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

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 bzw. dem höchsten 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 Ranking der Dokumente

Die gegebenen Paper können auch zur Indizierungszeit gerankt werden. Hierbei spielt die Relevanz zu einer Query noch keine Rolle. Es geht darum, welches Paper generell wichtiger ist, als ein anderes. Diese Gewichtung wird in das finale Ranking einberechnet. Es existieren diverse Faktoren nach denen die Relevanz von Papers im Bezug auf andere Papers festgemacht werden kann. Im Zuge der Bearbeitung wurden zum Einen die Anzahl der Klicks und die Verweildauer auf einem Paper einbezogen und zum Anderen die Zahl der Zitierungen in anderen Papers. Weitere Faktoren, wie bspw. Autoren, Sprache oder

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

Form wurden für das Ranking nicht hinzugezogen, könnten jedoch in weiteren Arbeiten zu dem Thema betrachtet werden.

In diesem Abschnitt wird auf die Anzahl der Zitierungen für jedes Paper eingegangen. In Anlehnung an den Begriff Pagerank wird dies hier als Paperrank bezeichnet. Webseiten nehmen Bezug auf andere Webseiten. Es ist möglich daran die Wichtigkeit einer Webseite festzumachen. Wird eine oft auf anderen Seiten erwähnt ist davon auszugehen, dass sie wichtiger ist, als eine weniger erwähnte. Bei Papers ist das ähnlich: Wird ein Paper oft zitiert so ist davon auszugehen, dass es für die Autoren von Papers eine höhere Relevanz hat, als Papers die weniger zitiert werden. Daher wurde dieser Aspekt für das Ranking der Dokumente für die Suchmaschine betrachtet. Außerdem kann als Nebeneffekt anhand des Paperranks die komplette Verteilung der Zitierungen dargestellt werden. Die Umsetzung erfolgte über ein Perlskript, es gab Probleme bei der Umsetzung mit Java. Das Skript kann Offline zur Indexing-Zeit ausgeführt werden. Es wird vom Hauptprogramm über die Javaklasse `RunScript.class` aufgerufen. Das Skript wurde für die Perl Version 5 getestet[9]. Das Skript wurde zur besseren Nachvollziehbarkeit in einzelne Schritte unterteilt auf die im Folgenden eingegangen wird. Zunächst wurden zur Feststellung einige Grunddaten aus den gegebenen wissenschaftlichen Texten benötigt. Daraus wurden im speziellen die Titel und die Quellen genutzt. Die Paper liegen dank der Vorverarbeitung in Form von Textdateien vor. Die Titel sind durch die XML-Tag <title> markiert. Es erfolgte eine Extraktion durch reguläre Ausdrücke auf den <title>-Tag und zur Sicherheit, da nicht alle Paper einen ordentlich befüllten Tag hatten auf den <Filename>-Tag. Die erste Herausforderung stellte die Extraktion der Quellen für jedes Paper dar. Diese sind nicht gesondert markiert, sondern sind im Text eingebettet. Dies wurde mit Hilfe einer Heuristik gelöst. Quellen sind typischer Weise nummeriert. Diese Nummerierung wird oftmals eingeschlossen durch eckige [] oder runde () Klammern. Daher wurde mit Hilfe regulärer Ausdrücke nach Klammern gesucht, welche ein- oder zweistellige Zahlen enthalten. War diese Bedingung erfüllt, wurde die entsprechende Zeile als Quelle extrahiert. Mit diesen Daten für Titel und Quellen wurde nun weitergearbeitet. Im nächsten Schritt werden die so entstandenen Quellen in Zeichenketten zerlegt. Für eine Länge von 30 Zeichen haben sich gute Ergebnisse eingestellt. Dies geschieht, da die Quellen nicht nur den Titel der zitierten Quelle enthalten, sondern auch Dinge wie bspw. Erscheinungsdatum, Author, Verlag, ISBN. Danach können die Zeichenketten mit den Titeln abgeglichen werden. Wenn der Titel die Zeichenkette

9 Hier bitte die Readme beachten.

beinhaltet, dann wird der Titel in den nächsten Bearbeitungsschritt übernommen. Hierbei werden beim Abgleich Sonderzeichen und Zahlen ausgenommen[10]. Daraufhin werden die Titel gezählt und sortiert. Es entsteht eine Liste mit den Papers zusammen mit der Anzahl mit der sie in den Quellen der anderen Paper vorkommen. Daraus kann nun entnommen werden, welche Paper die wichtigsten sind. Die Datei wurde in ein XML-File umgewandelt um eine reibungslose Weiterverarbeitung zu gewährleisten. Außerdem wurde die Datei, durch entfernen von unnötigen Leerzeichen etc., etwas bereinigt, um Speicherplatz einzusparen und eine höhere Effizienz zu gewährleisten. Das Ergebnis ist in Abbildung 2.1 dargestellt:

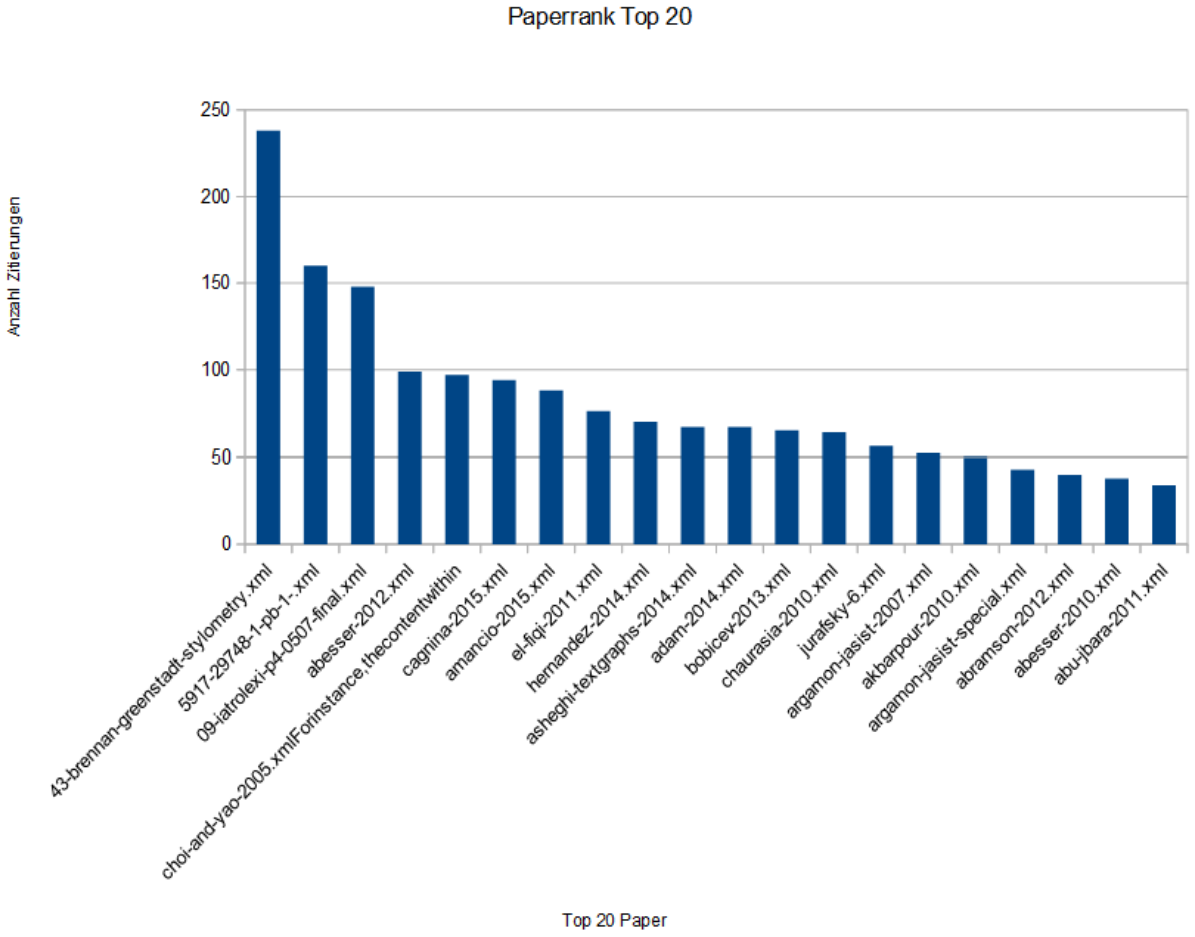

Abbildung 2.1: Anzahl der Zitierungen für die Top 20 Paper

Zum Abschluss wurden die entsprechenden gezählten Zitierungen in die Ausgangsdateien geschrieben. Sie wurden in <entry>- und <counter>-Tags verpackt, damit im Scoring

10 escaped

Schritt ein reibungsloser Zugriff möglich wird. Nun soll noch ein kurzer Ausblick gegeben werden, was nicht in das Resultat des Paperranks eingeflossen ist, aber die Ergebnisse noch verbessern könnte. Bei der Extraktion der Quellen ist sicher noch einiges an Optimierungspotenzial. Es könnten weitere Kriterien in die Auswahl einfließen, um zum einen mehr Quellen zu finden und zum anderen *Nichtquellen* zu eliminieren. Außerdem wurde nicht beachtet, welche Quelle die Zitierungen vornimmt. Generell hat sich das Vorgehen an keinem Algorithmus, wie bspw. dem Random Surfer Modell orientiert. Dies wäre auch ein interessanter Ansatz für weitere Nachforschungen. Nichtsdestotrotz entstand mit dieser Arbeit eine weitere Möglichkeit die gegebenen wissenschaftlichen Texte zu bewerten bzw. zu *ranken*. Des Weiteren ist nun eine Aussage darüber möglich, wie oft einzelne Paper in anderen Papern als Quellen herangezogen werden und wie die Verteilung über alle Texte ist.

3 Indexierung

4 Backend

4.1 Auswahl eines Backend-Frameworks

Zuerst wurde für die Programmierung einer Webanwendung ein geeignetes Framework gesucht, um Programmier-Paradigmen umzusetzen und die Architektur besser zu abstrahieren. Hierfür wurden zahlreiche Frameworks untersucht, welche Dependency Injection (DI), Inversion of Control (IoC) und Aspect-Oriented Programming (AOP) unterstützen. Aufgrund der Auswahl der Programmiersprache Java für die Umsetzung der Anwendung schränkte sich die Anzahl der Frameworks ein. Die Recherche ergab folgende drei Frameworks:

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

Tabelle 4.1: DI-Frameworks

HiveMind und Google Guice bieten gegenüber Spring Boot leichter verständliche Programmierungstechniken sowie einen prägnanteren und lesbareren Code. 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. Darüber hinaus bietet HiveMind die Möglichkeit, mit AOP zu arbeiten. Google Guice hingegen unterstützt Features wie Annotations und Generics, die ab Java 1.5 zur Verfügung stehen. Sie helfen dabei, eine weitgehend aufgeräumte und einfache Konfiguration zu ermöglichen. Google Guice und Spring Boot bieten sehr ähnliche Ansätze und kommen mit vielen Anforderungen, die Unternehmenssoftware erfüllen müssen, zurecht. Google Guice ist durch die geringere Komplexität leichter zu verstehen und insgesamt kleiner als Spring Boot. Jedoch bietet die Modularität von Spring Boot

den größeren Vorteil: Die Module können, je nachdem welche der Entwickler benötigt, ohne viel Aufwand hinzugefügt werden. Schlussendlich fiel die Entscheidung auf das Spring Boot-Framework, da dessen Flexibilität und Modularität die Entwicklung von Anwendungen stark vereinfacht und daher die beste Wahl darstellt.

4.2 Auswahl einer Datenbank

Als nächstes wurde für das spätere Speichern der Interaktion zwischen Backend, Suchergebnissen und User, genauer des User-Feedbacks 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 Integration in Anwendungen, In-Memory-Datenbank
Apache Cassandra	Java	Spaltenorientierte NoSQL-Datenbank, für sehr große strukturierte Datenbanken, hohe Skalierbarkeit und Ausfallsicherheit bei großen, verteilten Systemen
H2	Java	Schnell, Referenzielle Integrität, Transaktionen, Clustering, Datenkompression, Verschlüsselung und SSL, direkte Einbettung in Java-Anwendungen oder Betrieb als Server möglich, direkte Unterstützung in Spring Boot, In-Memory-Datenbank

Tabelle 4.2: Datenbanken

SQLite bietet einen leichten Einstieg in die Datenbanken. Dabei stellt SQLite den größten Teil des SQL-92-Standards zur Verfügung und kann Transaktionen, Unterabfragen und viele weitere Funktionen durchführen. Außerdem ist es eine In-Memory-Datenbank. Jedoch unterstützt Spring Boot diese Datenbank nicht von Haus aus und es müssten aufwendige Konfiguration vorgenommen werden.

Apache Cassandra ist eine spaltenorientierte NoSQL-Datenbank und ist für große strukturierte Daten, hohe Skalierbarkeit und Ausfallsicherheit ausgelegt. Für das vorliegende Projekt ist Apache Cassandra jedoch zu groß ausgelegt, da für das Backend mit geringeren Datenmengen gearbeitet werden soll.

H2 ist eine In-Memory-Datenbank, welche schnell ist und referenzielle Integrität, Transaktionen, Clustering sowie Datenkompression unterstützt. Außerdem kann Spring Boot mit dieser Datenbank ohne besondere Maßnahmen wie aufwändige Konfigurationen verwendet und in die vorliegende Anwendung integriert werden. Deshalb wurde entschlossen, H2 als Datenbank anzuwenden.

4.3 Erstellung des Backends

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

4.3.1 Kommunikation mit der Datenbank

Zunächst wurden Datenmodels wie `Query` oder `LoggingDocument` erstellt. Hieraus werden später die Tabellen der Datenbank generiert. Um mit der Datenbank kommunizieren zu können, werden Data Access Objects (DAO) als Kommunikationsschnittstellen erstellt. Ein DAO hat eine Anbindung zu den Spring-Boot Repositorys, welche in der Lage sind SQL-Query zu generieren und übermittelt diese an die Datenbank. Beispiele hierfür sind das Speichern und Abrufen von `LoggingDocument`-Daten, welche einen Teil des User-Feedbacks darstellen.

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

Im obigen Code-Ausschnitt wird mit Hilfe von Spring Boot die SQL-Qyery `findByDocId` aus dem `LoggingDocument` generiert. Dies findet über den Namen eines Interfaces statt. Die einzelnen Komponenten, welche implementiert wurden, kommunizieren nicht direkt über die DAOs mit der Datenbank, sondern über ein Interface. Dadurch ist eine lose Kopplung zwischen den Komponenten, DAOs und der Datenbank möglich. Damit ist die Datenbank ohne große Änderungen in den Implementierungen austauschbar. Folglich fehlen nur noch Änderungen in den Konfigurationen und eventuell in den DAOs.

```
1 public class LoggingDocServiceImpl implements ↩
    LoggingDocService {
2   public LoggingDocument findbyId(Long id) {
3   return loggingDocDao.findOne(id);
4   }}
```

Im vorliegenden Listing ist `LoggingDocService` als Beispiel für einen Service dargestellt. In der Implantation des Interfaces wird nun das DAO aufgerufen, beispielsweise die Methode `findbyId`.

4.3.2 Controller

Im nächsten Schritt wurden sogenannte Controller erstellt. Diese bilden eine wichtige Schnittstelle für die Kommunikation mit dem Frontend und Backend. Controller reagieren auf HTTP-Requests, welche von dem Frontend oder anderen Clients gesendet werden. Die Aufgabe ist es, für bestimmte Ressource-URLs spezielle Ereignisse auszuführen. Ein Beispiel hierfür ist Auswertung der Suchanfrage der Search-Zeile im Frontend und das Rücksenden der Suchergebnisse.

```
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;}
```

Im obigen Code-Beispielabschnitt ist erkennbar, dass, beim Auslösen eines Request bei der Path-URL ,,/‘‘ die Funktion `SearchPage` aufgerufen und eine Suche ausgeführt wird. Hierfür wird der Request-Parameter mit `query` ausgewertet. Die Suche erfolgt mithilfe der Komponente Lucene, welche bereits in Abschnitt 3 näher erläutert wurde. Im Anschluss werden die Suchergebnisse als `modelAndView`-Objekt dem Frontend übergeben.

4.3.3 Komponenten des Frontends

Nach dem Controller wurden die Komponenten, welche für das Frontend benötigt werden, konzipiert und im Anschluss implementiert.

Wurden die Suchen durchgeführt und die Lucene-Komponente die Suchergebnisse zurückgegeben, wird die gesamte Ergebnisliste gesplittet. Hierbei wird für die angeforderte Seite eine Subliste erstellt, in welcher nur die geforderten Suchergebnisse enthalten sind und die restlichen Ergebnisse verworfen werden. Hierdurch ist es nicht erforderlich, alle Ergebnisse zu transformieren. Dadurch arbeitet die Anwendung wesentlich schneller.

Als nächstes wurde ein Data Transfer Object (DTO) erstellt, um die relevanten Suchergebnisse, die in der Subliste enthalten sind, in das gewünschte Ausgabeformat zu überführen und in einer separaten Liste zu sammeln. Das DTO hat dabei unter anderem die Variablen Autor, Titel, Snippet oder den Redirect-Link, welcher auf die zugehörige PDF zeigt. Der Link hierfür wird mit der Methode `createLink` erzeugt. Zu diesem Zweck werden aus der docId, Query und dem Host ein Link erstellt. Als Beispiel wird folgender Link generiert:

http://localhost:8080/pdf/?docId=1&query=xyz.

Der Parameter `docId` ist dabei eine Id, welche die PDF zu dem Suchergebnis angibt und der Parameter `query` dient dem User-Feedback. Beim späteren Klick auf das Suchergebnis wird zum einen die dazugehörige PDF angezeigt und zum anderen gleichzeitig in der Datenbank das Dokument, welches mit dem einer bestimmten Query gefunden wurde, gespeichert. Damit ist es möglich, Rückschlüsse auf die Wichtigkeit des Dokuments zu ziehen. Der eben beschriebene Vorgang erfolgt mit der Methode `mapDocumentListToSearchResults`. Nach dem Erstellen der Liste der transformierten Suchergebnisse, wird sie zum Objekt `searchResultPage` hinzugefügt und um weitere Angaben ergänzt. Das ist im folgenden Code-Abschnitt ausschnittsweise zu sehen.

```
List<ScoreDoc> split = pager.split(list, currentPage);
searchResultList = querySearcher.
    mapDocumentListToSearchResults(split,query);
searchResultPage.setTotalResults(list.size());
searchResultPage.setResultsOnPage(searchResultList);
searchResultPage.setPage(currentPage);
```

Hinzu zum Beispiel kommt die Gesamtanzahl an Suchergebnissen oder auf welcher Page man sich befindet.

Die `searchResultPage` wird nun der Spring Boot Thymeleaf-Komponente übergeben und die `search.html`-Page erstellt. Hierfür wurde ein `search.html`-Template erstellt, in welchem Anweisungen zum Umgang mit den übergebenden Daten gegeben werden. Thymeleaf befolgt diese Anweisungen und wandelt sie in entsprechende HTML-Komponenten um, damit im Anschluss der der Umwandlung ein Webbrowser die Page anzeigen kann. Ein Beispiel der Anweisungen für Thymeleaf wird im folgenden Code-Abschnitt aufgezeigt.

```
<div th:each="result : ${searchResultPage.resultsOnPage}">
  <h3 class="card-title">
    <a th:href="${result.webUrl.href}"
      th:text="${result.title}">
    </a>
  </h3>
</div>
```

Das `searchResultPage`-Objekt wird aufgerufen. Daraufhin wird in einer Schleife die Liste der DTO-Objekte, die im `searchResultPage`-Objekt enthalten sind, mit Titel und `webURL` als HTML h3-Tag, welche einen Link darstellt, erstellt. Durch die Schleife wird somit für jedes Element ein eigenes HTML-Element generiert. Die Gestaltung der Oberfläche und deren Elemente wurde in separaten CSS- und Javascript-Dateien vorgenommen. Nach der Erstellung der `search.html`-Page wird die fertige Page über den Request zurückgegeben und der Webbrowser zeigt sie an.

5 Evaluation und Fazit