Bachelorarbeit

Semantische Suche auf einem WEB-Korpus

Philipp Bausch

27. 02. 2014

Albert-Ludwigs-Universität Freiburg im Breisgau
Technische Fakultät
Institut für Informatik

Bearbeitungszeitraum
27. 11. 2013 – 27. 02. 2014

Gutachter
Prof. Dr. Hannah Bast
Betreuer
Elmar Haussmann

Erklärung

Hiermit erkläre ich, dass ich diese Abschlussarbeit selbständig verfasst habe, keine anderen als die angegebenen Quellen/Hilfsmittel verwendet habe und alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten Schriften entnommen wurden, als solche kenntlich gemacht habe. Darüber hinaus erkläre ich, dass diese Abschlussarbeit nicht, auch nicht auszugsweise, bereits für eine andere Prüfung angefertigt wurde.

Freiburg, 27.02.14
Ort, Datum

Philipp Bausch

Inhaltsverzeichnis

Zusammenfassung **1**

1 Einleitung **3**

2 CommonCrawl **5**
2.1 Die ARC Datei 5
2.2 Der reine Text 6
2.3 Die Metadaten 6
2.4 Zugriff auf die Daten 7
2.4.1 Der URL-Index 7
2.4.2 Herunterladen und Speichern der Daten 7
2.5 Die Wahl der Seiten 9

3 Verarbeitung der Daten für Broccoli **13**
3.1 Generalisierung der Texte 13
3.1.1 Whitespace entfernen 13
3.2 Die UIMA-Pipeline 14
3.2.1 Grundlagen der Pipeline 14
3.2.2 Komponenten 15
3.2.3 Index Generierung 18

4 Evaluation **19**
4.1 Evaluation der Datenmenge 19
4.2 Analyse und Wichtigkeit der einzelnen Schritte 22
4.2.1 Herunterladen 22
4.2.2 UIMA-Pipeline 23
4.3 Probleme beim Index erstellen 24

5 Fazit **27**

Literaturverzeichnis **29**

Zusammenfassung

Broccoli ist eine auf den Texten von Wikipedia basierte Suchmaschine, welche das Durchsuchen dieser Texte mit dem Durchsuchen von strukturierten Daten (z.B. Ontologien) vereint. Wikipedia beinhaltet aber oft keine Berichte und Beschreibungen für einzelne kleinere Ereignisse. Eine in Politik interessierte Person ist vielleicht auf der Suche nach bestimmten Steuererhöhungsbeschlüssen oder -vorschlägen in den USA. Auf Wikipedia sucht man nach diesen Informationen oft vergebens. Aus diesem Grund wird in dieser Arbeit darauf eingegangen, wie die bisher von broccoli verwendeten Wikipedia Texte durch ein allgemeineren Web-Korpus ersetzt werden. Dieser wurde aus den von CommonCrawl 2012 veröffentlichten Datensatz extrahiert. Dazu ist es nötig die Texte von HTML Formatierungen zu befreien, da die von CommonCrawl angebotenen HTML-losen Texte nicht gezielt abrufbar sind und die Navigation enthalten ist. Hierzu wurde das Programm boilerpipe verwendet. Der Datensatz umfasst 3,8 Milliarden Seiten, die für jeden frei zugänglich sind. Außerdem wird beschrieben, wie Entitäten in den Texten erkannt werden, da dies bisher mit vielen Annahmen über die Strukturen in Artikeln von Wikipedia gelöst wurde. Um die Funktionsweise zu demonstrieren, wird dies für einen Korpus von rund 4,5 Millionen Webseiten untersucht. Dieser besteht aus rund 4 Millionen Seiten der TLD[1] .edu und aus den Archiven von 20 Nachrichten Seiten.

[1] Top-Level-Domain

1 Einleitung

Um das immer größer werdende Internet nach Inhalten zu durchsuchen, wird die Entwicklung von Suchmaschinen und deren Heuristiken für bessere Ergebnisse immer weiter vorangetrieben. Doch noch immer ist es für Suchmaschinen schwer zu verstehen, nach was der Benutzer sucht. Dieses Problem wird durch Verwendung von semantischen Suchmaschinen versucht zu Lösen. Semantische Suchmaschinen versuchen, eine möglichst natürliche Sprache zu verstehen und mit Hilfe dieser die vom Benutzer angeforderten Informationen als Ergebnis zu liefern. Eine der bekanntesten semantischen Suchmaschinen ist der *Knowledge Graph* von *google*. Er gibt dem Nutzer am rechten Rand[1] Informationen über erkannte Entitäten an.
Broccoli [BBBH12] ist eine semantische Volltextsuchmaschine, die versucht, den Anwender beim Erstellen der Suchanfragen bestmöglich zu unterstützen. Semantische Volltextsuche vereint Volltextsuche und die Suche durch strukturierte Daten (z.B. Ontologien). Dies geschieht, indem der Anwender in Echtzeit mit passenden Informationen versorgt wird, die für die Formulierung der Suchanfragen nötig sind. Durch die Verwendung des in [BB13] beschriebenen Index ist dies effizient möglich. In Abbildung 1.1 ist die passende grafische Oberfläche dazu zu sehen.
Der aktuelle Prototyp[2] von *broccoli* verwendet als Korpus Artikel von Wikipedia. Wikipedia ist eine sehr dichte Ansammlung von Informationen, aber diese sind durch die Richtlinien sehr speziell. Wikipedia hat eigens dazu einen Artikel, der beschreibt was Wikipedia **nicht**[3] ist. Ein Abschnitt in diesem Artikel befasst sich damit, dass Wikipedia keine Zeitung ist. Nachrichtenseiten werden selten nachträglich verändert und dann höchstens ergänzt. Im Gegensatz dazu werden die Artikel in Wikipedia stets aktuell gehalten. Der wichtigste Unterschied aber besteht darin, dass aufgrund strenger Relevanzkriterien viele Informationen in Wikipedia fehlen, die in Nachrichtenseiten zu finden sind. In dieser Arbeit geht es darum, Nachrichtenseiten als Ausgangspunkt für den Korpus zu verwenden. Dies macht Informationen auffindbar, die in Wikipedia nicht enthalten sind. Abbildung 1.1 zeigt eine einfache Anfrage, bei der nach Informationen über Steuererhöhungen in den USA gesucht wird. Selbige Anfrage liefert bei dem auf Wikipedia basierenden broccoli keine Ergebnisse.
Als Datenquelle wird auf CommonCrawl zurück gegriffen (siehe Kapitel 2). Die Daten von CommonCrawl werden mit Hilfe eines eigens geschriebenen Programms für die weitere Bearbeitung heruntergeladen. Das Programm selektiert außerdem gezielt HTML und schreibt die Seiten in eine Datei, die danach weiter verarbeitet wird. Zum

[1] Stand Februar 2014
[2] `http://broccoli.informatik.uni-freiburg.de/demos/BroccoliFreebase/`
[3] `http://en.wikipedia.org/wiki/Wikipedia:What_Wikipedia_is_not`

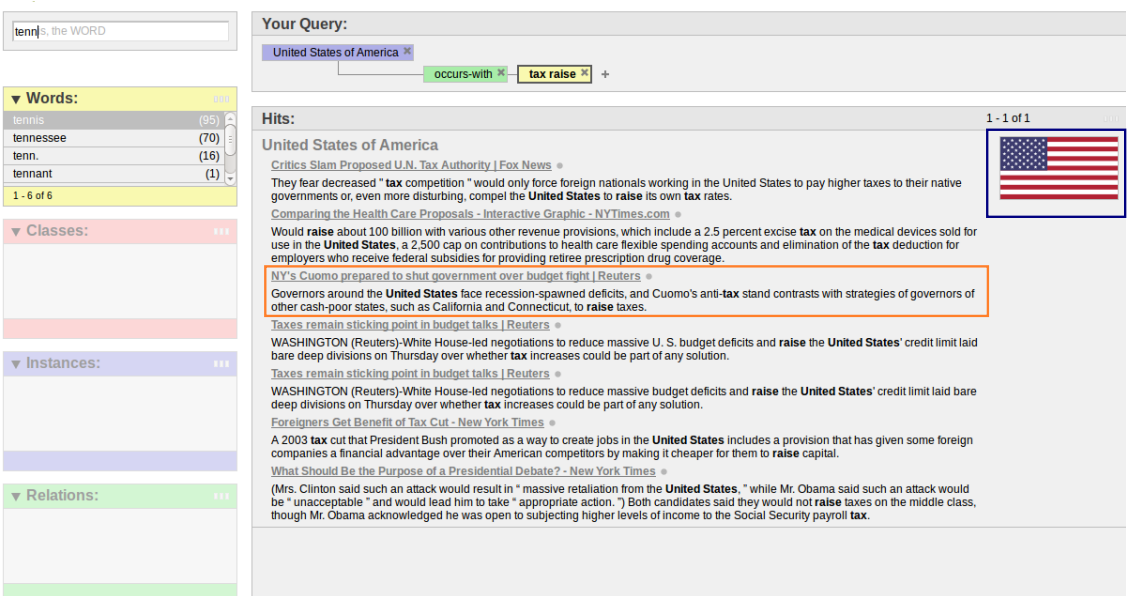


Abbildung 1.1: Die Benutzeroberfläche von broccoli
Hier werden Informationen über Steuererhöhungen in den USA erfragt. Orange ist ein Ergebiseintrag markiert.

Extrahieren des Textes wurde boilerpipe verwendet und analysiert. Ein wichtiger Bestandteil der weiteren Verarbeitung ist es Entitäten in diesen Texten zu erkennen. Dazu wurde ein neuer Entitätenerkenner entwickelt.

2 CommonCrawl

CommonCrawl ist eine ehrenamtliche Stiftung, die es sich zur Aufgabe gemacht hat, die Inhalte der Seiten des Internets zu speichern und für jeden frei zugänglich zu machen. Dies ist für eine Person meist allein durch die entstehenden Kosten nicht realisierbar. CommonCrawl möchte ein möglichst transparentes Web, das Bildung, Forschung und Wirtschaft vorantreibt.
Die Daten sind auf den Servern von *Amazon Web Services*[1] als S3[2] erreichbar. Dies bedeutet, das komplette Datensätze als HTTP-Request oder per *Amazons EC2* abrufbar sind. Insgesamt umfasst das Abbild von 2012 rund 3,8 Milliarden Seiten und ist unkomprimiert über 100 TB groß. Sebastian Spiegler hat die Daten von CommonCrawl genauer in Hinblick auf Codierungen und Dateiformaten untersucht und hat Diagramme und Statistiken aufgestellt um dies zu veranschaulichen [Spi13]. Abbildung 4.5 und Abbildung 4.1 sind Auszüge seiner Arbeit, die den Kontrast zu dem extrahierten Korpus veranschaulichen. CommonCrawl bietet ihre Daten in drei verschiedenen Formaten an[Ste12]; dies sind: *ARC Dateien*, reiner Text und reine Metadaten.

2.1 Die ARC Datei

ARC ist ein Dateiformat, welches von *The Internet Archive*[3] vorgeschlagen wurde um viele kleine Dateien kompakt zu speichern. Dies war vor allem in den 90er Jahren wichtig, da Betriebssysteme nicht mit vielen Dateien in Verzeichnisstrukturen zurecht kamen. CommonCrawl verwendet ARC Dateien um die vollständige HTTP-Response für jede Seite kompakt abzuspeichern. Wie man in Abbildung 2.1 sehen kann, beinhaltet ein Eintrag wichtige Metadaten, wie Zeit des Aufrufs, Codierung der Seite und Komprimierung. Hier ist vor allem die Codierung wichtig. Eine einheitliche Codierung ist für Suchmaschinen essenziell, weil diese alle Anfragen gleich verarbeiten, und man bei der Textverarbeitung keine Artefakte erzeugen möchte (z.B bei 8-Bit ASCII). Um einzelne Einträge einer ARC Datei zu lesen, sollte ein externer Index von Name zu ARC Datei, Offset in der Datei, und Größe des Eintrags erstellt werden.

[1] aws.amazon.com
[2] Simple Storage Service
[3] www.archive.org

```
http://www.spiegel.de/international/0,1518,druck-345720,00.html 195.71.11.67 20120214055058 text/html 24686
HTTP/1.0 200 OK
Date:Tue, 14 Feb 2012 05:48:36 GMT
Server:Apache-Coyote/1.1
X-Powered-By:Servlet 2.4; JBoss-4.0.3SP1 (build: CVSTag=JBoss_4_0_3_SP1 date=200510231054)/Tomcat-5.5
Cache-Control:max-age=120
Expires:Tue, 14 Feb 2012 05:50:37 GMT
X-Host:lnxp-2885
X-Robots-Tag:noindex, nofollow, noarchive
Content-Type:text/html;charset=ISO-8859-1
Vary:Accept-Encoding
Content-Encoding:gzip
Content-Length:8341
X-Cache:MISS from lnxp-3958.srv.mediaways.net
X-Cache-Lookup:MISS from lnxp-3958.srv.mediaways.net:100
Via:1.1 www.spiegel.de, 1.0 lnxp-3958.srv.mediaways.net (squid/3.1.4)
Connection:close
x-commoncrawl-DetectedCharset:ISO-8859-1

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN" "http://www.w3.org/TR/html4/loose.dtd"><head>
...
```

Abbildung 2.1: Eintrag einer Seite in der ARC Datei
Hier ist der Anfang eines Eintrages der ARC Datei zu sehen. In rot sieht man den Inhalt der Ressource, in diesem Fall HTML. In blau sieht man relevante Metadaten, die für die Arbeit interessant waren. Das Feld ***Content-Type*** *wurde beispielsweise genutzt um auf das Format zu schließen.*

2.2 Der reine Text

Der von CommonCrawl angebotene reine Text besteht aus den Inhalten der HTML, RSS und Texteinträge, die in den ARC Dateien beschrieben werden. Für das Vorhaben dieser Arbeit fehlte jedoch ein Index, der je eine URL auf diese Inhalte verlinkt. CommonCrawl speichert diesen nämlich nach einem anderen Schema ab. Wie oben beschrieben, kann man an einzelne Einträge in ARC Dateien durch Offset und Größe gelangen. Die Textdateien sind hingegen sequenziell in Dateien zusammengefasst, sortiert nach Reihenfolge des Verarbeitens. Aus diesen Gründen musste der reine Text selbst produziert werden. CommonCrawl gibt an, dass der reine Text rund einem fünftel der rohen Datenmenge entspricht. Bei dem trivialen extrahieren des Textes ist ein ähnliches Ergebnis zu Stande gekommen.

2.3 Die Metadaten

Die Metadaten sind als JSON Datei abrufbar. Es beinhaltet generelle Informationen über den Ablauf des HTTP-Request, wie Zeit der Anfrage, Größe der erhaltenen Datei, Dateiformat und Codierung. Für HTML, RSS und Atom stehen sogar weitere spezialisierte Attribute zur Verfügung. Die Metadaten beinhalten außerdem Felder, die es ermöglichen auf den Eintrag in einer ARC Datei zuzugreifen. Leider sind die Metadaten wie die Daten des reinen Texts angeordnet. Deshalb wurden auch hier benötigte Metadaten selbst erzeugt.

2.4 Zugriff auf die Daten

Um einen Korpus zu erstellen, der einen logischen Zusammenhang hat, braucht man einen Weg, gezielt an die gewünschten Seiten zu gelangen. Hierzu wurde der URL-Index[4], der von Scott Robertson, dem Gründer von `www.triv.io` [tri13], erstellt wurde, genutzt. Dieser erleichtert die Suche nach Seiten, die in den Korpus integriert werden sollen, da er eine URL und den zugehörigen Eintrag in den ARC Dateien verbindet. Nachdem man herausgefunden hat, welche Einträge benötigt werden, werden diese heruntergeladen und für einen späteren Zeitpunkt auf der Festplatte gespeichert. Diese Schritte werden nun detailliert beschrieben.

2.4.1 Der URL-Index

Der URL-Index nutzt Präfixe um schnell die gewünschten Einträge zu finden. Dies ermöglicht ebenso ganze Web-Domains-/Sub-Domains leicht ausfindig zu machen. Der Index ist ungefähr 218 GB groß, welches das komplette Herunterladen auf Speicherplatz armen Systemen erschwert. Deshalb wurde der Index als Präfix-B-Baum [BU77] erstellt, welcher wahlfreien Zugriff auf einzelne Blöcke ermöglicht. In Abbildung 2.2[5] kann man sehen, wie die einzelnen Blöcke zusammenspielen. Hat man einen Präfix gefunden, der größer ist als jener nach dem man sucht, springt man zu dem zuletzt referenzierten Block. Wenn der Block ein Index-Block ist, wiederholt man diesen Schritt. Falls er ein Daten Block ist, durchsucht man diesen nach dem ersten Vorkommen des Präfixes. Von da ab wird sich jeder Eintrag gemerkt, bis der Präfix nicht mehr zutrifft. Dies kann sich über mehrere Blöcke erstrecken. Die Einträge beinhalten unter anderem den Speicherort der ARC Datei, und den zum extrahieren des Eintrages benötigten Offset.

2.4.2 Herunterladen und Speichern der Daten

Die Implementation die für das Herunterladen der Daten des Korpus verantwortlich war, ist in C++ geschrieben. Die Entscheidung ist durch persönliche Vorliebe und Interesse entstanden. Aber auch der Geschwindigkeitsaspekt wurde in Betracht gezogen. Die Implementation nutzt CURL[6] um wahlfrei auf Index-Blöcke und ARC Einträge aus dem Internet zuzugreifen. Diese werden dann gespeichert um bei wiederholten Ausführungen schneller abrufbar zu sein. Danach werden in einer Art *Pipeline* (siehe Abbildung 2.3) die Codierung geändert, Zeilenumbrüche entfernt, und schließlich die eigentliche TSV Datei erstellt. Durch Nutzung von OpenMP[7] läuft dies parallel

[4] `http://s3.amazonaws.com/aws-publicdatasets/common-crawl/projects/url-index/url-index.1356128792`
[5] `https://github.com/trivio/common_crawl_index/raw/master/docs/tree.png?raw=true`
[6] `http://curl.haxx.se/`
[7] `http://openmp.org/wp/`

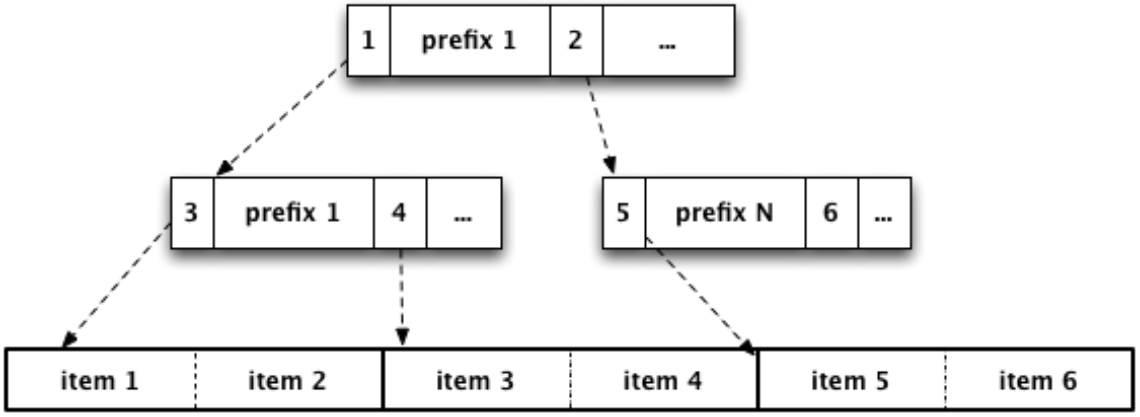


Abbildung 2.2: Der Aufbau des CommonCrawl URL Index
Hier ist zu sehen, wie einzelne Blöcke und deren Einträge auf weitere Blöcke verweisen. Die unterste Ebene beschreibt Datenblöcke, sie beinhalten alle Informationen um direkt auf eine ARC Datei zuzugreifen.

ab und nutzt hohe Bandbreiten, falls diese verfügbar sind, um schnell neue Korpora erzeugen zu können. Abbildung 2.6 beschreibt die Parallelisierung mit Pseudo Code in dem die OpenMP Instruktionen enthalten sind. An dieser Stelle ist es durchaus denkbar, weitere Verarbeitungsschritte einzubauen, um die für I/O-Zugriffe nötige Zeit sinnvoller zu nutzen. Kapitel 4 geht genauer darauf ein, wie diese Erkenntnis gelang.

2.4.2.1 Implementations Details

Es werden nun Teile der Implementation vorgestellt und beschrieben. Bei diesen handelt es sich um den *FileProxy*, *IndexProxy* und *SimpleARCExtractor*.

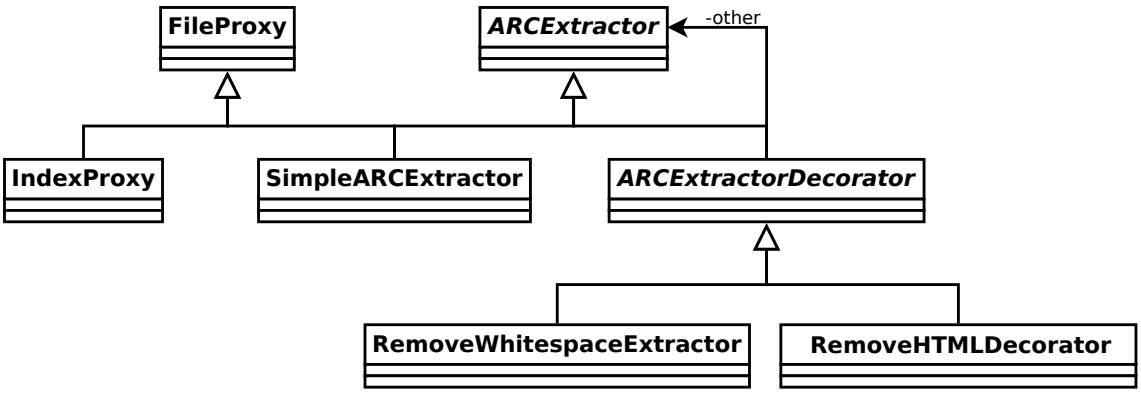


Abbildung 2.3: Ein grober Überblick über den Korpus-Generator
Der rechte Teil des UML Zeigt, wie die einzelnen Komponenten des Extrahierens und Verarbeiten zusammenspielen. Es wurde das ***Decorator-Pattern*** *verwendet. Der FileProxy ermöglicht, Daten direkt von der Festplatte zu laden oder herunterzuladen.*

Der *FileProxy* stellt die Funktionalität bereit, um auf Online Ressourcen zuzugreifen oder sie von der Festplatte zu laden, wenn diese bereits heruntergeladen worden sind. Das bedeutet, man greift nur ein mal auf die Internet-Quelle zu. Dies sollte nur verwendet werden wenn sichergestellt ist, dass die Ressource sich nicht verändert. In Abbildung 2.4 sieht man, wie ein Zugriff auf den Inhalt abstrahiert wurde. Die

```
1 std::string getContents(const std::string& fileName,
2                 const std::string& url,
3                 const size_t startByte,
4                 const size_t endByte,
5                 bool storeOnDisk = true,
6                 CURL* curlHandle = NULL) const;
```

Abbildung 2.4: Auszug aus der Header Datei von *FileProxy*

Möglichkeit Start- und Endbyte nutzen zu können war nötig, um aus großen Dateien nur Teile zu entnehmen. Eine ARC Datei beinhaltet Millionen von Einträgen und oft wurden nur eine Hand voll der Einträge benötigt. Das Mitreichen des *CURL Handle* Pointers macht das Parallelisieren möglich, da diese Methode sonst immer das gleiche Handle benutzt. Der erste Parameter gibt an welcher Dateiname der URL und dem ByteInterval entspricht. Für *IndexProxy* ist die Datei immer die Blocknummer des Indexes. Für den *SimpleARCExtractor* ist es immer eine Nummer, die durch Hashing der URL entstanden ist.
Der *IndexProxy* nutzt und erweitert die Funktionalität des *FileProxy*, um auf einem URL-Index wie in Unterabschnitt 2.4.1 beschrieben zu arbeiten. Die Signatur der Methode in Abbildung 2.5 bietet einen einfachen Weg, eine Liste von *urlLocation_t* der Quellen, die auf den Präfix *request* zutreffen zu erhalten. Die Einträge der Liste

```
1 // Returns the location(a url to the arc file containing the requested page)
2 std::vector<urlLocation_t> getLocation(const std::string& request) const;
```

Abbildung 2.5: Auszug aus der Header Datei von *IndexProxy*

beinhalten jeweils eine URL, und alle nötigen Informationen um den dazugehörigen Eintrag in der ARC Datei zu finden.
Der *SimpleARCExtractor* nutzt wie der *IndexProxy* den *FileProxy*. Er nutzt jedoch *urlLocation_t* Einträge um auf Inhalte zuzugreifen. Diese Inhalte sind Einträge in den ARC Dateien. Diese werden dekomprimiert und es wird dafür gesorgt, dass der Inhalt von der HTTP-Response befreit wird, aus der zuvor noch die Codierung extrahiert wird.

2.5 Die Wahl der Seiten

Die Wahl der Seiten wurde vorerst auf englischsprachige Nachrichten Seiten beschränkt. Der Gedanke hierzu war, Seiten zu nutzen die möglichst viel Informationen enthal-

```
1  // Start of parallel region.
2  // Threads are created here.
3  #pragma omp parallel
4  {
5    // split into chunks
6    while (processed != size) {
7      size_t toDo = min(size - processed, MAXCHUNKSIZE);
8      // Process current chunck
9      #pragma omp for schedule(dynamic)
10     for (size_t i = 0; i < toDo; ++i) {
11       // Access Data
12       // Write Data to first memory of thread
13       ...
14     }
15     // Here is an implicit omp barrier. Threads wait
16     // until all Threads reach this point.
17
18     // swap first memory to be threadsafe.
19     #pragma omp single
20     {
21       // Update how much got processed
22       processed += toDo;
23       // swap first and second memory
24       swap(mem_1, mem_2);
25     }
26     // Here is an implicit omp barrier. Threads wait
27     // until all Threads reach this point.
28
29     #pragma omp single nowait
30     {
31       // Write second memory of each thread into file.
32       ...
33       // Print progress
34       ...
35     }
36     // Here is NO implicit omp barrier. Threads continue.
37   }
38 }
```

Abbildung 2.6: Pseudo-C-Code der Parallelisierung
Diese Code-Stelle beschreibt, wie der Zugriff auf die Inhalte der ARC Dateien parallelisiert wurde. Dies beschleunigt den Ablauf besonders, wenn viele Daten aus dem Internet geladen werden müssen.

ten und möglichst jene, die nicht oder nur wenig auf Wikipedia vorhanden sind. Wenn man den Wikipedia Markup betrachtet, sieht man, das neue Artikel am kleinsten sind. Dies muss bei der Formulierung der Suchanfragen bedacht werden, denn Nachrichtenseiten schreiben beispielsweise nur selten über die Pyramiden von Gizeh. Hierzu wurde kurzerhand eine Liste von Seiten im Internet ausfindig gemacht. `http://news.nettop20.com/` lieferte einen gutes Kernstück mit rund 600 Tausend Seiten, die von CommonCrawl erfasst wurden. Allerdings sollte diese Arbeit deutlich mehr Seiten beinhalten. Deshalb wurden 4 Millionen Domains beigefügt, die auf .edu enden. Diese beiden Quellen, unterscheiden sich bei genauerem betrachten stark voneinander. Beispielsweise kann man die *application/pdf* aus Abbildung 4.3 und Abbildung 4.4 voneinander abziehen und sieht, das der Anteil von PDF Dokumenten auf Nachrichten Seiten weniger als einem Prozent entspricht. Auf **.edu** ist

dieser 15 Prozent. Dies ist damit zu begründen, das Publikationen von Instituten an Universitäten ebenfalls von CommonCrawl erfasst werden. Von diesen Quellen wurden alle Texte benutzt die entweder HTML Formatiert sind oder ohne Formatierung vorliegen.

3 Verarbeitung der Daten für Broccoli

Um allgemeine Suchanfragen auf den heruntergeladenen Texten zu formulieren, muss zuerst die Textmenge generalisiert werden. Die Abbildung 4.2 und Abbildung 4.1 veranschaulichen, wie die Codierungen der verwendeten Teilmenge von dem kompletten Korpus von *CommonCrawl* abweicht. UTF-8 bietet sich als einheitliche Codierung an, da das vorhandene System bereits mit UTF-8 arbeitet, ein großer Teil der Daten bereits in UTF-8 sind und die Bibliothek *boost_locale* die Funktionalität liefert, Texte nach UTF-8 zu konvertieren.
Aus diesen Texten werden dann alle Whitespace entfernt, um die Daten als TSV Datei speichern zu können. Zu dem Zeitpunkt der Bearbeitung liegen die Texte einheitlich (UTF-8; HTML oder Text) in der TSV Datei, die dann mit einer UIMA-Pipeline verarbeitet wird, um Wörter und Entitäten und deren Kontexte zu extrahieren. Der in Abbildung 2.3 angegebene *RemoveHTMLDecorator* wurde nur zu Beginn verwendet und wurde dann durch einen Aufruf von boilerpipe in der UIMA-Pipeline ersetzt.

3.1 Generalisierung der Texte

Die Generalisierung der Texte geschieht in dem C++ Programm, welches auch für das Herunterladen der Daten verwendet wird. Das *Design Pattern* welches in Abbildung 2.3 zu sehen ist, ermöglicht es weitere Verarbeitungen schnell einbauen zu können. Dies sind für die jetzige Verarbeitung das Entfernen von aufeinander folgenden *Whitespace* um TSV Dateien erzeugen zu können.

3.1.1 Whitespace entfernen

Um alle *Whitespace* in einem String durch Leerzeichen zu ersetzen und alle mehrfachen zu entfernen wurde ein Decorator geschrieben der über den Text iteriert, sich das letzte Symbol merkt und Whitespace überspringt, wenn das letzte Zeichen bereits ein Whitespace ist. Whitespace sind in diesem Programm die folgenden Zeichen:Hexadezimalwert; SPACE:0x20, HTAB:0x09, NEWLINE:0x0a, VTAB:0x0b

FEED:0x0c CARRIAGERETURN:0x0d. Dies ermöglicht es eine TSV Datei als Ausgabe zu nutzen, da nur noch das Leerzeichen als Whitespace in den Daten selbst vorkommt.

3.2 Die UIMA-Pipeline

Nachdem die Daten physikalisch in einem einheitlichen Format vorliegen, kann damit begonnen werden die bereits für *broccoli* verwendete UIMA-Pipeline auf die Daten aus dem Web anzupassen. Hierzu muss ein neuer Parser und Entitätenerkenner geschrieben werden. Der Parser liest die TSV Datei und leitet diese weiter. Er liest URL, Titel und Text ein. Die Notwendigkeit eines neuen Entitätenerkenners wurde erst im Bearbeitungsprozess der Arbeit erkenntlich, da an dieser Stelle einige Probleme aufgetaucht sind.
Der Entitätenerkenner versucht Entitäten, aus einer Liste von Entitäten mit Aliase, in den Texten zu erkennen. Um dies zu erreichen muss ein Alias der Entität identisch im Text vorkommen. Dies kann jedoch zu Mehrdeutigkeiten führen. Zum Beispiel ist die AG *Apple Inc.* auch unter dem einfachen Namen *Apple* bekannt. Aber auch das Obst teilt diesen Namen. Deshalb muss an der Stelle im Text entschieden werden, welches die richtige Entität ist. Es existiert außerdem die Frage, ob man Entitäten erlaubt sich zu überlappen. Hierzu wurde eine leicht erweiterbare Klasse geschrieben, die Entitäten Überlappungen, Mehrdeutigkeiten und indirekte Entitäten in allgemeinen Texten eindeutig macht.
Um das Erkennen der Entitäten zu verbessern wurde noch ein POS-Tagger eingebunden. Ein POS-Tagger versucht Sätze in deren Bestandteile zu zerlegen (Subjekt, Prädikat, ..).

3.2.1 Grundlagen der Pipeline

UIMA ist eine Abkürzung für ***U****nstructured* ***I****nformation* ***M****anagement* ***A****rchitecture*. Das Projekt wurde 2005 von IBM gestartet und ist 2006 in die Verwaltung von Apache übergegangen. UIMA eignet sich besonders deshalb für große Datenmengen, da es die Funktionalität bietet, über das Netzwerk verteilt die Daten zu Verarbeiten. Große Teile der verwendeten Pipeline sind in JAVA geschrieben, UIMA bietet aber auch die Möglichkeit an, Komponenten in C++ zu schreiben. UIMA nutzt Annotationen, in der Regel Start- und Endposition in Verbindung mit weiteren Metadaten, um das schnelle Weiterreichen zwischen den Komponenten zu ermöglichen. Diese Struktur macht es einfach, bestehende Elemente der Pipeline für andere Zwecke zu nutzen. Aus diesem Grund, war es möglich, den *Tokenizer*, welcher für Sätze und Wörter Annotationen erstellt, und den *Writer* aus dem bestehenden *broccoli* zu übernehmen. Die Pipeline wird mit Hilfe einer *XML* Datei beschrieben. Sie beschreibt unter anderem wie viele Prozesse benutzt werden, und welche Kom-

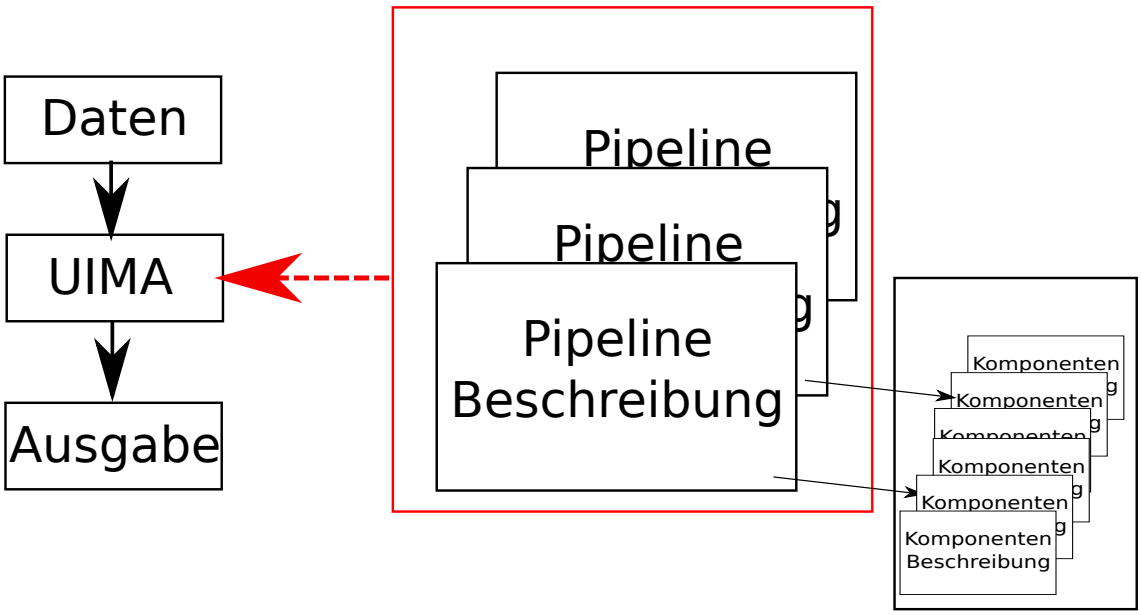


Abbildung 3.1: Der Aufbau der UIMA-Pipeline
Rechts sieht man die einzelnen Komponenten Beschreibungen, die dann an das UIMA Framework weitergereicht werden, welches dann die Daten gemäß der Beschreibungen verarbeitet.

ponenten zum Einsatz kommen. UIMA verwendet diese Beschreibung um die Daten danach selbstständig verarbeiten zu können Abbildung 3.1.

3.2.2 Komponenten

UIMA sieht es vor Texte in einer Pipeline zu verarbeiten, um eine einheitliche Struktur zum Verarbeiten großer Datenmengen bereit zu stellen. Dies wird in der Regel in 3 Schritten durchgeführt. Als ersten müssen die Daten eingelesen werden. Danach können diese verarbeitet werden. Am Ende werden die Daten "verbraucht". Alle Komponenten können einer dieser Kategorie zugewiesen werden. In dieser Arbeit nutzt die Pipeline 5 Komponenten. Der *Parser*, der die TSV Datei einliest und die Texte mit Hilfe von *boilerpipe* von HTML befreit. Der *Tokenizer*, der durch Heuristiken Sätze und Wörter annotiert. Der *Entitätenerkenner*, der Entitäten im Text durch String Vergleiche versucht ausfindig zu machen und diese dann mit Hilfe von Regeln eindeutig macht. Sowie der *Writer*, der alle erkannten Entitäten und Wörter in Dateien schreibt um daraus einen Index für *broccoli* zu generieren (Abbildung 3.2).

3.2.2.1 Parser

Der Parser ist eine UIMA-Komponente, welche dafür sorgt, das die Prozessoren weitere Daten erhalten. Sie ließt die erzeugte TSV Datei zeilenweise ein, und schreibt

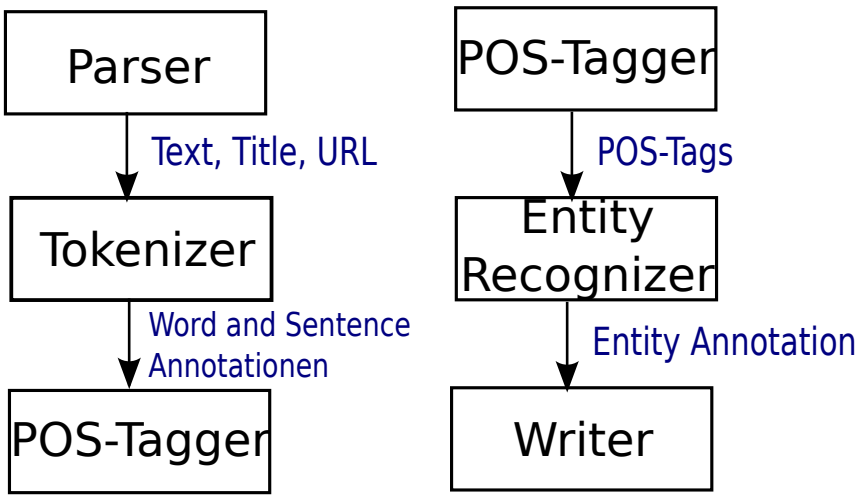


Abbildung 3.2: Der Aufbau der verwendeten Pipeline
Der Aufbau der Verwendeten Pipeline. In blau sieht man, was in jedem Schritt beigefügt wird. D.h. der Writer kann auf alle Annotationen zugreifen.

Annotationen, die bereits in diesem Schritt feststehen. Dies sind: Dokumenttext und -größe; Titel; URL. Zum Zeitpunkt des Einlesens, sind große Teile der Texte noch als HTML formatiert. Aus diesem Grund, wird *boilerpipe* benutzt. Boilerpipe ist ein in JAVA geschriebenes Projekt, das HTML formatierte Texte von den Werbungen, Navigationen und anderen Textunterbrechungen befreit. Diese Bibliothek benutzt und erweitert Konzepte von [KFN10]. Boilerpipe wurde vor allem deshalb benutzt, da es schnell zu guten Ergebnissen gelangt. Boilerpipe extrahiert nach Regeln, die durch einfache Features, wie durchschnittliche Satz Länge, Texte aus HTML Dateien. In dieser Anwendung wurde der *Artikel Extrahierer* verwendet, da dieser vor allem auf Nachrichten Seiten den Artikel findet und von HTML befreit ausgibt. In der Evaluation wird darauf kurz eingegangen. Ab diesem Punkt ist in der Pipeline kein HTML mehr aufzufinden. Alle weiteren Verarbeitungsschritte verwenden den reinen Text, der daraus extrahiert wurde.

3.2.2.2 Der Tokenizer

Der Tokenizer ist eine von *broccoli* übernommene Komponente, die den Text mit Heuristiken versucht in Wörter und Sätze zu unterteilen. Dies ist ein wichtiger Bestandteil der Pipeline, da sowohl der Entitätenerkenner als auch der Writer mit diesen Markierungen arbeiten. Die Markierungen dienen dazu, so einfach wie möglich über den Text laufen zu können. Es wird einfach über die Liste aller Markierungen iteriert und Wort für Wort und Satz für Satz die folgenden Verarbeitungsschritte durchgeführt.

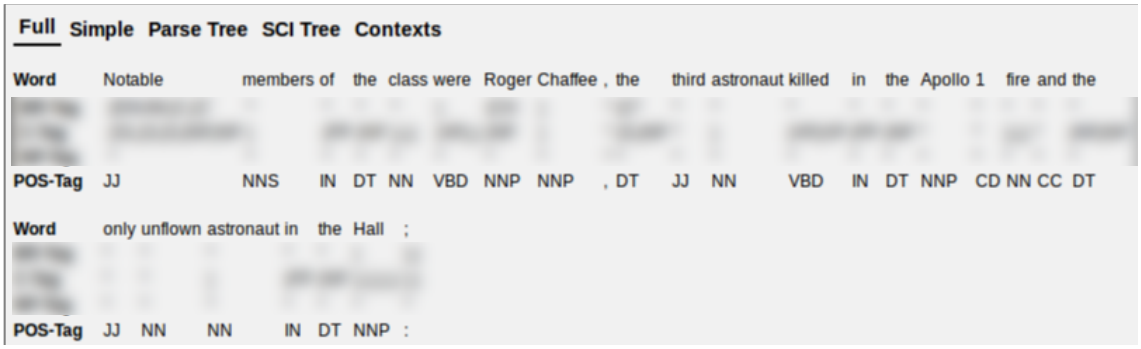


Abbildung 3.3: Beispiel POS-Tagging Ausgabe im Broccoli Interface
Man sieht, welchem Wort der POS-Tagger welchen Tag zugewiessen hat. VBD gibt an, dass dies ein Verb in Vergangenheitsform ist. NNS steht für ein Nomen in Pluralform. CC wird benutzt um Satzverbindungen wie 'und' und 'oder' anzugeben.

3.2.2.3 POS-Tagger

Diese Komponente dient hauptsächlich dazu, *False Positives* in der Entitätenerkennung zu vermeiden. Durch die Hinzunahme von Part-Of-Speach Tagging, kann man das Erkennen der Entitäten auf Nomen beschränken. Dies ist durchaus sinnvoll, da es eine Vielzahl von Entitäten gibt, die einem Adjektiv gleichen. In der Evaluation wird darauf genauer eingegangen. Abbildung 3.3 zeigt die Ausgabe in dem WebInterface. Man sieht, dass der Text mit POS-Tags markiert ist, für die weitere Verarbeitung der Tags im Entitätenerkenner wurden nur die Wörter benutzt, die mit einem N beginnen. Dies sind Nomen jeglicher Form. Der Stanford POS-Tagger verwendet Ansätze aus [TKMS03] um diese Aufgabe zu lösen.[1]

3.2.2.4 Entitätenerkenner

Entitäten in Texten zu erkennen ist ein Problem, das bei semantischen Suchen immer wieder einen Kernpunkt des Problems darstellt. Wenn ein Mensch einen Text ließt kann er fast sofort feststellen, um was es sich bei diesem Wort oder Textausschnitt handelt. Dies ist für Computer noch immer ein Problem, obwohl dies bereits seit 20 Jahren untersucht wird. Deshalb muss man noch immer für verschiedene Daten spezialisierte Entitätenerkenner schreiben.
Bei den verwendeten Daten handelt es sich um Text der aus Seiten aus dem Web mit boilerpipe extrahiert wurden. Außerdem wurden Zeilenumbrüche beim Erstellen der TSV Datei entfernt, welches das Erkennen von Zeilenumbrüchen zum Erkennen von Satzenden verhindert. Der Entitätenerkenner, der für diesen Zweck geschrieben wurde, nutzt triviale Ansätze um dieses Problem anzugehen. Die Entitäten-Liste wurde von dem vorhandenen *broccoli* übernommen. Sie wurde aus der *Freebase* Ontologie extrahiert. Sie beinhaltet eindeutigen Entitäts Namen, Klassen der Entität,

[1] `http://nlp.stanford.edu/software/tagger.shtml`

Beliebtheitswert und Aliase. Die Klasse einer Entität ist grob gesagt die Kategorie. Beispielsweise hat die Entität *broccoli* die Klasse *plant*. Der Beliebtheitswert gibt an, wie oft diese Entität benutzt wird.
Zum Aufspüren der Entitäten, wird über die Wörter iteriert und festgestellt, ob das Wort oder eine Kombination dieses Wortes mit vorherigen mit einem Alias der Entitäten-Liste übereinstimmt. Falls dies auf mehrere zutrifft, wird die Entität gewählt, die den höheren Beliebtheitswert hat. Nachdem man sich für diese Entität entschieden hat, wird geprüft, ob sich diese mit anderen Entitäten überschneidet. Falls dies der Fall ist, wird die Entität gewählt, die länger ist. Im Titel wurde die Entitätenerkennung mit selbigen Annahmen angewandt. Dies wurde von dem bestehenden *broccoli* bisher nur bei Wortgleichheit erkannt.
Wenn man Entitäten gefunden hat, ist es wichtig auch indirekte Vorkommen dieser zu erkennen. Eine weibliche Person wird beispielsweise oft in Texten nur noch mit *she* referenziert. Hierzu wird der bestehende Test auf Pronomen verwendet. Dieser nutzt die zuletzt erkannte Entität und überprüft, ob das Geschlecht mit dem Pronomen übereinstimmt.

3.2.2.5 Writer

Der Writer wurde aus der bestehenden *broccoli* Pipeline übernommen. Er schreibt alle Wörter und erkannte Entitäten, sowie entsprechende Kontext Abschnitte in Dateien. Diese werden dann zum Erstellen des Indexes benutzt.

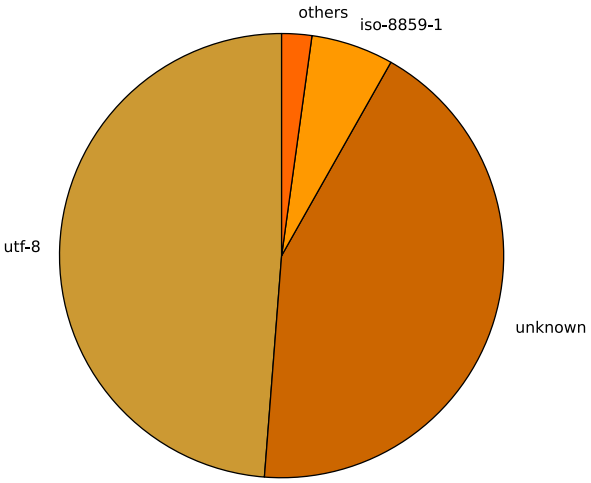
3.2.3 Index Generierung

Zum Erstellen des Indexes wurde der vorhandene IndexBuilder verwendet. Die Details sind in [BB13] genauer beschrieben. Dieser ließt die Wörter und Entitäten Liste ein, und verarbeitet das zusammenhänge zwischen Entitäten und einzelnen Kontexten effizient abbildet. Außerdem wird dieser danach Komprimiert um Speicherplatz zu sparen. Dies resultiert in einem binären invertierten Index. Dies ist eine sehr stark vereinfachte Beschreibung.

4 Evaluation

4.1 Evaluation der Datenmenge

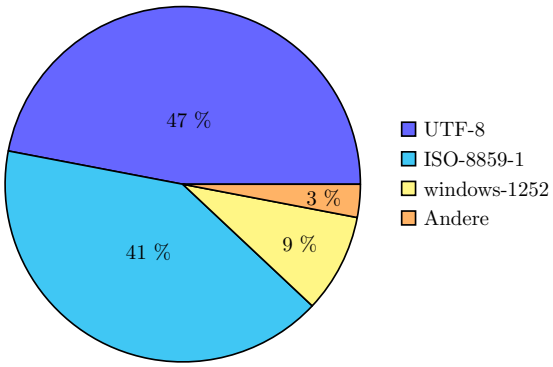
Es wird nun auf die genaue Zusammensetzung der Daten eingegangen. Hierzu wird erläutert, wie diese Ergebnisse entstanden sind, was diese Aussagen, wie sie verbessert wurden.
Zum Erstellen der Verteilungen wurde der Inhalt nicht berücksichtigt. Das bedeutet, dass jegliche Quelle genutzt wurde, nicht nur HTML und reine Texte. In den dadurch entstandenen Verteilungen, kann man einen klaren Kontrast zu dem von Spiegler analysierten Daten hat [Spi13]. Man sieht, das die Wahl der Seiten die Codierung *unknown* sehr stark verdrängt hat. Dies stimmt mit der Analyse Spieglers überein, da er untersucht hat, welche TLD's die meisten *unknown* Codierungen enthalten, all jene sind in dem benutzten Korpus nicht enthalten.



Character encoding	Abs. freq.	Rel. freq.
utf-8	1,866,333,314	0.4874
unknown	1,647,477,248	0.4303
iso-8859-1	229,671,038	0.0600
windows-1251	26,798,707	0.0070
iso-8859-2	10,088,397	0.0026
iso-8859-15	8,605,343	0.0022
windows-1256	5,454,253	0.0014
shift-jis	5,289,261	0.0014
windows-1252	5,173,227	0.0014
euc-jp	4,201,400	0.0011
others	18,074,100	0.0047

Abbildung 4.1: Die Verteilung von Codierungen im gesamten CommonCrawl Korpus

Als nächstes wird die Verteilung der Typen im Korpus untersucht. Hier wird nochmals angemerkt, dass in dem benutzten Korpus deutlich mehr PDF Dokumente enthalten sind Abbildung 4.3. Dies ist durch die Hinzunahme von *.edu* Seiten aufgetreten, wie man in Abbildung 4.4 erkennen kann. An dieser Stelle wurde damit experimentiert Text aus den PDF-Dokumenten zu extrahieren. Dafür wurde die Kommandozeilen Funktionalität *pdftotext* bzw. *pdftohtml* von *xpdf* aus dem C++ Programm



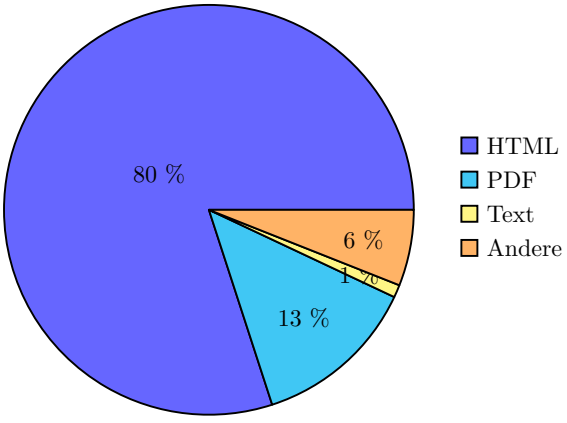
(a)

Anzahl Seiten	Codierung
4 417 301	Gesamtzahl
2 107 314	UTF-8
1 837 911	ISO-8859-1
422 240	windows-1252
9 941	ISO-8859-2
7 746	US-ASCII
7 691	ISO-8859-15
5 255	Big5
5 017	macintosh
3 764	GB2312

(b)

Abbildung 4.2: Die Verteilung der Codierungen in der Datenmenge
*In **a)** sieht man die prozentuale Verteilung der Codierungen und in **b)** wird die exakte Menge der häufigsten Codierungen angegeben.*

aufgerufen. Dies extrahierte aber viele Formatierungen und war durch den *system()* Aufruf nicht gut anwendbar. Aufgrund von Zeitknappheit, wurde die Bearbeitung dieses Problems nicht weiter verfolgt. Für zukünftige Arbeiten ist es denkbar, die Texte aus PDF's mit in den Korpus zu integrieren. Dies ist vor allem dann sinnvoll, wenn diese in einer ähnlichen Häufigkeit auftreten, wie es hier der Fall war. Zum erzeugen dieser Daten, wurde ein TSV-File erzeugt, welches Domain und Format und Codierung enthalten hat. Danach wurde dieses durch Sortieren, Gruppieren und Abzählen der Vorkommen zu den hier angegebenen Verteilungen.



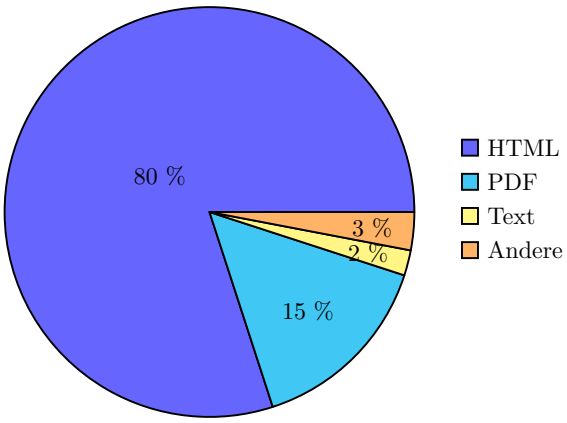
(a)

Anzahl Seiten	Typ der Ressource
4 417 301	Gesamtzahl
3 555 951	text/html
578 646	application/pdf
62 362	text/plain
12 577	text/xml

(b)

Abbildung 4.3: Die Verteilung der Typen in der rohen Datenmenge.

Bei der Analyse der Wörter in der Wörter Datei, welches von der UIMA-Pipeline

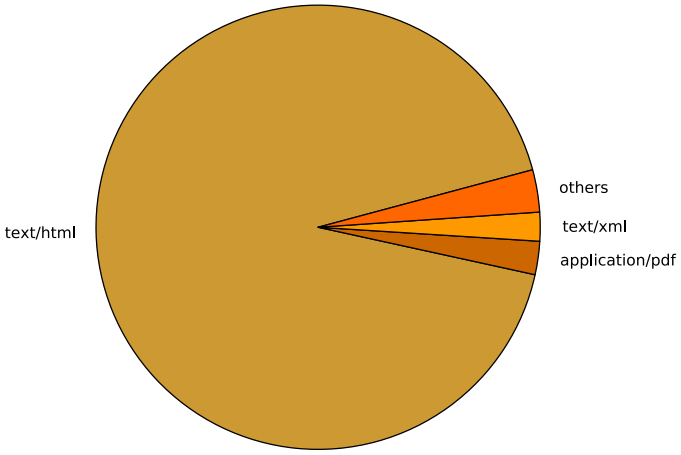


(a)

Anzahl Seiten	Typ der Ressource
3 838 819	Gesamtzahl
3 078 685	text/html
577 660	application/pdf
62 262	text/plain
12 527	text/xml

(b)

Abbildung 4.4: Die Verteilung der Typen in der rohen Datenmenge der TLD **.edu**.



Media type	Abs. freq.	Rel. freq.
text/html	3,532,930,141	0.9227
application/pdf	92,710,175	0.0242
text/xml	80,184,383	0.0209
text/css	22,872,511	0.006
application/x-javascript	21,198,040	0.0055
image/jpeg	14,116,839	0.0037
application/javascript	11,548,630	0.003
text/plain	10,713,438	0.0028
application/msword	6,648,861	0.0017
application/xml	4,999,123	0.0013
application/rss+xml	4,200,583	0.0011

Abbildung 4.5: Die Verteilung von Ressource-Typen im gesamten CommonCrawl Korpus

erstellt wird. War es vor allem interessant nach langen Wörtern zu suchen, da vermutet wurde, dass diese durch Codierungs Fehler und fehlende Whitespace entstehen können. Die längsten Wörter waren jedoch ASCII-Arts, das Ergebnis der Potenz 1907^{1151}, welches aus 3776 Ziffern bestand, DNA-Sequenzen und ein Ausdruck von Freude einer Person, welcher 'yes' ausdrückt, sich aber über mehrere hundert Zeichen erstreckt. Bei dieser Analyse sind auch nicht englisch sprachige Texte aufgetaucht, die keine Leerzeichen zur Worttrennung nutzen (z.B. Japanisch). Da in dieser Arbeit aber bloß englische Suchanfragen gestellt werden, ist auf dies nicht weiter eingegangen worden.

4.2 Analyse und Wichtigkeit der einzelnen Schritte

Die nachstehende Tabelle beschreibt die Architektur, auf der diese Daten entstanden sind.

Beschreibung	Wert
CPU(s):	24
Vendor ID:	GenuineIntel
CPU MHz:	2527.175

Tabelle 4.1: Verwendete Architekturdaten.

Es wird im folgenden, immer darauf eingegangen, wie die Zeiten beobachtet wurden, und wie sich diese Komponente auf die weiteren Schritte auswirkt. Im genauen handelt es sich bei diesen um die Daten von *Metropolis*, einem für die Arbeit bereitgestellten Servers.
Nun wird Schrittweise jede Funktionalität angesprochen, wie sich diese auf folgende Schritte auswirkt, und die Zeiten, die für diese beobachtet wurden.

4.2.1 Herunterladen

Für das Programm, das zum Herunterladen des Korpus und das entfernen der Whitespace verantwortlich war, wurden 80 Threads verwendet. Dies verringert die benötige Zeit zum Herunterladen, bringt jedoch den Nachteil, dass bei Vorhandensein der Daten auf der Festplatte das Programm langsamer wird. Das Herunterladen wird durch die Parallelisierung beschleunigt, da viele TCP-Connections aufgebaut werden und deshalb *TCP Slow-Start* einen großen Faktor spielt. Dies wird durch die geringe Größe der einzelnen Dateien verstärkt.
Wie oben erwähnt, werden die HTML Seiten mit dem Hash ihrer URL gespeichert. Beim Vergrößern des Korpus sollte in Betracht gezogen werden, das Hash Kollisionen auftreten können. Bei der verwendeten Korpus Größe und der Verwendung von einem 8-Byte Integer sollte eine Kollision unwahrscheinlich sein, aber ist nicht unmöglich.
Wenn die Daten auf der Festplatte vorliegen, dauert das extrahieren des Korpus rund 3 Stunden. Dies entspricht rund 19ms pro Seite, wenn man von einer gleichmäßigen Verteilung ausgeht, und die Parallelisierung naiv verrechnet. Hier mit einbegriffen ist Datei Einlesen, Selektion der Formate, Dekomprimieren, Entfernen der *Whitespace* und Schreiben in die gemeinsame Datei. Als Ausgabe entsteht eine 82GB große Datei mit 3 136 954 Seiten (> 120GB ohne Selektion der Formate). Diese Zahl ist kleiner als die Anzahl aller html Seiten im gewählten Korpus, da auch darauf geachtet wurde, das die Seiten einen Titel-Tag angeben. Leider kommt es durchaus vor, das Seiten ein Titel-Tag angeben diesen aber leer lassen.

4.2.2 UIMA-Pipeline

Es wird nun die Laufzeit und Wichtigkeit der einzelnen Komponenten untersucht. Zudem wird boilerpipe mit dem naiven Ansatz der HTML Extraktion (Entfernen aller HTML-Tags) verglichen und es wird die Qualität der Entitätenerkennung betrachtet.

4.2.2.1 Boilerpipe und Parser

In Abbildung 4.6 sieht man den Aufbau einer HTML Nachrichten Seite. Diese besteht zu einem großen Teil aus Formatierungen, Werbung, Abstimmungen, und Texten. Das naive Extrahieren des Textes wird durch entfernen der HTML-Tags in Abbildung 4.8 gezeigt. Zum Kontrast dazu wurde das Ergebnis von boilerpipe in Abbildung 4.7 dargestellt. Wie man erkennen kann, hat boilerpipe beispielsweise die Navigation Vollständig entfernt. Dies ist sehr hilfreich für das Erstellen des Indexes, da die Navigation der Seite nicht für die Suche von Inhalten relevant ist. Boilerpipe bietet eine Seite an, auf der dies für andere Beispiele gezeigt werden kann.[1] Hier ist ein weiteres Beispiel verlinkt.
Link: ArticleExtraction von einer Reuters.com Seite[2]
Link: KeepEverythingExtraction(naiv) der selben Seite[3]

Die Laufzeit von boilerpipe wird gemeinsam mit dem Einlesen angegeben, da diese in der selben UIMA-Komponente sind. In der oben angegebenen Architektur verbrachten die 10 Threads zusammen rund drei Stunden und 45 Minuten in dieser Komponente. Dies entspricht einem reellen Zeitaufwand von 22 Minuten. Diese Zeiten wurden beim Einlesen der 82GB Datei beobachtet.

4.2.2.2 Tokenizer

In der Tokenizer Komponente der Pipeline verweilten die Threads zusammen für fast 30 Minuten. Dies entspricht rund 3 Minuten bei paralleler Ausführung. Wenn man nun die Feststellung nutzt, dass weniger als ein fünftel der eingelesenen Daten hier ankommen, wird eine Verarbeittungsrate von $95MB/s$ ermittelt.

4.2.2.3 Entitätenerkenner

Der Entitätenerkenner liest zu beginn die Entitäten-Liste ein, welche rund 60 GB an RAM verbraucht, wenn diese in zu JAVA Hash-Map verarbeitet wird. Hier wurde damit experimentiert einen Trie statt eine Hash-Map zu nutzen. Dies wurde jedoch

[1] BoilerpipeWebApp: `http://boilerpipe-web.appspot.com/`
[2] `http://preview.tinyurl.com/onb5p4g`
[3] `http://preview.tinyurl.com/p8x9p9w`

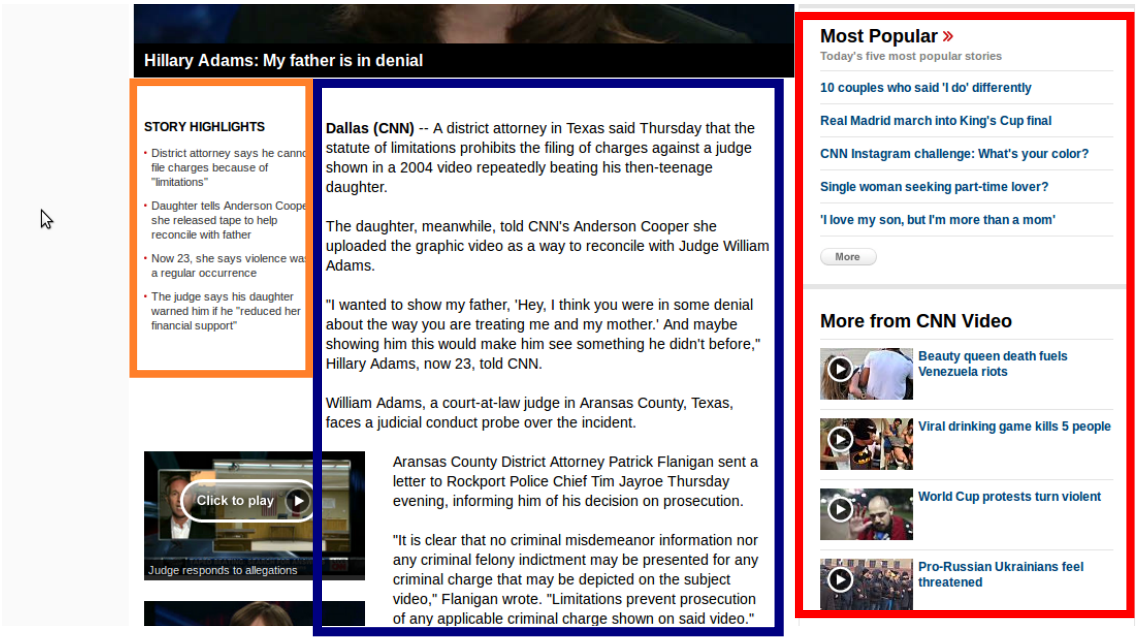


Abbildung 4.6: Eine Nachrichten Seite
In blau sieht man den Artikel der Seite, der von vielen weiteren Elementen umgeben ist. Auf der rechten Seite(rot) ist eine Seiten-spezifische Werbung markiert, und auf der linken Seite(orange) stehen knappe Informationen zum Artikel(Seiten-spezifisch).

verworfen, da die naive Implementation zu viel Arbeitsspeicher verbrauchte. Ein Trie oder eine andere sortierte Datenstrukturen, würde das Prüfen auf einen Präfix ermöglichen. Dies könnte die Ausführung des Entitätenerkenners deutlich beschleunigen, da weniger Verbindungen von Wörtern bearbeitet werden müssen.
Der neue Entitätenerkenner erkennt im Vergleich zu dem für Wikipedia verwendeten, deutlich mehr Entitäten. Das Verhältnis von Wörtern zu Entitäten entspricht $1 : 4$, welches darauf schließen lässt, dass viele der erkannten Entitäten Fehler seid. Um dies zu analysieren, wurde die Wörter Liste an zufälligen Stellen betrachtet. Hierbei wurde auch festgestellt, dass trotz den vorher genannten Selektierungen der Daten viele fehlerhafte Wörter enthalten sind.
Um vor allem unerwünschte Sprachen zu entfernen, ist es denkbar Ansätze von [Kel11] aufzugreifen. Dieser Arbeitet auf einem älteren Datensatz sollte aber auch auf dem Datensatz von 2012 anwendbar sein. Hierfür war jedoch nicht genügend Zeit.

4.3 Probleme beim Index erstellen

Das Erstellen des Indexes für den Korpus der nur aus Nachrichtenseiten bestand, dauerte rund eine Stunde. Als der Korpus um die .edu Seiten erweitert wurde, konnte der IndexBuilder lange nicht mehr benutzt werden. Es stellte sich heraus, dass ein

NUL-Byte in den Daten diesen Fehler verursachte. Deshalb muss noch ein Mal auf die Wichtigkeit der Qualität der Datenmenge aufmerksam gemacht werden.

DA: Statute of limitations prevents charges in judge's beating of daughter
By Tracy Sabo, Ashley Hayes and Moni Basu, CNN
November 4, 2011 -- Updated 1232 GMT (2032 HKT)
STORY HIGHLIGHTS
District attorney says he cannot file charges because of "limitations"
Daughter tells Anderson Cooper she released tape to help reconcile with father
Now 23, she says violence was a regular occurrence
The judge says his daughter warned him if he "reduced her financial support"
Dallas (CNN) -- A district attorney in Texas said Thursday that the statute of limitations prohibits the filing of charges against a judge shown in a 2004 video repeatedly beating his then-teenage daughter.
The daughter, meanwhile, told CNN's Anderson Cooper she uploaded the graphic video as a way to reconcile with Judge William Adams.
"I wanted to show my father, 'Hey, I think you were in some denial about the way you are treating me and my mother.' And maybe showing him this would make him see something he didn't before," Hillary Adams, now 23, told CNN.
William Adams, a court-at-law judge in Aransas County, Texas, faces a judicial conduct probe over the incident.
Judge responds to allegations...

Abbildung 4.7: Der Anfang des Textes nach boilerpipes *ArticleExtractor*
Die Farbe entspricht dem Inhalt des zugehörigen Kasten in Abbildung 4.6.

Entertainment
Tech
Travel
iReport
SHARE THIS
Print
Email
More sharing
Reddit
StumbleUpon
Delicious
DA: Statute of limitations prevents charges in judge's beating of daughter
See show times »
By Tracy Sabo, Ashley Hayes and Moni Basu, CNN
November 4, 2011 -- Updated 1232 GMT (2032 HKT)
STORY HIGHLIGHTS
District attorney says he cannot file charges because of "limitations"
Daughter tells Anderson Cooper she released tape to help reconcile with father
Now 23, she says violence was a regular occurrence
The judge says his daughter warned him if he "reduced her financial support"

Abbildung 4.8: Der Anfang des Textes nach HTML-Tag entfernen
Die Farbe entspricht dem Inhalt des zugehörigen Kasten in Abbildung 4.6. Man erkennt, das die ersten Zeilen der Navigation entsprechen und keinen semantischen Inhalt beisteuern. Nach diesen tritt dann auch der Artikel auf, gefolgt von weiteren Navigationen.(nicht sichtbar in der Abbildung)

5 Fazit

Diese Arbeit zeigt, dass das Internet ein sehr chaotischer Ort zum Auffinden von Daten ist. Deshalb ist es notwendig, sich bereits bei der Selektion des Korpus damit vertraut zu machen, welche Fehlerquellen in den gewünschten Dokumenten enthalten sind. Um Textanalysen auf diesen zu implementieren, muss dafür gesorgt werden, dass die Texte von den umliegenden Störquellen möglichst gut befreit werden.
In dieser Arbeit wurde dieses Problem durch die Verwendung von boilerpipe und einer einheitlichen Codierung gelöst. Außerdem wurde beschrieben, wie man weitere Ansätze nutzen kann, um beispielsweise nicht englische Texte zu ignorieren [Kel11]. Es wurde auch darauf eingegangen, dass viele Annahmen zum Erkennen von Entitäten, die gute Ergebnisse auf Texten in Wikipedia produzierten, für allgemeine Webseiten zu keinen guten Ergebnissen führen.

Literaturverzeichnis

[BB13] Bast, Hannah ; Buchhold, Björn: An index for efficient semantic full-text search. In: *Proceedings of the 22nd ACM international conference on Conference on information & knowledge management* ACM, 2013, S. 369–378

[BBBH12] Bast, Hannah ; Bäurle, Florian ; Buchhold, Björn ; Haussmann, Elmar: Broccoli: Semantic full-text search at your fingertips. In: *arXiv preprint arXiv:1207.2615* (2012)

[BU77] Bayer, Rudolf ; Unterauer, Karl: Prefix B-trees. In: *ACM Transactions on Database Systems (TODS)* 2 (1977), Nr. 1, S. 11–26

[Kel11] Kelcey, Mat: *Tokenizing the visible texts of common crawl.* `http://matpalm.com/blog/2011/12/10/common_crawl_visible_text/`. Version: 2011

[KFN10] Kohlschütter, Christian ; Fankhauser, Peter ; Nejdl, Wolfgang: Boilerplate detection using shallow text features. In: *Proceedings of the third ACM international conference on Web search and data mining* ACM, 2010, S. 441–450

[Spi13] Spiegler, Sebastian: *Statistics of the Common Crawl Corpus 2012.* `https://docs.google.com/file/d/1_9698uglerxB9nAglvaHkEgU-iZNm1TvVGuCW7245-WGvZq47teNpb_uL5N9/edit`. Version: 2013

[Ste12] Stephens, Chris: *About the Data Set.* `https://commoncrawl.atlassian.net/wiki/display/CRWL/About+the+Data+Set`. Version: 2012

[TKMS03] Toutanova, Kristina ; Klein, Dan ; Manning, Christopher D. ; Singer, Yoram: Feature-rich part-of-speech tagging with a cyclic dependency network. In: *Proceedings of the 2003 Conference of the North American Chapter of the Association for Computational Linguistics on Human Language Technology-Volume 1* Association for Computational Linguistics, 2003, S. 173–180

[tri13] triv, Scott R.: *Common Crawl URL Index.* `https://github.com/trivio/common_crawl_index`. Version: 2013