UNIVERSITÄT LEIPZIG

Digital Text Forensics Search

Information Retrieval
WS 2017/2018

Betreurer: Martin Potthast
martin.potthast@uni-leipzig.de

Autor: David Drost
dd42cequ@studserv.uni-leipzig.de
Matrikelnummer:

Autor: Edward Kupfer
ek96foje@studserv.uni-leipzig.de
Matrikelnummer:

Autor: Hendrik Sawade
hs34byhe@studserv.uni-leipzig.de
Matrikelnummer: 3745956

Autor: Tobias Wenzel
tw54byka@studserv.uni-leipzig.de
Matrikelnummer: 3733301

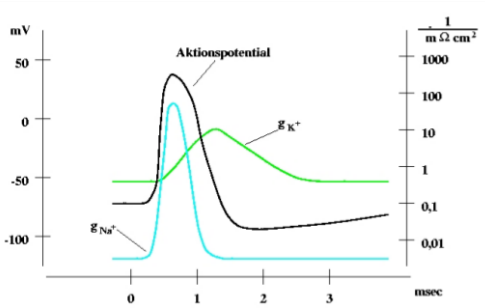
Abgabedatum: Leipzig, den 5. März 2018

Inhaltsverzeichnis

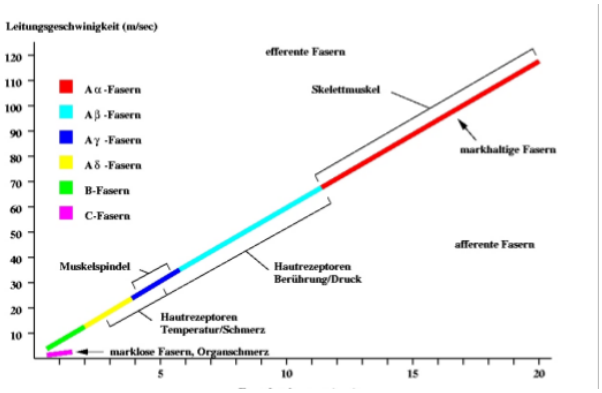
1 Einleitung

Hier schreibt man dann was. Und so . Offenbar sieht das nicht so schön aus.

kommentiere ihr Sachen.



(a) Pyramidenzelle



(b) Neocortex

Abbildung 1.1: Links: AP. Rechts: Leitungsgeschwindikeit von verschiedenen Fasern.

2 Vorverarbeitung

Im folgenden Abschnitt wird beschrieben, wie der betrachtete Datensatz in eine sinnvolle indizierbare Form gebracht wird. Zunächst wird in Abschnitt 2.1 ein Überblick über die Auswahl der Fremdsoftware gegeben. Anschließend werden in Abschnitt 2.2 die einzelnen Schritte der Verarbeitung betrachtet.

2.1 Auswahl

Es existieren einige Toolboxen, die die Extraktion von Text und Meta- Informationen vornehmen können. Aufgrund der Wahl der Programmiersprache Java (siehe Kapitel 1) wurde dazu das Apache Tika™ Toolkit in Betracht gezogen. Es bietet einfach gestaltete Schnittstellen zu Open Source Libraries um je nach Datenformat eine geeignete Datenextraktion vorzunehmen. Neben PDFs lassen sich auch DOCs, Power Point Präsentationen und Bilder (OCR) ansprechen. Die Komponente der PDF-Extraktion, Apache PDFBox®, lässt sich auch außerhalb von Tika nutzen. Die Paper-Auswahl besteht überwiegend aus PDFs[1], deswegen wurde Apache PDFBox® ausgewählt, um einen-Overhead zu vermeiden. Bei der Zunahme von weiteren Formaten lässt sich der Code leicht mit entsprechenden Apache Tika™- Methoden austauschen.

Bei der Untersuchung des Datensatzes wird deutlich, dass nur ein geringer Anteil an Papern korrekte Meta-Informationen angegeben werden. Um Titel aus den Texten zu extrahieren wurde deswegen Docear PdfDataExtractor[2] eingesetzt. Die Software sucht, neben weitern Heuristiken, auf der ersten Seite des Artikels nach langen zusammenhängenden Wortsequenzen.

1 1595 PDFs, 11 Docs und 1 HTML
2 Link: `https://www.docear.org/tag/pdf-title-extraction/`

2.2 Komponenten

In den folgenden Abschnitten werden die einzelnen Schritte der Vorverarbeitung kurz dargelegt. Dabei liegt der Fokus auf der Extraktion bzw. die Gewinnung der Meta-Daten.

2.2.1 Meta-Daten-Extraktion

Sind die Daten extrahiert, so wird zunächst überprüft, ob der Titel der Meta-Daten unzulässige Zeichen enthält oder eine unzulässige Länge hat. Ist dies nicht der Fall, wird mit Docear PdfDataExtractor versucht, im Text einen validen Titel zu finden. Valide Titel werden an eine Schnittstelle der Digital Bibliography & Library Project (DBLP)[3] gesendet und mit der dort vorliegenden Datenbank verglichen. Die Attribute des Artikels mit der höchsten Übereinstimmung (Score) werden für den aktuellen Artikel übernommen. Es besteht die Möglichkeit, den Datensatz als XML-Datei lokal zu durchsuchen und mit weiteren Daten anzureichern. In Unterabschnitt 2.2.2 soll weiter darauf eingegangen werden. Konnte nach wie vor kein valider Titel entnommen werden, wird eine Zeichenkette festgelegter Länge als Heuristik für den Title angenommen. In diesem Fall wird eine Named Entity Recognition auf den ersten Wörtern des Artikel-Textes gefahren, um mögliche Autoren zu finden. Hierzu wird Apache OpenNLP verwendet[4].

Aufgrund der vergleichsweise geringen Anzahl an Papern werden die Artikel-Daten einzeln als XML-Files gespeichert. Das erleichtert die Überprüfung während der Entwicklungsphase. Das Haupt-Element article enthält die Elemente

- metaData mit title, authors, publicationDate, refCount und
- textElements mit abstract und fullText.

Um die von der ASCII Codierung abweichende Zeichen korrekt darzustellen, werden die Text-Elemente als nicht interpretierte Zeichen (CDATA) gespeichert. Zusätzlichen werden fileName, filePath sowie parseTime festgehalten, die in der weiteren Verarbeitung benötigt werden.

3 Link zu DBLP: `http://dblp.uni-trier.de/`
4 Link zu OpenNLP: `https://opennlp.apache.org/`

2.2.2 Heuristische Titel Suche

Die extrahierten Daten zeigen auch nach den in Unterabschnitt 2.2.1 beschriebenen Schritten ein nicht zufriedenstellendes Ergebnis. Im folgenden soll eine heuristische Titel Suche vorgestellt werden[5], die den Text mit Attributen einer vorliegenden Meta-Daten Kollektion vergleicht. Der Algorithmus folgt der Annahme, dass die Meta-Daten mit der höchsten Übereinstimmung die korrekten sein müssen, wenn diese als Vergleichsdaten vorliegen. Als Daten-Quelle wurde zum einen eine Auswahl an Papern des DBLP, die in relevanten Journalen erscheinen[6] und eine bereits vorhandene Auswertung von Zitations-Analysen[7] zusammengefasst. Der Algorithmus durchläuft je extrahiertem Artikel die folgende Prozedur:

Eine vorgegebene Anzahl an Wörtern des Artikels wird als zu vergleichender Text gespeichert. Während der Stax-Parser über die Meta-Daten-Kollektion läuft, werden die Elemente als aktuelles Artikel-Objekt gespeichert. Es wird ein Score berechnet, der eine mögliche Übereinstimmung anzeigen soll. Enthält der entsprechende Text Wörter der Attribute Titel, Autoren oder Publikations-Zeitpunkt, erhält der Artikel Punkte. Genaue Übereinstimmung des Titels bzw. der Autoren erhalten zusätzliche Boni, die relativ zur Länge der Attribute berechnet werden[8]. Der Artikel mit der höchsten Punktzahl wird akzeptiert. Da nicht garantiert werden kann, dass die korrekte Artikel Daten gefunden werden, muss der Score einen Schwellwert überschreiten.

2.2.3 Referenz-Analyse

5 Vorschlag: Martin Potthast.
6 → auf Tabelle verweisen.
7 wie zitiere ich denn den Herren?
8 Titel: 50, Autoren: 30

3 Indexierung

4 Backend

4.1 Auswahl eines Backend-Frameworks

Zuerst wurde für die Programmierung einer Webanwendung, um besser abstrahieren und Programmier-Paradigmen umsetzen zu können, ein geeignetes Framework gesucht. Hierfür wurden zahlreiche Frameworks mit der Unterstützung von Dependency Injection (DI) und Aspect-Oriented Programming (AOP) untersucht. Aufgrund der Auswahl der Programmiersprache Java für die Umsetzung der Anwendung schränkte sich die Anzahl der Frameworks ein. Die Recherche ergab folgende vier Frameworks:

Frameworks	Sprache	Eigenschaft
PicoContainer	Java	DI, AOP, schlank
HiveMid	Java	DI, AOP-ähnliches Feature, IoC Container
Google Guice	Java	DI, AOP, IoC Container, Annotations, Generics, modular
Spring	Java	DI, AOP, IoC Container, Annotations, Generics, modular

Tabelle 4.1: DI-Frameworks

HiveMind und Guice bieten leichter verständliche Programmierungstechniken sowie einen prägnanteren und lesbaren Code gegenüber Spring. HiveMind fokussiert sich auf das Verbinden von Services. Seine Konfiguration erfolgt über eine XML-Datei oder eine eigene Definitions-Sprache. Hierdurch ist HiveMind ein kleiner und simpel gestalteter DI-Container. Guice hingegen unterstützt die neuen Features, wie Annotations und Generics die ab Java 1.5 zur Verfügung stehen. Dies hilft dabei, eine weitgehend aufgeräumte und einfache Konfiguration zu ermöglichen. HiveMind bietet zusätzlich die Möglichkeit, mit AOP zu arbeiten. Guice und Spring bieten sehr ähnliche Ansätze und kommen mit vielen Anforderungen, die Unternehmenssoftware erfüllen müssen, zurecht. Spring‘s Modularität bietet dabei einen größeren Vorteil. Die Module können in Spring, je nachdem welche Module der Entwickler benötigt, ohne viel Aufwand hinzugefügt werden. Guice ist durch

die geringere Komplexität zwar leichter zu verstehen und insgesamt kleiner als Spring. Dieser Punkt kann aber vernachlässigt werden, da durch die Modularität von Spring das Framework nach den Voraussetzungen des Entwicklers zusammengestellt werden kann. Schlussendlich wurde sich für das Spring-Framework entschieden, da die Flexibilität und Modularität von Spring die Entwicklung von Anwendungen stark vereinfacht und daher die beste Wahl darstellt.

4.2 Auswahl einer Datenbank

Als nächstes wurde für die spätere Speicherung der Interaktion zwischen Backend, Suchergebnissen und User, also das User-Feedback eine geeignetes eingebettetes Datenbanksystem gesucht. Die Recherche ergab folgende Datenbanken:

Datenbank	Sprache	Eigenschaft
SQLite	C	SQL-92-Standard, Transaktionen, Unterabfragen (Subselects), Sichten (Views), Trigger und benutzerdefinierte Funktionen, direkte Integrierung in Anwendungen, In-Memory-DB
H2	Java	Schnell, Referenzielle Integrität, Transaktionen, Clustering, Datenkompression, Verschlüsselung und SSL ,kann direkt in Java-Anwendungen eingebettet oder als Server betrieben werden, Direkte Unterstützung in Spring, In-Memory-DB
Apache Cassandra	Java	Spaltenorientierte NoSQL-Datenbank, Für sehr große strukturierte Datenbanken, Hohe Skalierbarkeit und Ausfallsicherheit bei großen, verteilten Systemen

Tabelle 4.2: Datenbanken

SQLite bietet einen leiten einstieg in die SQL-Welt. Dabei bietet es den größten Teil des SQL-92-Standards und kann Transaktionen, Unterabfragen und viele weitere Funktionen. Außerdem ist es eine In-Memory-DB. Leider unterstützt Spring Boot diese Datenbank nicht von Haus aus und es müssten Anpassungen in der Konfiguration vorgenommen werden. Apache Cassandra ist eine Spaltenorientierte NoSQL-Datenbank und ist für große strukturierte Daten, hohe Skalierbarkeit und Ausfallsicherheit ausgelegt. Da diese Datenbank für große Datenmengen ausgelegt ist, ist diese Datenbank für dieses Backend zu groß ausgelegt. H2 ist eine In-Memory-DB, welche schnell ist, Referenzielle Integrität,

Transaktionen, Clustering, Datenkompression unterstützt. Außerdem kann Spring Boot mit dieser Datenbank ohne besondere Maßnahmen verwendet und in diese Anwendung integriert werden. Daher wurde sich Schlussendlich für die H2 Datenbank entschieden.

4.3 Erstellung des Backends

Nach der Auswahl der Backendtechnologien wurde die Grundarchitektur des Backendes konzipiert und implementiert.

4.3.1 Kommunikation mit der Datenbank

Zunächst wurden Datenmodels, wie die `Query` oder`LoggingDocument` erstellt. Hieraus werden später die Tabellen der Datenbank generiert. Um mit der Datenbank kommunizieren zu können, gehört die Erstellung von DAOs dazu, welche die Kommunikationsschnittstelle bilden. Ein DAO generiert eine SQL Query und übermittelt diese an die Datenbank.

```
1 public interface LoggingDocDao extends JpaRepository<
    LoggingDocument, Long> {
2 LoggingDocument findByDocId(Long docId);
3 }
```

Beispiele hierfür sind die Speicherung und Abrufen von LoggingDocument Daten, welche ein Teil des User-Feedbacks darstellen. In dem zu betrachtenden Code Ausschnitt sieht man wie mit Hilfe von Spring die SQL Qyery findByDocId von einem LoggingDocument geniert wird. Dies findet über den Namen eines Interfaces statt. Die einzelnen Komponenten, welche implementiert wurden, kommunizieren nicht direkt über die DAOs mit der Datenbank, sondern dazwischen befindet sich noch ein Abstrahiertes Service Interface. Dadurch ist eine lose Kopplung zwischen den Komponenten, DAO und Datenbank möglich. Dies dient dazu, dass die Datenbank ohne große Änderungen in den Implementierungen austauschbar ist. Es ist dadurch nur notwendig, in den Konfigurationen und eventuell in den DAOs Änderungen vorzunehmen. Ein Beispiel für einen Service ist der `UserLoggingService`.

```
1 public class LoggingDocServiceImpl implements
    LoggingDocService {
```

```
2   public LoggingDocument findbyId(Long id) {
3   return loggingDocDao.findOne(id);
4   }}
```

In der Implantation des Interfaces wird nun das DAO aufgerufen. Wie hier im Beispiel zu sehen die Methode findbyId.

4.3.2 Controller

Die nächste Schritt ist es sogenannte Controller zu erstellen. Diese bilden eine wichtige Schnittstelle für die Kommunikation mit dem Frontend und dem Backend. Controller reagieren auf HTTP-Requests, welche von dem Frontend oder anderen Clients gesendet werden. Die Aufgabe ist es für die bestimmten Ressource-URLs bestimmte Ereignisse auszuführen. Ein Beispiel hierfür ist die Suchanfrage der Search Zeile im Frontend auszuwerten und die Suchergebnisse zurück zu senden.

```
1  @RequestMapping(method = RequestMethod.GET, path = "/")
2  public ModelAndView searchPage(
3  @RequestParam(defaultValue = "")
4  String query) {
5  ModelAndView modelAndView = new ModelAndView("search");
6  ...
7  List<ScoreDoc> list = querySearcher.search(query);
8  ...
9  modelAndView.addObject("searchResultPage", searchResultPage);
10 return modelAndView;}
```

In diesem Code Beispielabschnitt sieht man, dass wenn ein Request bei der Path-URL „/“ ausgelöst wird, wird die Funktion SearchPage aufgerufen und eine Suche ausgeführt. Hierfür wird der RequestParameter query ausgewertet, gesucht und anschließend die Suchergebnisse in einem modelAndView Objekt dem Frontend übergeben.

5 Evaluation und Fazit