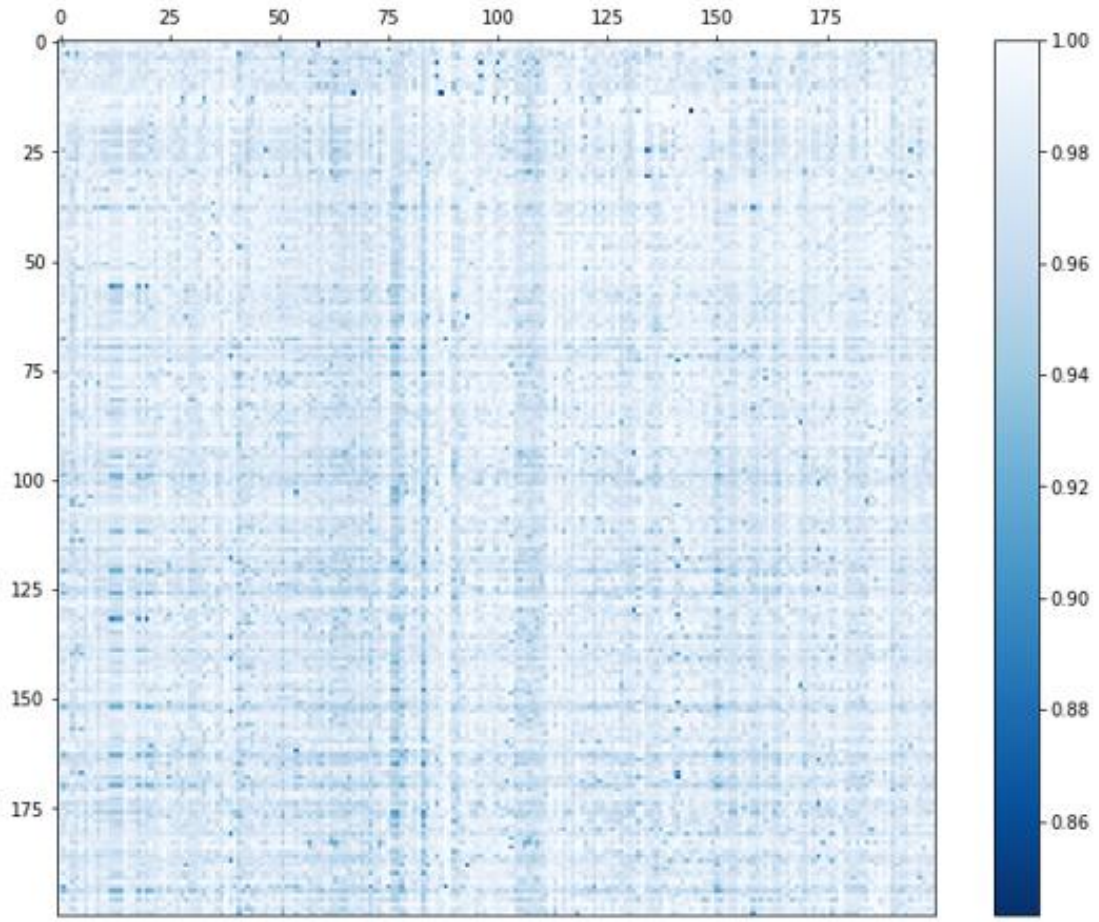
```
array([[ 0.   ,  0.873,  0.89 , ...,  0.925,  0.914,  0.877],
       [ 0.873,  0.   ,  0.748, ...,  0.883,  0.867,  0.78 ],
       [ 0.89 ,  0.748, -0.   , ...,  0.904,  0.883,  0.864],
       ...,
       [ 0.925,  0.883,  0.904, ...,  0.   ,  0.932,  0.876],
       [ 0.914,  0.867,  0.883, ...,  0.932,  0.   ,  0.885],
       [ 0.877,  0.78 ,  0.864, ...,  0.876,  0.885,  0.   ]])
```

Ähnlichkeitsanalysen anhand der Cosinus-Distanz

Hier bilden wir basierend auf der Cosinus-Distanz die Ähnlichkeit von Artikeln ab. Die Beispiel-Visualisierung vergleicht alle US-amerikanischen Artikel miteinander. Die Null-Diagonale ist auch hier gut zu erkennen. Die Skala geht beim Cosinus wie bereits erwähnt von null bis eins. Die Art der Visualisierung kommt aus der Bibliothek matplotlib und heißt interpolation=„nearest“ (Quelle: https://matplotlib.org/examples/images_contours_and_fields/interpolation_methods.html).



Die nächste Beispiel-Grafik ist eine Abbildung der Distanzen zwischen australischen und amerikanischen Artikeln. Hier fängt die Skala erst bei einem Wert von ungefähr 0,84 an und geht bis zu einem von Wert von eins. Dies beweist, dass es keine besonders ähnlichen (und garantiert keine identischen) Artikel zwischen den zwei Ländern gibt.

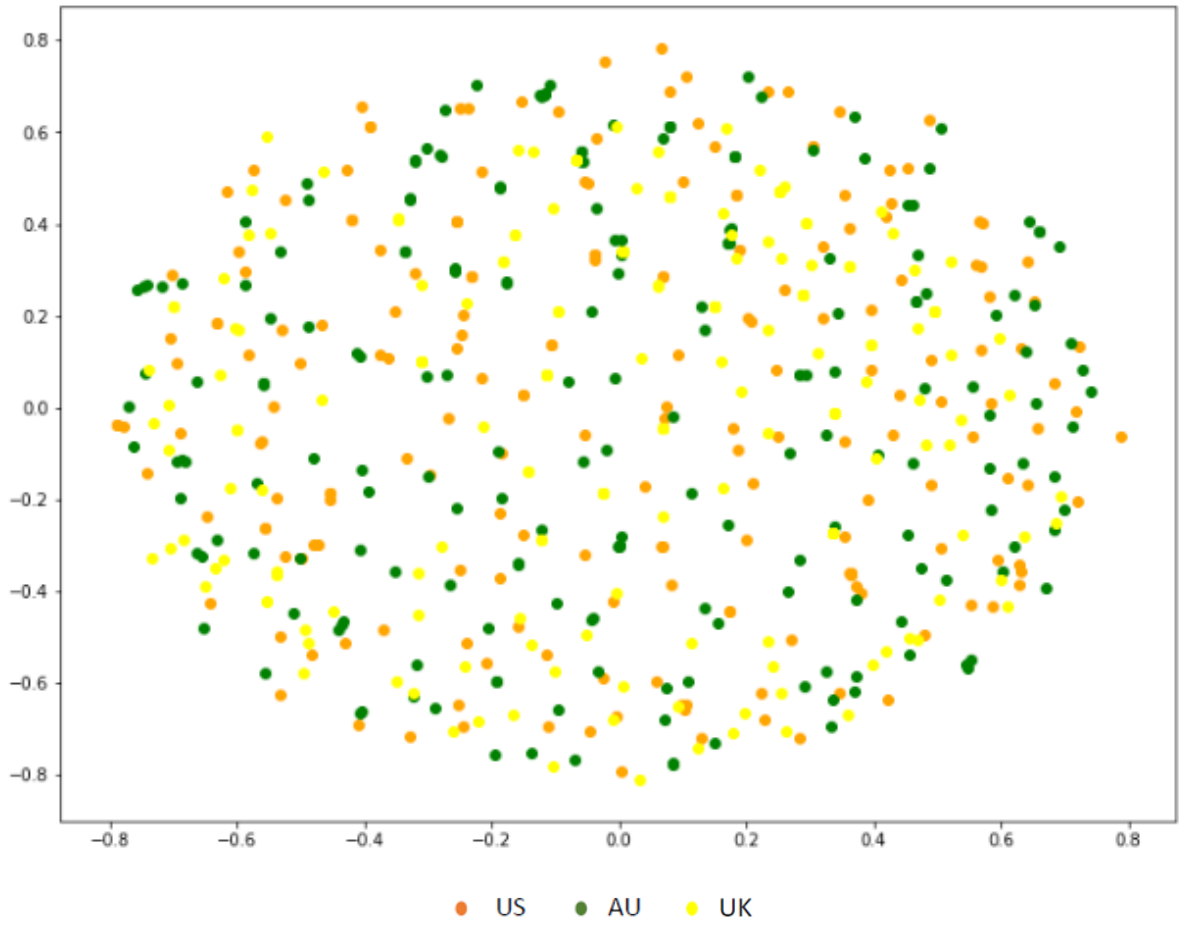


Multidimensional Scaling

Beim Multidimensional Scaling (MDS) sind die Entfernungen zwischen allen Punkten jeweils proportional zu den paarweisen Entfernungen. So können die Distanzen zwischen allen Punkten ohne Verzerrungen und in proportionalen Ausmaßen visualisiert werden (Quelle: https://ncss-wpengine.netdna-ssl.com/wp-content/themes/ncss/pdf/Procedures/NCSS/Multidimensional_Scaling.pdf).

Bereits bevor die Metadatenanalyse begann, führten wir erste Versuche mit MDS durch, die aber nicht weiterführend waren. Dabei übergaben wir der Klasse MDS eine Liste aus Tupeln (Koordinaten), und versuchten sie über KMeans und MeanShift zu visualisieren, nichts von alldem ergibt etwas Sinnvolles. Bei der korrekten Implementierung haben wir wieder die Klasse MDS aus der Python-Bibliothek sklearn.manifold verwendet. Dabei wurde diesmal die Cosinus-Distanz-Matrix aus der Distanzberechnung benutzt, diese mit der Funktion fit_transform der Klasse MDS übergeben und ein

Diagramm geplotted. Dieses haben wir für alle drei Länder (US, UK und AU), für jeweils 100 Artikel durchgeführt, sodass eine Grafik entstand, welche die Entfernungen der Artikel zueinander darstellt. Dabei repräsentieren die orangen Punkte die amerikanischen Zeitschriften, die grünen Punkte die australischen Zeitschriften und die gelben Punkte die britischen Zeitschriften.



Bei der Implementierung des Multidimensional Scaling treten zwei Probleme auf, die wir bis zum jetzigen Zeitpunkt nicht lösen konnten:

Zum einen gibt es keine eindeutige Zuordnung der Punkte zu ihrem Artikel, da keine sinnvolle, lesbare Beschriftung der Punkte möglich ist.

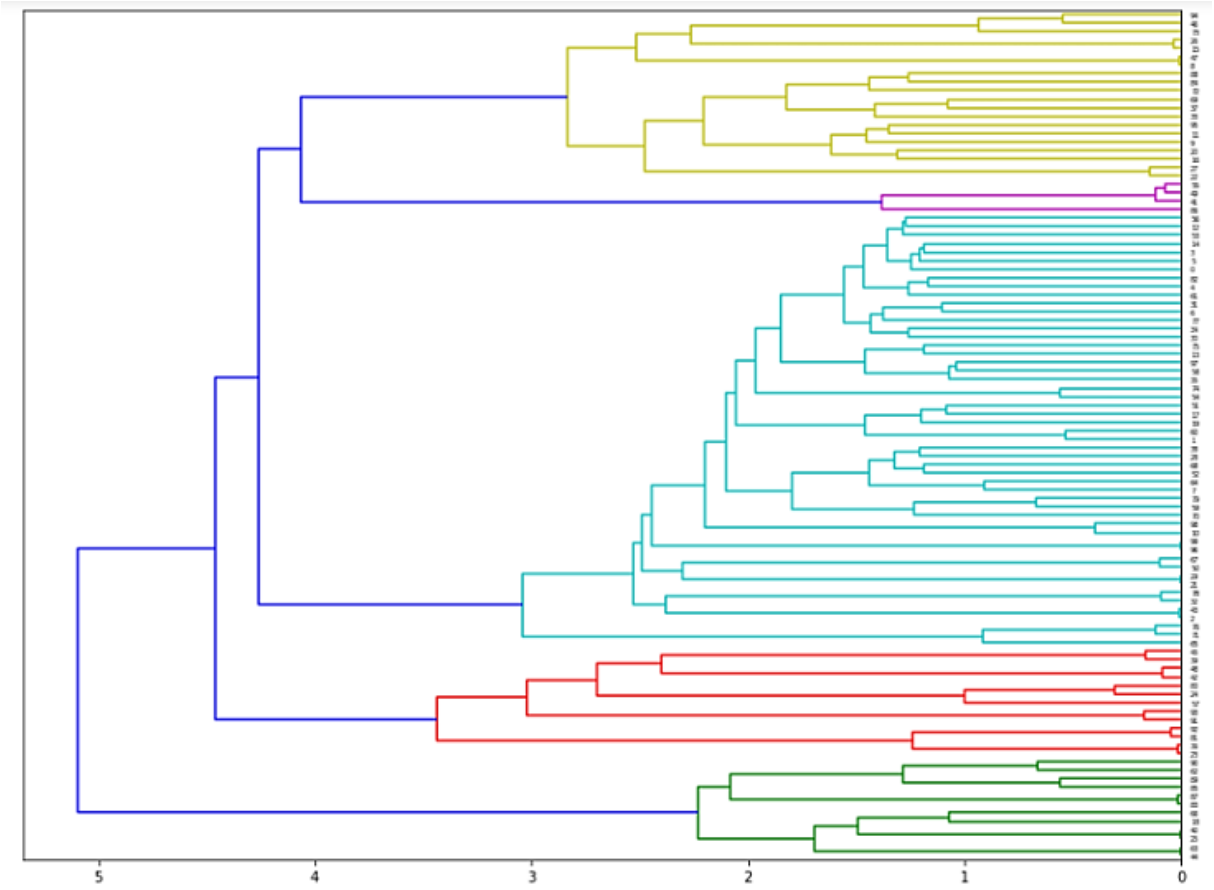
Zum anderen sind es momentan drei separate Grafiken (eine für jedes Land), die in ein einziges Diagramm gespeichert werden. Die Punkte sind also zwar innerhalb der Länder voneinander abhängig und zueinander proportional, dies trifft aber nicht länderübergreifend zu. Würde man die Analyse für alle Länder gleichzeitig auszuführen, könnten die Artikel länderübergreifend voneinander abhängig dargestellt werden. So könnte anhand der Farbgebung beobachtet werden, ob Artikel in ihren Ländern gruppiert oder völlig unabhängig von ihrer Länderzugehörigkeit geclustert werden. Dies würde die Frage beantworten, ob die Länderzugehörigkeit einen Einfluss auf die Ähnlichkeit der Artikel hat.

Ward-Verfahren

Auf Grundlage der berechneten Cosinus-Distanz ist es uns möglich, die Artikel in Clustern zu sortieren. Für das Clustering haben wir das Ward-Verfahren benutzt, welche keine einzelnen Cluster bildet, sondern eine Hierarchie, in der ähnlichere Artikel enger gruppiert werden.

Jeder Artikel wird zuerst wie ein eigener Cluster behandelt. Anschließend wird das nächste Cluster gesucht und mit dem ersten zusammengeführt, so wird ein größeres Cluster gebildet. Dies wird solange durchgeführt, bis ein einziger großer Cluster entstanden ist. Das jeweils nächste Cluster ist dabei immer jenes, dessen Zusammenführung zur minimalen Erhöhung der gesamten Fehlerquadratsumme führen würde. Die wiederum berechnet sich aus der Summe der quadratischen Distanzen der Objekte eines Clusters vom Clustermittelwert (Quelle: http://optiv.de/Methoden/ClustMet/index.htm?17).

Für unsere Implementierung haben wir die Funktion ward aus scipy.cluster.hierarchy benutzt. Sie führt den oben beschriebenen Algorithmus aus und erzeugt die gewünschte Baumstruktur.



Da wir am rechten Rand des Diagrammes nur einen Index der analysierten Artikel (Anzahl 1-100) angezeigt haben, lässt sich nicht genau sagen, ob diese Artikel sich wirklich ähnlich sind. Daher haben wir stichprobenartig anhand der Indizes zwei beieinander geclusterte Artikel identifiziert und die Ähnlichkeit manuell geprüft.

'MAIDUGURI Nigeria double suicide bombing market Nigerias northern city Maiduguri kill least 12 people wound 48 others Wednesda
y resident emergency official say Ibrahim Usman trader Muna Garage market say first bomber attack inside market remain outside
kill Suddenly loud bang everywhere become disorganize Mr Usman say count 12 corps 40 injure victim official charge rescue opera
tion State Emergency Management Agency Bello Dambatta say bomber inside market woman one remain outside man Advertisement Altho
ugh immediate claim responsibility attack similar many carry Islamist extremist group Boko Haram Muna Garage site camp displace
people attack several time past year Boko Haram insurgent Maiduguri capital Borno State refuge thousand people displace insurge
ncy Nigerias northeast Please verify youre robot click box Invalid email address Please reenter must select newsletter subscrib
e View New York Times newsletter Boko Haram form Maiduguri kill 20000 people nineyear insurgency group use woman child suicide
bomber often abduct indoctrinate'

'video release Nigeria week Boko Haram purportedly show schoolgirl abduct town Chibok nearly four year ago vow stay militant ne
ver return home dozen girl young woman clothe Islamic garb appear footage release Sunday Boko Haram wag brutal insurgency north
east Nigeria since 2009 girl face cover see young child Chibok girl one girl say Hausa language video accord translation news a
gency one cry u come back grace Allah never come back video publish Monday news site Sahara Reporters journalist specialize cov
er Boko Haram conflict unclear whether girl fact among 276 kidnap Boko Haram militant board school Chibok northeast Nigerias Bo
rno State April 2014 Nigerian government official respond ABC News repeat request comment video question authenticity Allen Man
asseh spokesman village Chibok tell ABC News cannot verify identity girl see video Boko Haram seek establish Islamic state spre
ad terror across Nigerias mountainous border year Niger Chad Cameroon surround Lake Chad Basin jihadist group whose name roughl
y translate Western education forbid kill 20000 people displace 23 million accord late figure United Nations Boko Harams uprise
fuel largely group systematic campaign abduct child force thousand girl boy rank accord report issue April 2017 United Nations
Childrens Fund UNICEF Still kidnapping Chibok schoolgirl shock world lead launch social medium campaign million people around g
lobe include highprofile political figure celebrity call girl rescue tweet hashtag BringBackOurGirls girl manage escape others
late rescue free follow negotiation fate many remain unknown Manasseh frontline member BringBAckOurGirls movement say group did
nt release statement regard new footage typical propaganda Boko Haram past release similar video purport show miss Chibok girl
Chibok girl new video release Sunday say word Boko Haram leadership want say Manasseh tell ABC News know story cannot deceive n
ew video'

Da die beiden Artikel tatsächlich ähnliche Themen haben (Nigeria), gehen wir davon aus, dass das Ward-Verfahren in unserer Implementation funktioniert und ein erfolgreiches Clustering vornimmt. Weiterführend wäre es interessant, Artikel aller Länder gleichzeitig mit dem Ward-Verfahren zu untersuchen und statt Indices die Länderkürzel als Label anzuzeigen. Wie auch bei MDS könnte man so untersuchen, ob Artikel länderspezifisch geclustert werden und ob die Länderzugehörigkeit einen Einfluss auf die Ähnlichkeit der Artikel hat.

3.3 Leistungsanalyse

Bei der Implementierung traten mehrere Probleme auf, die bereits im Abschnitt 2.2 kurz erwähnt wurden. An dieser Stelle werden zwei Probleme beschrieben, die besonders viel Zeit in Anspruch genommen haben.

1. Performance-Probleme

Während des gesamten Projekts musste darauf geachtet werden, ob der implementierte Code performant genug ist, da im Bereich Big Data mit sehr großen Datenmengen gearbeitet wird. Die Performance-Probleme sind bei der Reintextextraktion, beim Preprocessing, bei der Metadatenanalyse und bei der Ähnlichkeitsanalyse aufgetreten. Dabei musste stets darauf geachtet werden, dass im Code z.B. nicht so viele For-Schleifen vorkommen. Um zu prüfen, ob der implementierte Code performant genug ist, wurde er zuerst auf eine kleinere Anzahl von Artikeln angewendet (meistens nur 100 Artikel) und die dafür notwendige Zeit mit dem timeit-Package gemessen. Hier ist ein Beispiel für ein Performance-Problem beim Preprocessing (s. Versuchscode.py).

In [12]:
```
####### Preprocessing - WORKING VERSION ######
##### Delete Stop Words & Punctuation #####

from nltk.corpus import stopwords
from nltk.tokenize import word_tokenize
import string
import unicodedata
import sys
from nltk.stem import WordNetLemmatizer

lemmatizer = WordNetLemmatizer()
stop_words = set(stopwords.words("english"))

# Iteriert über alle Artikeln und übergibt Zeilenindex von einem Artikel (art_index) und Artikel selbst (art)
for article_index,article in big_dataframe["Article"].astype("U").iteritems():

    # Checkt jedes Zeichen, ob es ein Satzzeichen ist
    nopunc = [char for char in article if char not in string.punctuation]

    # Alle Zeichen wieder zurückjoinen
    nopunc = "".join(nopunc)

    # Löscht alle Stopwörter
    filtered_words = [word for word in nopunc.split() if word.lower() not in stop_words] # filtered_words ist eine Liste aus einz

    x=0
    for word in filtered_words:

        filtered_words[x] = lemmatizer.lemmatize(word)
        x = x+1

    filtered_text = " ".join(filtered_words)

    # Löscht alle Unicode-Zeichen
    tbl = dict.fromkeys(i for i in range(sys.maxunicode)
                        if unicodedata.category(chr(i)).startswith("P"))

    big_dataframe.set_value(article_index, "Article", filtered_text.translate(tbl))
```

Wie in der obigen Abbildung gezeigt ist, wird zeilenweise über das Dataframe iteriert und aus jedem Artikel Satzzeichen und Stopwörter gelöscht. Außerdem wird hier ein Lemmatizer statt PorterStemmer verwendet, um verschiedene Formen von Wörtern zu vermeiden (z.B. anstatt letters wird nur letter gespeichert). Außerdem werden noch Unicode-Zeichen gelöscht.

Der in der Abbildung gezeigte Code ist aber ineffizient: Für das Cleanen von den ersten 100 Artikeln werden 127,69 Sekunden benötigt. Für die Zeitmessung wurde ein timeit-Package benutzt:

```
import timeit

start_time = timeit.default_timer()
longPreprocessing(big_dataframe)
print(timeit.default_timer() - start_time)
```

Es ist nicht schwer zu berechnen, dass für 180 000 Artikel ungefähr 230.400 Sekunden notwendig sind (3.840 Minuten = 64 Stunden). Deswegen wurde der Code auf folgende Art und Weise modifiziert:

```
from nltk.corpus import stopwords
from nltk.tokenize import word_tokenize
import string
import unicodedata
import sys
from nltk.stem import WordNetLemmatizer


lemmatizer = WordNetLemmatizer()
stop_words = set(stopwords.words("english"))

# Iteriert über alle Artikeln und übergibt Zeilenindex von einem Artikel (art_index) und Artikel selbst (art)
for article_index,article in big_dataframe["Article"].head(n=100).astype("U").iteritems():

    # Checkt jedes Zeichen, ob es ein Satzzeichen ist
    nopunc = [char for char in article if char not in string.punctuation]

    # Alle Zeichen wieder zurückjoinen
    nopunc = "".join(nopunc)

    filtered_words = [lemmatizer.lemmatize(word) for word in nopunc.split() if word.lower() not in stop_words]

    filtered_text = " ".join(filtered_words)

    # Löscht alle Unicode-Zeichen
    tbl = dict.fromkeys(i for i in range(sys.maxunicode)
                        if unicodedata.category(chr(i)).startswith("P"))

    big_dataframe.set_value(article_index, "Article", filtered_text.translate(tbl))
```

Nach folgender Codemodifikation werden nur 106 Sekunden für 100 Artikel benötigt. Daraus folgt, dass für 180.000 Artikel 190.800 Sekunden oder 3.180 Minuten oder 53 Stunden notwendig sind. Das ist aber immer noch zu langsam.

Nach darauf folgender Codeanalyse wurde festgestellt, dass der Unicode-Part („Löscht alle Unicode-Zeichen“) sehr ineffizient ist. Deswegen wurde folgende Alternative angewendet:

```
In [46]: def shortPreprocessing(b_dataframe):

    lemmatizer = WordNetLemmatizer()
    stop_words = set(stopwords.words("english"))

    # Iteriert über alle Artikeln und übergibt Zeilenindex von einem Artikel (art_index) und Artikel selbst (art)
    for article_index,article in b_dataframe["Article"].astype("U").iteritems():

        # Löscht alle Unicode-Zeichen
        decodedString = (article.encode("ascii", "ignore")).decode("utf-8")

        # Checkt jedes Zeichen, ob es ein Satzzeichen ist
        nopunc = [char for char in decodedString if char not in string.punctuation]

        # Alle Zeichen wieder zurückjoinen
        nopunc = "".join(nopunc)

        filtered_words = [lemmatizer.lemmatize(word) for word in nopunc.split() if word.lower() not in stop_words]


        filtered_text = " ".join(filtered_words)


        b_dataframe.set_value(article_index, "Article", filtered_text)
    return b_dataframe
```

Nun werden nur ca. 8 Sekunden für 100 Artikel benötigt. Für 180.000 Artikel sind das dann 14.400 Sekunden oder 240 Minuten oder 4 Stunden.

Nach weiteren Versuchen den Code zu beschleunigen (z.B. ein Dictionary für Lemmatizer einrichten), konnten keine Verbesserungen erreicht werden. Jedoch ist der ausgerechnete Wert auch relativ und steht in direkter Abhängigkeit zur Länge der Artikel, die nicht immer gleich ist. Zum Beispiel waren für ca. 32.000 Artikel nur 490 Sekunden nötig:

```
In [133]: us_dataframe=big_dataframe[big_dataframe["Country"]=="US"]
          uk_dataframe=big_dataframe[big_dataframe["Country"]=="UK"]
          au_dataframe=big_dataframe[big_dataframe["Country"]=="AU"]

In [136]: au_dataframe["Article"].count()

Out[136]: 48169

In [137]: uk_dataframe["Article"].count()

Out[137]: 31970

In [138]: us_dataframe["Article"].count()

Out[138]: 95968

In [141]: start_time = timeit.default_timer()
          shortPreprocessing(uk_dataframe)
          print(timeit.default_timer() - start_time)

          491.37862243126256
```

Aus diesem Grund wurde der Code nicht weiter modifiziert.

2. Memory-Error

Der Memory-Error trat beim Preprocessing und der Ähnlichkeitsanalyse auf. Dabei geht es darum, wie es sich bereits aus dem Namen ableiten lässt, dass nicht genug RAM-Speicher für die Bearbeitung vorhanden ist (Quelle: https://medium.com/@AgenceSkoli/how-to-avoid-memory-overloads-using-scikit-learn-f5eb911ae66c). An dieser Stelle wird der erwähnte Error am Beispiel der Ähnlichkeitsanalyse behandelt, da dieses Problem ungelöst geblieben ist.

Um Artikel auf Ähnlichkeit untersuchen zu können, müssen diese zuerst mit Hilfe von CountVectorizer in Document-Term-Matrizen umgewandelt werden. Dabei enthalten diese Matrizen die Häufigkeit für jedes Wort, die in Artikeln vorkommen. Dafür wird ein Vocabulary aufgebaut, das alle möglichen Wörter aus allen Artikeln enthält. Bei 180.000 Artikeln wird dieses Vocabulary schnell umfangreich, so dass ein Memory-Error entsteht:

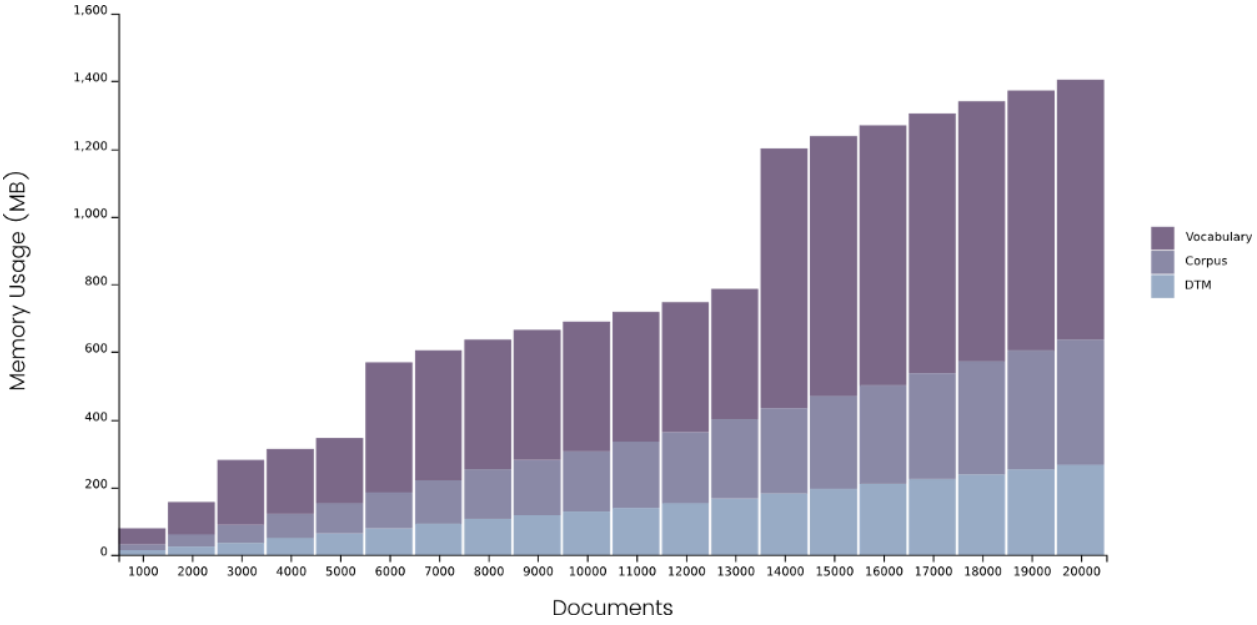


Abb.: https://medium.com/@AgenceSkoli/how-to-avoid-memory-overloads-using-scikit-learn-f5eb911ae66c

Um dieses Problem zu lösen, wurde eine Chunking-Methode (s. Versuchscode.py) angewendet. Dabei wird das gesamte Dataframe in Gruppen (Chunks) aufgeteilt, so dass man über diese Chunks einzeln iterieren kann:

```
n = 500  #Größe der Chunks
list_df = [au_dataframe[i:i+n] for i in range(0,au_dataframe.shape[0],n)] #Teilt das dataframe in kleinere Teile auf
# Output: eine Liste aus Paketen. Jedes Paket enthält eine kleinere Dataframe - Chunk
```

Anschließend kann man über die Chunks iterieren, um die Matrizen aufzubauen. Jedoch ist es nicht so leicht, wie es aussieht, da in jedem Iterationsschritt ein neues Vocabulary aufgebaut wird und die erzeugten Matrizen miteinander inkompatibel sind, da sie auf unterschiedlich aufgebaute Vocabularies verweisen.

Dieser Punkt ist in diesem Projekt noch offen geblieben, so dass die Ähnlichkeitsanalyse nicht auf allen Artikeln durchgeführt werden konnte und nicht aussagekräftig genug ist (s. Abschnitt 3.2.6 „Vectorizing“).

4. Fazit und Ausblick

Der erste Teil unserer Projektarbeit befasst sich mit den Vorbereitungen, die getätigt werden müssen um aus verschiedenen Nachrichtentexten eine einheitliche Datenbasis zu schaffen. Inhalt dieser Vorbereitung war es den Webcrawler zum Laufen zu bekommen und die HTML-Datenstrukturen in eine Reintextform zu überführen. Der nachfolgende Teil beschäftigt sich mit der Vorbereitung der Texte, sprich dem Entfernen von überflüssigen Füllwörtern, Satzzeichen und Sonderzeichen. Dies wird unbedingt benötigt um eine aussagekräftige Datenbasis zu schaffen. Der Hauptteil unserer Projektarbeit befasst sich schlussendlich mit der Datenanalyse von englischsprachigen Nachrichten aus verschiedenen Ländern. Dafür haben wir besonders die vielfältigen Funktionen von Python und seinen Standardbibliotheken bezüglich der Datenanalyse und Visualisierung benutzt.

Ziel unserer Untersuchungen war es, einen Überblick über die Ähnlichkeit von Nachrichtenartikeln aus den USA, Großbritannien und Australien zu bekommen und diese grafisch darzustellen. Bei unserer Umsetzung sind diesbezüglich einige interessante Erkenntnisse zum Vorschein gekommen. Zum Beispiel produzierten die US-amerikanischen Webseiten überproportional mehr Artikel, britische Texte waren dagegen länger und hatten mehr Unique Words. Die Unique Words scheinen außerdem von der Textlänge abhängig zu sein. Des Weiteren hatten die verschiedenen Dialekte keinen Einfluss auf die Wortlänge und die Wortarten (Part of Speech). Distanzen, MDS und das Ward-Verfahren zeigen, dass es durchaus ähnliche Artikel gibt, diese müssten im nächsten Schritt näher untersucht werden. Uns ist bewusst, dass weitere Analysen notwendig sind, um aussagekräftigere Schlüsse zu ziehen.

Abschließend lässt sich sagen, dass sich die verschiedenen Funktionen von Python sehr gut eignen, um große Datenmengen zu analysieren und zu strukturieren. Letztendlich fehlte uns die Zeit, uns weitergehend mit Methoden zu befassen, die einen effizienteren Umgang mit so großen Daten erlaubt hätte. Die Ideen für aussagekräftigere Analysen waren da, doch die Probleme mit den überlasteten Arbeitsspeichern waren der Hauptgrund, dass wir diese nicht umsetzen konnten. Nachdem solche Probleme beispielsweise mit Chunking gelöst werden, wäre als nächstes interessant, die Vektorisierung und alle darauf aufbauenden Ähnlichkeitsanalysen auf Basis von TF-IDF zu gestalten. Weitere nicht umgesetzte Pläne waren, sowohl die MDS-Analyse als auch das Ward-Verfahren für alle Länder gleichzeitig auszuführen, sodass die Artikel länderübergreifend voneinander abhängig dargestellt werden. So könnte im MDS über die Farbgebung und im Ward-Verfahren über die Länderkürzel abgelesen werden, inwieweit die Länderzugehörigkeit die Ähnlichkeit der Artikel beeinflusst.

Die Projektarbeit zeigt, dass es noch vielfältige Möglichkeiten gibt, tiefer in die Textanalyse einzutauchen.

Anhang

Reflektion

Am Anfang unseres Projektes fiel es uns relativ schwer einen roten Faden zu finden, da uns nicht bewusst war, welcher Anspruch bestand und wie wir am besten an das Thema heran gehen. Zunächst mussten wir uns erst in die Programmiersprache Python einarbeiten, wodurch wir oft bezweifelten, ob wir überhaupt irgendwann in der Lage wären, die schwierigeren Konzepte der Textanalyse umzusetzen. Im Laufe der Zeit und nach anfänglichen Recherchen konnten wir erste Methoden und Konzepte herausfiltern, die sich letztendlich nach langem Rumprobieren als zu fortgeschritten oder für unsere Zwecke wertlos herausstellten.

Nur die Implementierung der notwendigen Vorbereitungen und des Preporcessing lief von Anfang an in die richtige Richtung. Ein Verständnis für unser Projektziel gewannen wir aber erst richtig nach einem Gespräch mit unserem Betreuer Dr. Julian Kunkel. Von da an wendete sich die Arbeit und wurde um ein Vielfaches produktiver. Somit begann die erfolgreiche Analyse der Metadaten und schlussendlich die Ähnlichkeitsanalyse.

Es half sehr, nicht so viele uneinsehbare, komplexe Bibliotheken zu benutzen und stattdessen fast alles „per Hand“ zu implementieren. Durch das ständige Programmieren konnten wir viel lernen und praktische Erfahrung sammeln. Wir haben zum Beispiel bemerkt, dass viele Probleme, die am Anfang lange Recherchen nach sich zogen, in der Endphase des Projektes ohne weiteres gelöst werden konnten. Generell gestaltete sich das Recherchieren ab der Metadatenanalyse viel produktiver und effizienter. Am Ende sind wir mit unserem Ergebnis sehr zufrieden.

Da wir im Programmieren allgemein Anfänger sind, haben uns regelmäßige Treffen und der ständige Austausch untereinander sehr geholfen. Den größten Anteil an der erfolgreichen Durchführung des Projektes hatte das Programmieren in der Gruppe. Keiner hat für sich alleine programmiert, sondern der Code entstand nach und nach in Gruppenarbeit. Dies war vor allem auch notwendig, da alles auf einander aufgebaut hat und wir im Voraus nie sicher waren, was im nächsten Schritt zu tun war. Letztendlich kann man sagen, dass die hervorragende Gruppendynamik das Projekt getragen hat.

Arbeitsaufteilung

Aufgrund dessen, dass wir unser Verständnis für die Inhalte, sowie das Programmieren des Codes nach und nach aufgebaut haben, lässt sich nicht feststellen, wer was gemacht hat, da der Code in Zusammenarbeit entstanden ist. So hat beispielsweise jeder an dem Crawler Code gearbeitet, bis dieser funktioniert hat. Dabei hat jeder etwas beigetragen.
Die Arbeit nun fair aufzuteilen scheint uns unmöglich. Wenn wir jemandem ein Kapitel zuordnen, das vermeintlich weniger „wert“ ist als ein anderes, ergibt sich unberechtigter Weise eine schlechtere individuelle Note, was unserer Meinung nach sehr unfair wäre, da jeder den gleichen Aufwand geleistet hat.

Sara:
Code: POS, TF-IDF, Metadatenanalyse, Cosinus-Distanzen, Interpolation-Matrix für Distanzen, Ward-Verfahren, Visualisierungen
Bericht: Einleitung, Design, Metadatenanalyse, Ähnlichkeitsanalyse (TF-IDF, Gensim, KMeans und MeanShift Clustering), Fazit und Ausblick

Alex:
Code: Lemmatizing, Metadatenanalyse, BOW, euklidische Distanzen, MDS, Chunking, Visualisierungen
Bericht: Tools, Ähnlichkeitsanalyse (Part of Speech, Bag of Words, Vectorizing, Euklidische Distanz und Cosinus Distanz, Multidimensional Scaling, Ward-Verfahren), Reflektion

Tatyana:
Code: Cluster, BeautifulSoup, Dataframe-Verwaltung (Pickle usw.), Metadatenanalyse, Vectorizing, Performance-Probleme, Visualisierungen
Bericht: Crawler, BeautifulSoup, Preprocessing, Visualisierung, Ähnlichkeitsanalyse (NearestNeighbors, Ähnlichkeitsanalysen anhand der Cosinus-Distanz), Leistungsanalyse

Alle: Einstieg in Python

Quellenverzeichnis

Python allgemein

https://de.wikipedia.org/wiki/Python_(Programmiersprache)

https://hdm-stuttgart.de/~maucher/Python/html/Datentypen.html

https://py-tutorial-de.readthedocs.io/de/python-3.3/index.html

https://python-kurs.eu/index.php

https://python.org/

Verschiedene Konzepte und Methoden

Manning, C.D.; Raghavan, P.; Schutze, H. (2008). "Scoring, term weighting, and the vector space model". *Introduction to Information Retrieval* (PDF). p. 100. (https://nlp.stanford.edu/IR-book/pdf/06vect.pdf)

http://andrewgaidus.com/Finding_Related_Wikipedia_Articles/

https://buhrmann.github.io/tfidf-analysis.html

https://de.dariah.eu/tatom/working_with_text.html

https://en.wikipedia.org/wiki/Tf-idf

http://lsa.colorado.edu/papers/dp1.LSAintro.pdf

https://medium.com/@AgenceSkoli/how-to-avoid-memory-overloads-using-scikit-learn-f5eb911ae66c

http://mlwiki.org/index.php/Vector_Space_Models

https://ncss-wpengine.netdna-ssl.com/wp-content/themes/ncss/pdf/Procedures/NCSS/Multidimensional_Scaling.pdf

http://optiv.de/Methoden/ClustMet/index.htm?17

https://pythonprogramming.net/

https://radimrehurek.com/gensim/tutorial.html

https://stackoverflow.com/

https://youtube.com/ , diverse Tutorials und Videos zum Thema Python und Bibliotheken

Bibliotheken und Dokumentationen

https://www.crummy.com/software/BeautifulSoup/bs4/doc/

https://matplotlib.org/

https://matplotlib.org/examples/images_contours_and_fields/interpolation_methods.html

https://www.nltk.org/

http://www.numpy.org/

https://pandas.pydata.org/

http://scikit-learn.org/stable/

http://scikit-learn.org/stable/auto_examples/text/document_clustering.html

http://scikit-learn.org/stable/modules/neighbors.html

http://scikit-learn.org/stable/modules/clustering.html

http://scikit-learn.org/stable/tutorial/machine_learning_map/index.html

https://seaborn.pydata.org/