Masterarbeit

Strukturierte Extraktion von Text aus PDF

Fabian Schillinger

12.05.2015

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

Gutachter
Prof. Dr. Hannah Bast

Betreuer
Claudius Korzen

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.

Ort, Datum	Unterschrift

Inhaltsverzeichnis

Abstract

Wissenschaftliche Artikel werden, vor allem in der Informatik, oft online im PDF-Format angeboten. Ein großer Vorteil ist die Plattformunabhängigkeit. Zeichenkodierungen und Schriftarten sind genauso wie Grafiken, Tabellen oder Algorithmen eingebunden. Für einige Anwendungen, wie z. B. die semantische Suche, wird jedoch nur der Informationen enthaltende Fließtext benötigt. Um den Text eines PDF-Dokuments zu erhalten, bieten viele Anwendungen die Möglichkeit diesen zu extrahieren. Diese Anwendungen haben aber Schwierigkeiten mit Grafiken oder Tabellen, denn diese beinhalten auch Text, der mit extrahiert wird. Weitere Schwierigkeiten für diese Systeme sind z. B. Fußnoten oder mehrspaltige Layouts. Fußnoten können einzelne Sätze, die auf zwei Seiten oder Spalten stehen, unterbrechen. Bei mehrspaltigen Layouts kommt es oft vor, dass die Aufteilung in Spalten ignoriert werden. Um nur den Informationen enthaltenden Fließtext aus PDF-Dokumenten zu extrahieren sind noch weitere Schwierigkeiten zu überwinden. Das hier vorgestellte System schafft es, die meisten dieser Schwierigkeiten zu überwinden und aus wissenschaftlichen Artikeln den Fließtext strukturiert zu extrahieren. Dabei wird das zu untersuchende Dokument in verschiedene Blöcke aufgeteilt. Diese Blöcke werden dann klassifiziert, indem z. B. nach Schlüsselworten wie “Abstract” gesucht wird. Nach und nach werden nicht benötigte Blöcke ignoriert und am Ende nur die benötigten Blöcke ausgegeben. Dabei wird sowohl eine hohe Präzision von mehr als 0,98, als auch eine hoher Sensitivität von mehr als 0,96 erreicht. Der kombinierte F_1-Wert liegt bei etwa 0,97. Auf Grund der geringen Komplexität ist auch der Zeitaufwand für die strukturierte Textextraktion gering.

1. Einleitung

1.1. Motivation

Für die Extraktion von Text aus PDF-Dokumenten kann man sich verschiedene Szenarien vorstellen. Die Bereitstellung von Texten für Vorlesesysteme, um Dokumente sehbehinderten Menschen zugänglich zu machen oder die Aufbereitung von Dokumenten für andere Dateiformate, z. B. um die Texte reflowable[1] zu machen oder, um sie auf anderen Geräten besser darstellen zu können. Ebenfalls denkbar ist auch die Verbesserung der Lesbarkeit durch die Darstellung in anderen Schriftarten oder -farben oder durch die Entfernung störender Fußnoten, Wasserzeichen oder Abbildungen. Ein weiterer wichtiger Punkt ist die Aufbereitung der Texte für semantische Suche[2]. Semantische Suche wird von *Broccoli*, einer am Lehrstuhl für Algorithmen und Datenstrukturen an der Universität Freiburg entwickelte Suchmaschine, verwendet[BBBH12]. Um die Inhalte der wissenschaftlichen Artikel für *Broccoli* zugänglich zu machen, ist es nötig den Inhalt der Artikel in Form von vollständigen Sätzen zu extrahieren. Diese Suche kann für das am gleichen Lehrstuhl entwickelte System *Icecite* verwendet werden. *Icecite* ist ein komfortables Programm zur Anzeige und Organisation von wissenschaftlichen Artikeln im PDF-Format. Es bietet z. B. die automatische Extraktion von Referenzen an und lädt referenzierte Quellen automatisch aus dem Internet herunter[BK13]. PDF ist ein von *Adobe Systems* entwickeltes digitales Dokumentenformat zur Darstellung von Dokumenten [ISO08]. PDF ist ein plattformunabhängiges Format, dessen Vorteile unter anderem darin bestehen, dass sowohl Schriftarten, als auch z. B. Tabellen und Grafiken in das Dokument integriert sind. Ebenso werden Sonderzeichen überall dargestellt, da die Zeichenkodierung einheitlich ist. Auf Grund dieser Vorteile ist PDF ein sehr weit verbreitetes Format, welches von vielen Organisationen zur Bereitstellung von Texten verwendet wird. Zeitungen, Bücher und Prospekte sind als PDF erhältlich, ebenso wie online bereitgestellte wissenschaftliche Artikel. Viele Programme bieten einfache Möglichkeiten, Text aus PDF-Dokumenten zu extrahieren. Dabei wird aber der komplette Text unstrukturiert extrahiert. Tabellen, Grafiken oder Fußnoten etwa unterbrechen dabei Sätze und eine Verwendung für die anfangs genannten Ziele

[1] Stark vereinfacht wird dabei dem Text ein Bereich zugeteilt, den er einnehmen darf[Hen01]. Wenn die Schriftgröße variiert wird, oder der Bereich sich verändert, indem z. B. Tabellen oder Grafiken ein- oder ausgeblendet werden, wird der Text automatisch gebrochen.

[2] Semantische Suche versucht dabei den Inhalt der Suchanfrage zu verstehen und somit relevante Suchergebnisse zu finden[GMM03].

ist fast unmöglich. Das hier vorgestellte System erlaubt es, Text strukturiert aus wissenschaftlichen Artikeln, die im PDF-Format vorliegen, zu extrahieren. Dabei wird die Struktur des Textes erhalten. Jedes Kapitel des zu extrahierenden wissenschaftlichen Artikels wird als vollständiger Teil extrahiert. Die Reihenfolge der Kapitel bleibt erhalten und die Inhalte innerhalb der Kapitel werden in der korrekten Reihenfolge ausgegeben. Sämtliche störenden Elemente, wie Tabellen, Grafiken, Fußnoten, Algorithmen, Trennzeichen, Quellenangaben, etc. werden entfernt, da sie den Fließtext unterbrechen können. Diese Elemente werden im folgenden *uninteressant* genannt. Es wird also der reine Informationen enthaltende Fließtext des wissenschaftlichen Artikels ausgegeben. Der im folgenden *interessant* genannte Fließtext wird Abschnittsweise ausgegeben. Ebenso ausgegeben werden der Titel, der Autor, bzw. die Autoren und der Abstract.

1.2. Schwierigkeiten der Textextraktion aus PDF

Da PDF als ein Format zur Darstellung von Dokumenten entwickelt wurde und nicht zur strukturierten Speicherung von Dokumenteninhalten, ist die Extraktion von Text aus den Dokumenten teilweise schwierig. Die Schwierigkeiten bei den hier betrachteten wissenschaftlichen Artikeln bestehen vor allem aus mehrspaltigen Layouts. Dabei muss die Extraktion nach einem Wort am rechten Rand einer Spalte in der nächsten Zeile der gleichen Spalte fortgesetzt werden. Schwierigkeiten bereiten auch Grafiken, Tabellen, Algorithmen und ähnliche Blöcke, die Text beinhalten. Sie müssen aus der Extraktion ausgeschlossen werden, da sie den Fließtext unterbrechen. Sätze können dadurch auseinander gerissen werden und ihren Sinn verlieren. Zu diesen problematischen Blöcken gehören auch Bildunterschriften oder Fußnoten. Fußnoten beinhalten zwar Informationen, aber ihre Platzierung am unteren Rand der Seiten führt oft dazu, dass Sätze auseinander gerissen werden. Die Markierungen der Fußnoten innerhalb des Fließtextes, Worttrennungen und Trennzeichen unterbrechen Sätze und Worte. Deshalb müssen diese ebenso entfernt werden. Obwohl Quellenangaben in wissenschaftlichen Artikeln unverzichtbar sind, beinhalten sie keine verwertbaren Informationen für die semantische Suche und werden daher entfernt. Weitere Schwierigkeiten für Programme zur Textextraktion sind Kopf- und Fußzeilen. Sie beinhalten oft nur Wasserzeichen, die Kapitelüberschrift, die Seitenangabe, den Namen des Autors bzw. der Autoren oder den Namen des Artikels. Weitere Textbereiche, die keine verwertbaren Informationen beinhalten sind etwa der Titel des Artikels sowie die Autorenangabe. Teilüberschriften trennen den Text zwar für den Leser in sinnvolle Bereiche auf, beinhalten aber auch keine weiteren Informationen. Teilüberschriften werden zwar nicht aus der Extraktion ausgeschlossen, aber in der Ausgabe-Datei des Systems als solche markiert. Das in dieser Arbeit vorgestellte System überwindet diese Schwierigkeiten und erlaubt es so den Fließtext ohne unterbrechende Elemente zu extrahieren.

1.3. Erste Schritte zur Klassifizierung

Die einfache Extraktion von Text aus PDF-Dokumenten kann mit Hilfe von verschiedenen Programmen oder Bibliotheken erfolgen. Für diese Arbeit wurde die *JAVA* Bibliothek *Apache PDFBox*^{TM} verwendet. Zusätzlich wurde die dazugehörige Bibliothek *FontBox*, für die Extraktion der Schrifteigenschaften, eingesetzt. *PDFBox* wurde gewählt, da es einfach zu integrieren ist, unter der der Apache Lizenz 2.0 veröffentlicht wurde und eine große Nutzerbasis hat. *PDFBox* kann sowohl einzelne Buchstaben, als auch Wörter extrahieren. Für die verwendete Extraktion wurde die Klasse `PDFTextStripper` erweitert, sodass die verwendete Schriftart, -farbe und -größe für jedes Zeichen einzeln ausgegeben werden kann.
Wenn nur das Layout eines wissenschaftlichen Artikels betrachtet wird sieht man, dass jeder Artikel aus vielen einzelnen Blöcken zusammengesetzt ist. Der erste Block, auf der ersten Seite, beinhaltet normalerweise den Titel der Arbeit. Der nächste Block beinhaltet die Autorenangaben, gefolgt von einem Block, der den Abstract beinhaltet. Oft kann man dabei, ohne den Text zu lesen, einschätzen, um was für einen Teil es sich bei jedem Block handelt. Dazu werden nur die Schrifteigenschaften benötigt. Blöcke, die Algorithmen beinhalten, sind daran zu erkennen, dass sie in einer nichtproportionalen Schriftart gedruckt sind. Desweiteren sind Überschriften meist größer oder fett gedruckt, Fußnoten dagegen meist kleiner. Die Quellenangaben eines wissenschaftlichen Artikels werden meistens gegenüber den anderen Abschnitten hervorgehoben und befinden sich am Ende. Bildunterschriften sind oft auf Grund der anderen Schrifteigenschaften ebenso von anderen Textteilen zu unterscheiden. Dies waren die grundsätzlichen Gedanken bei der Entwicklung des hier dargestellten Systems. Um diese Gedanken zu überprüfen, wurden verschiedene wissenschaftliche Artikel untersucht und die Merkmale der Schrift extrahiert. Basierend auf Schriftart, -größe, -stärke[3] und -lage[4] wurden anstatt der Buchstaben, unterschiedlich eingefärbte Rechtecke gedruckt, um eine erste Einschätzung über diese Hypothese zu bekommen. In Abb. 1.1 sind zwei Seiten eines wissenschaftlichen Artikels zu sehen. Zu erkennen ist, dass der Abstract die gleiche Farbe und somit Formatierung wie der Rest des Fließtextes hat. Bildbeschreibungen haben ebenso die gleiche Formatierung. Fußnote, Titel, Autoren und Teilüberschriften können jedoch anhand der anderen Farben erkannt werden. Diese ließen sich somit basierend auf ihren Schrifteigenschaften ausschließen. Andere Artikel ergaben ein ähnliches Bild und bestätigten die Hypothese somit teilweise.

[3]Schriftstärken sind z. B. Normalschrift oder Fettschrift.
[4]Die Schriftlage ist z. B. kursiv.

Abbildung 1.1.: Zu sehen sind zwei Seiten eines wissenschaftlichen Artikels. Die einzelnen Buchstaben wurden durch Rechtecke ersetzt. Diese wurden basierend auf Schriftart, -größe, -stärke und -lage eingefärbt. Im linken Bild deutlich zu erkennen sind Titel (blau) und Autoren (türkis), des Artikels. Eine Fußnote am linken unteren Rand wurde schwarz markiert. Auf beiden Seiten sind Tabelle, Grafiken und ein Algorithmus in roter Farbe gedruckt. Der Fließtext wurde gelb und die Teilüberschriften in grün eingefärbt. Der Abstract weist bei diesem Artikel die gleiche Schriftformatierung auf wie der Fließtext. Dieser Teil des Artikels könnte nicht basierend auf den Schrifteigenschaften ausgeschlossen werden.

1.3.1. Klassifizierung durch Schriftart, Schriftgröße und Schriftstärke

Basierend auf den ersten Ergebnissen wurde versucht, nur anhand der Schriftart und -größe eine Klassifizierung der Buchstaben zu erhalten. Dabei sollten Titel, Autoren, Grafiken, Tabellen, Algorithmen, Fußnoten und Teilüberschriften aus dem Text entfernt werden. Eine weitergehende Klassifizierung wäre dadurch vereinfacht. Es müssten nur Abstract, Bildunterschriften und Quellenangaben weiter aussortiert werden. Obwohl die erste visuelle Analyse vielversprechend war, zeigten sich Schwierigkeiten bei diesem Ansatz. In einigen Artikeln waren z. B. die Teilüberschriften fett geschrieben, in anderen in verschiedenen Schriftgrößen, manchmal auch sowohl fett als auch größer, oder sogar in einer anderen Schriftart. Bei den Grafiken und Fußnoten verhielt es sich ähnlich.
Um die Klassifizierung zu erreichen, wurde in einem ersten Ansatz jedem Buchstaben ein Wert zugewiesen. Dieser Wert errechnete sich aus den Schrifteigenschaften Schriftart, -größe und -stärke. Anschließend wurden Buchstaben mit bestimmten Werten aussortiert, in der Hoffnung dadurch unerwünschte Blöcke zu entfernen. Dies führte nicht zum Erfolg. Der nächste Ansatz war, für jede Schriftformatierung eine Häufigkeitsverteilung zu errechnen. Jeder Buchstabe sollte dann in verschiedene Klassen eingeordnet werden. Einige der Klassen waren z. B. größer oder kleiner als die meistverwendete Schriftgröße. Anhand dieser Klassen sollten dann die Buchsta-

ben aussortiert werden. Jedoch konnte kein Zusammenhang gefunden werden, der für alle, oder wenigstens eine große Anzahl, wissenschaftlicher Artikel gleichermaßen galt.

1.3.2. Klassifizierung durch eine Support Vector Machine

Da eine einfache Klassifizierung basierend auf Schriftart, -größe und -stärke nicht möglich war, wurde ein Versuch unternommen, durch die Support Vector Machine[5] *LIBSVM*[CL11] die Klassifizierung zu erreichen. SVM sind zum Klassifizieren von großen Datenmengen geeignet. Je nach Umfang eines wissenschaftlichen Artikels sind meistens zwischen 10.000 und 40.000 Zeichen zu klassifizieren. Als Merkmale wurden einzelne Eigenschaften des Buchstabens, wie z. B. Schriftart bis hin zu allen Eigenschaften inklusive der Position des Zeichens und der Schriftfarbe verwendet. Mithilfe verschiedener Trainingsmengen wurde versucht Zeichen zu klassifizieren. Als Trainingsmengen wurden sowohl einzelne Artikel, als auch eine größere Anzahl von Artikeln gemeinsam verwendet. Dabei wurde auch die Anzahl der Klassen variiert. Eine Klasseneinteilung bestand aus den beiden Klassen Fließtest und Nicht-Fließtext. Eine andere Klasseneinteilung z. B. bestand aus elf Klassen für Fließtext, Titel, Autoren, Abstract, Teilüberschriften, Fußnoten, Abbildungen, Tabellen, Algorithmen, Bildunterschriften und Quellenangaben. Bei einigen Artikeln waren die Ergebnisse nutzbar. Bei anderen Artikeln waren die Ergebnisse jedoch nicht zu gebrauchen.
Da die Klassifizierung im ersten Ansatz für jeden einzelnen Buchstaben durchgeführt wurde, war der nächste Ansatz Buchstaben zuerst zu Worten zusammenzufügen und diese dann zu klassifizieren. Der *PDFTextStripper* von *PDFBox* kann Worte aus dem Text zusammenfügen, jedoch beinhalteten einige der zusammengefügten Worte Leerzeichen. Andere Worte beinhalteten sowohl Buchstaben als auch Zeichen, wieder andere erstreckten sich über mehrere Zeilen. Diese Worte wurden mit dem in Kap. 3.4.1 beschriebenen Algorithmus auf sinnvolle Worte aufgeteilt. Basierend auf diesen Worten wurde erneut die Klassifizierung mit *LIBSVM* versucht. Die Ergebnisse waren besser aber nicht befriedigend, weshalb keine weiteren Versuche mit *LIBSVM* unternommen wurde.

1.3.3. Klassifizierung von Textblöcken

Der ursprüngliche Gedanke war es, aufgrund von bestimmten Eigenschaften eines Textblocks darauf schließen zu können, um was für eine Art Textblock es sich handelt. Die ersten Ansätze verwendeten einzelne Buchstaben oder einzelne Worte. Da

[5] Eine SVM ermöglicht es Objekte in verschiedene Klassen einzuteilen, z. B. Punkte in einem Koordinatensystem. Eine SVM muss zuerst mit einer Menge trainiert werden, für die die jeweilige Klasse bekannt ist.[Bur98].

die Ergebnisse nicht zufriedenstellend waren, wurde im nächsten Ansatz eine Klassifizierung ganzer Blöcke versucht. Ein rekursiver Algorithmus wurde implementiert, um Seiten in mehrere Blöcke aufzuteilen. Dabei wird eine Seite zuerst durch mehrere horizontale Linien zerlegt. Anschließend werden diese Blöcke weiter vertikal aufgeteilt, um dann die erhaltenen Teile wieder horizontal zu teilen. In Kap. 3.4.4 wird der Algorithmus genauer beschrieben.
Anschließend wurden die Blöcke dann wieder untersucht auf verwendete Schriftart, -größe und -stärke. Ebenso wurde untersucht, ob sich die Dichte, also die Anzahl an Wörtern pro Block, unterscheidet. Hintergrund hierfür ist die Annahme, dass sich in Tabellen und Grafiken weniger Zeichen pro Zeile befinden. Auch eine Kombination aus den Formatierungseigenschaften wurde verwendet um die Blöcke zu klassifizieren. Allerdings waren die ersten Ergebnisse nicht gut.

1.4. Der Weg vom wissenschaftlichen Artikel zum Fließtext

Der letztendlich für die Arbeit verwendete Ansatz war, ausgehend von den in Blöcken unterteilten Seiten, immer mehr Text auszusortieren. Mit möglichst großflächigen Blöcken beginnend, werden anschließend nach und nach weitere Blöcke aussortiert. Zuerst werden Tabellen und Grafiken verworfen. Dazu werden Blöcke, die verglichen mit dem Rest des Textes, vermehrt Großbuchstaben und eine größere Menge von Zahlen beinhalten, aussortiert. Dabei werden teilweise auch die Beschriftungen und Algorithmen aussortiert. Anschließend werden die Blöcke auf ihren Inhalt hin überprüft. Hierbei werden zuerst die eventuell noch vorhandenen Bildunterschriften, welche meist mit den Schlüsselworten “Table”, “Figure” oder “Algorithm”[6] beginnen aussortiert. Danach wird die erste Seite analysiert, da sich darauf in den ersten beiden Textblöcke Titel und Autoren befinden. Anschließend wird der darauf folgende Abstract mit dem Schlüsselwort “Abstract” aussortiert. Bei einigen Artikeln befinden sich danach zwei Blöcke, die Kategorien und Schlüsselworte beinhalten. Diese werden entfernt, wenn die entsprechenden Überschriften “Categories” und “Keywords” gefunden werden. Als nächstes werden alle Blöcke, mit anderer Schriftart[7], auf das Schlüsselwort “Algorithm” untersucht und aussortiert. Als letztes folgen Fußnoten, die mit einem Sonderzeichen oder einer Zahl beginnen und fast immer in einer kleineren Schriftart geschrieben sind. Diese Reihenfolge erscheint sinnvoll, da bereits aussortierte Textblöcke nicht weiter untersucht werden müssen. Die verbleibenden Textblöcke werden anschließend hintereinander gereiht und bei jeder Überschrift aufgeteilt. Das Ergebnis wird als XML-Datei abgespeichert.

[6] Ausgehend von der Annahme, dass die meisten wissenschaftlichen Artikel in englischer Sprache geschrieben sind untersucht das System die Artikel auf die englischen Schlüsselworte.

[7] Aus den Namen, der in PDF verwendeten Schriftarten, kann nicht geschlossen werden, ob die Schriftart nichtproportional ist.

2. Strukturierte Textextraktion aus PDF im Überblick

Zur Extraktion von Text aus PDF lassen sich viele verschiedene Programme, Bibliotheken, Anleitungen und wissenschaftliche Arbeiten finden. Eine kurze Google-Suche ergibt mehr als 20 verschiedene Treffer. Fast alle Programme extrahieren den Text nur unstrukturiert. In den folgenden Abschnitten werden daher nur Systeme genannt, die den Text strukturiert oder nach bestimmten Merkmalen gefiltert ausgeben können und deren Dokumentationen die Funktionsweise beschreiben.
Dennoch dürfen die Programme *PDFlib TET 4.4* (*TET*), *Rossinante Web Service* von *XEROX* und *Structured Extract System based on Feature Rules* in dieser Auflistung nicht fehlen, da sie die Möglichkeit strukturierter Textextraktion ebenso bieten.
TET kann laut Herstellerangaben Wortgrenzen erkennen, getrennte Silben wieder enttrennen, Absätze in der korrekten Lesereihenfolge verknüpfen und den Text als XML-Datei ausgeben. Desweiteren erkennt *TET* Tabellen. Kopf- und Fußzeilen können entfernt werden, allerdings muss dazu für jedes PDF-Dokument ein Bereich angegeben werden, in dem sich diese befinden. Dieser Bereich wird dann entfernt. Die Erkennung von relevanten und nicht relevanten Inhalten, wie z.B. Fußnoten oder Quellenangaben fehlt bei *TET*[PDF]. Implementierungsdetails sind nicht zu finden, da es sich um ein kommerzielles Programm handelt.
Rossinante Web Service kann nach einer Registrierung kostenlos benutzt werden. Es erkennt Kopf- und Fußzeilen, Seitenzahlen, Bildbeschreibungen und Fußnoten, ebenso wird die Inhaltsangabe erkannt. Der Text kann automatisch in Blöcke aufgeteilt werden und in der korrekten Reihenfolge ausgegeben werden. Auch hier fehlt die Möglichkeit, nicht relevante Blöcke automatisch zu entfernen[XER].
Structured Extract System based on Feature Rules ist ein System, um speziell aus wissenschaftlichen Artikeln Informationen strukturiert zu extrahieren. Der Algorithmus ermöglicht es, jede Art von Bereich wie Titel, Abbildung oder Tabelle zu erkennen. Für jede Art von Bereich können verschiedene Regeln aufgestellt werden. Regeln können etwa die Schriftart, -größe, -stärke oder die Zeilenhöhe beinhalten. Die Ergebnisse des Algorithmus hängen stark von den verwendeten Regeln ab, die darüber hinaus für jedes Layout erstellt werden müssen[CC13]. Damit ist der Algorithmus viel zu unflexibel für eine große Anzahl unterschiedlicher Layouts.

2.1. LA-PDFText

Das System *Layout-Aware PDF Text Extraction* (*LA-PDFText*) wurde für die *Bio NLP*-Community entwickelt, um den Inhalt wissenschaftlicher Artikel strukturiert zu extrahieren, da der Zugriff auf die Informationen in wissenschaftlichen Artikeln für die Entwicklung von Text Mining-Anwendungen[1] sehr wichtig ist. Bei der Entwicklung wurden drei Ziele verfolgt. Das Erste ist das erfolgreiche Verarbeiten von Artikeln, die entweder einspaltige, zweispaltige oder gemischte Layouts haben. Das zweite Ziel ist die fehlerfreie Extraktion von Text, der dabei in einzelne Abschnitte gegliedert wird. Das dritte Ziel ist, dass dabei keine Abbildungen, Tabellen, oder ähnliches den Text unterbrechen dürfen.

2.1.1. Funktion

Die Funktionsweise lässt sich grob in drei Schritte unterteilen. Zuerst werden zusammenhängende Textblöcke generiert. Anschließend werden diese Textblöcke in verschiedene Kategorien eingeordnet. Als letztes werden die Textblöcke in der richtigen Reihenfolge zusammengesetzt und jeweils zusammen mit der Teilüberschrift ausgegeben.
Um die Textblöcke zu erstellen, wird zuerst mit der Bibliothek *JPedal* eine Bounding-Box um jedes Wort erstellt. Diese Bounding-Box wird um einen statistischen Wert erweitert. Dieser Wert wird für jede Seite errechnet. Überschneiden sich zwei Bounding-Boxes und sind die Schrifteigenschaften gleich, werden die Worte zu einem Bereich zusammengefasst. Sind die Blöcke berechnet, wird jeder Block regelbasierend in eine Kategorie klassifiziert. Diese Regeln können von den Benutzern des Systems angepasst werden, sodass wissenschaftliche Artikel unterschiedlicher Layouts analysiert werden können. Im letzten Schritt werden Blöcke gleicher Kategorie zusammengefasst und anschließend in der korrekten Reihenfolge ausgegeben[RPHB12].

2.1.2. Bewertung

LA-PDFText erreicht ähnliche Ziele, wie die vom in dieser Arbeit dargestellten System. Allerdings unterscheiden sich die Wege zu den Zielen. *LA-PDFText* erstellt ebenso Textblöcke, jedoch werden diese aus einzelnen Worten zusammengesetzt und nicht die Seite in Textblöcke unterteilt. Anschließend werden die Textblöcke kategorisiert und letztendlich in der korrekten Reihenfolge ausgegeben. Die Regeln von *LA-PDFText* sind sehr viel strenger, als die des in dieser Arbeit vorgestellten Systems. *LA-PDFText* ist somit deutlich weniger flexibel. Die erreichte Präzision liegt bei 0,96, die Sensitivität bei 0,89, was einen F_1-Wert von 0,91 ergibt. Diese Werte beziehen sich aber auf die korrekte Klassifizierung in die jeweilige Kategorie. Die

[1] Text-Mining ist die Extraktion von Informationen aus unstrukturierten Texten, wie etwa aus Büchern[FD95].

Werte für das in dieser Arbeit vorgestellte System beziehen sich jedoch auf die Klassifizierung in interessanten und uninteressanten Text.

2.2. SectLabel

SectLabel ist ein System, um die logische Struktur von PDF-Dateien zu finden. Dabei werden, wenn möglich, die durch OCR[2] gewonnenen Schrifteigenschaften und Textpositionen genutzt. Das System ermöglicht die Klassifizierung von Textteilen in Bereiche wie z. B. Titel, Abstract oder Fließtext und wurde speziell für wissenschaftliche Artikel entwickelt. Das System besteht aus zwei primären Komponenten. Die Komponente "Logical structure classification" ermöglicht eine Einordnung in 23 Kategorien wie Titel, Überschrift, etc. Die zweite Komponente ist die "generic section classification", die z. B. Überschrift in Einleitung, Abstract, etc. klassifizieren kann.

2.2.1. Funktion

Um wissenschaftliche Artikel in verschiedene Bereiche klassifizieren zu können, muss das System zuerst trainiert werden. Der "logical structure extractor" extrahiert die relative Position von Wörtern innerhalb eines Dokuments. So wird etwa gelernt, dass der Titel sich auf der ersten Seite meistens im oberen Drittel befindet, während das Literaturverzeichnis am Ende einer wissenschaftlichen Arbeit zu finden ist. Außerdem werden Inhalte wie "1.1" gesucht, was auf eine Teilüberschrift hindeutet, oder "1www" was eine Webadresse in einer Fußnote sein könnte. Andere Muster, wie Webadressen oder E-Mail-Adressen deuten auf Inhalte im Literaturverzeichnis hin, während runde Klammern Formeln innerhalb des Fließtextes sein könnten. Als letzte Klassifizierungshilfe wird die Zeilenlänge verwendet. Ist die Zeile lang, kann sie Teil des Fließtextes sein, ist sie kurz, kann die Zeile eine Überschrift sein.
Die Überschriften werden mit Hilfe des "generic section extractors" analysiert. Dieser untersucht wieder die relative Position innerhalb des Dokuments. Außerdem werden die ersten beiden Worte der Überschriften auf bestimmte Schlüsselworte untersucht, um den logischen Sinn jedes Bereichs klassifizieren zu können. Ein Kapitel "Related Work" kann z. B. die Überschrift "Previous Work" oder "Literature Review" haben. Bereiche, die diese Überschrift haben, werden als logischer Bereich "Related Work" klassifiziert.
Wenn Schrifteigenschaften, wie Schriftgröße, -lage oder -stärke zur Verfügung stehen, werden diese ebenfalls verwendet. Dies erhöht die Genauigkeit der Klassifizierung drastisch, da z. B. bei manchen der untersuchten Artikel keine Nummerierung vor den Überschriften steht. Aus den durch die beiden Extraktoren gewonnenen Informationen werden zwei Modelle, "logical structure model" und "generic section

[2]OCR steht für *optical character recognition*. OCR ermöglicht es in Bildern Text zu erkennen.

model" erstellt. Mit deren Hilfe wird in der Testphase ein wissenschaftlicher Artikel klassifiziert[KLN10].

2.2.2. Bewertung

Auch *SectLabel* versucht die gefundenen Bereiche in verschiedene Klassen einzuordnen, während das in dieser Arbeit vorgestellte System nur interessanten und uninteressanten Text unterscheidet. Dabei werden gute Ergebnisse erreicht, die aber ebenfalls nicht direkt mit den Ergebnissen aus dem in dieser Arbeit vorgestellten System vergleichbar sind. Da *SectLabel* zuerst trainiert werden muss, ist eine gute Klassifizierung von der ausgewählten Trainingsmenge abhängig. Eine aufwendige Möglichkeit wäre es, für verschiedene Konferenzen, etc. verschiedene Modelle zu erstellen, um eine hohe Genauigkeit zu erreichen.

2.3. A System for Converting PDF Documents into Structured XML format

Das System wurde für allgemeine PDF-Dokumente entwickelt, aber lässt sich auch auf wissenschaftliche Artikel anwenden. Es werden zwei Anwendungsfälle beschrieben. Der erste Fall beschreibt die Nutzung des Systems, um alte Reparaturanleitungen für Autos in ein geeignetes digitales Format zu konvertieren. Im zweiten Fall geht es darum, mit OCR erfasste Fertigungsaufträge in Abschnitte aufzuteilen. Dabei werden Kopf- und Fußzeilen erfasst, die Lesereihenfolge eingehalten und eine Strukturierung anhand der jeweiligen Inhaltsangabe vorgenommen.

2.3.1. Funktion

Das System extrahiert zuerst den Text aus PDF-Dokumenten zusammen mit den jeweiligen Koordinaten. Anschließend wird mit einem in [Meu05] beschriebenen modifizierten XY-Cut-Algorithmus[3] der Text aufgeteilt und Bilder entfernt. Dabei können sowohl externe Bilder, die als Bilddateien eingefügt, als auch Vektorbilder, die aus Pfadangaben bestehen, entfernt werden. Dabei werden auch die Bildunterschriften entfernt. Danach werden Kopf- und Fußzeilen entfernt. Hier macht das System sich die geringe Variabilität des Textes in den Kopf- und Fußzeilen zu Nutze. Zahlen werden bei der Suche nach Kopf- und Fußzeilen durch Platzhalter ersetzt. Somit steht in einer Fußzeile, die die Seitenzahl angibt auf jeder Seite das gleiche. In Kopfzeilen dagegen steht oft die jeweilige Kapitelüberschrift. Erstrecken sich Kapitel über

[3]Der XY-Cut-Algorithmus teilt z. B. eine Textseite in verschiedene Teilblöcke auf. Dazu wird entweder zuerst horizontal oder vertikal der größte leere Bereich gesucht, mit dem die Seite aufgeteilt werden kann. Anschließend werden der erhaltenen Teilbereiche rekursiv weiter aufgeteilt. Dabei wird jeweils abwechselnd horizontal oder vertikal aufgeteilt.

mehrere Seiten wiederholt sich auch hier der Inhalt. Anschließend wird die Lesereihenfolge gefunden, indem wieder der modifizierte XY-Cut-Algorithmus verwendet wird. Dabei wird der Text auch in einzelne Bereiche aufgeteilt. Zum Schluss wird, wenn ein Inhaltsverzeichnis existiert, eine darauf basierende Anordnung der Blöcke vorgenommen[DM06].

2.3.2. Bewertung

Das System ermöglicht es, Text aus PDF-Dokumenten in eine strukturierte XML-Datei zu übertragen. Dabei werden Bilder und Bildunterschriften entfernt. Eine Verwendung für wissenschaftliche Arbeiten wäre möglich, da das System nicht für eine bestimmte Art von Text entwickelt wurde. Die Möglichkeit, Tabellen und Fußnoten zu extrahieren, fehlt allerdings, weshalb das System für die in dieser Arbeit verfolgten Ziele nicht geeignet ist.

2.4. PTX

PTX ist ein Framework, welches die hierarchische Struktur von wissenschaftlichen Artikeln im PDF-Format extrahiert. Dabei wird davon ausgegangen, dass wissenschaftliche Artikel meistens von Forschern aus dem jeweiligen Fachgebiet untersucht werden. Diese Annahme macht sich das System zu Nutze, indem es bei der Analyse ein Regelwerk verlangt, welches zum Klassifizieren verwendet wird. Die Ausgabe von *PTX* ist eine XML-Datei, die den Text in Abschnitten wie Titel, Autor, Abstract, Textkörper oder Literaturverzeichnis beinhaltet.

2.4.1. Funktion

Die Funktion ist in drei Stufen aufgeteilt. Die erste Stufe ist die Schrifterkennung mit Hilfe von OCR. Dabei werden Buchstaben und dazugehörige Eigenschaften wie Schriftart, -größe etc. extrahiert. Innerhalb der Seiten befinden sich grafische Zonen, die Abschnitte beinhalten. Abschnitte wiederum beinhalten Zeilen, welche Zeichen beinhalten. Zonen können auch Tabellen oder Grafiken beinhalten. Die zweite Stufe filtert die OCR-Ausgabe. Dabei werden auch Wörter aus einzelnen Buchstaben zusammengefasst. In der dritten Stufe wird ein spezielles Regelwerk angewendet. Dieses Regelwerk besteht aus Perl-Code. Jedes Journal, Buch, etc. benötigt mehrere Regelwerke, welche jeweils für einen Bereiche gelten. Ein Regelwerk für Zonen wird ausgeführt, wenn in der Eingabedatei eine Markierung für eine beginnende Zone gefunden wird. Regelwerke können die aktuellen Eigenschaften des Bereichs lesen, sowie alle vorangegangenen Eigenschaften. Auf diese Weise kann der Übergang, von einer Zone in die Nächste, den Beginn des Abstracts markieren, wenn gerade die erste Seite analysiert wird[HLT05].

2.4.2. Bewertung

Bei diesem System wird auch keine Einordnung in interessanten und uninteressanten Text durchgeführt. Dennoch ermöglicht das System eine sichere Einordnung in verschiedene Textbereiche wie Titel, Abstract oder Literaturverzeichnis. Dabei wurde in einer Testmenge von wissenschaftlichen Artikeln eine Präzision von 0,975 und eine Sensitivität von 0,962 erreicht. Der F_1-Wert liegt bei 0,968. Bei einer zweiten Menge von wissenschaftlichen Artikeln erreichte das System eine Präzision von 0,93 mit einer Sensitivität von 0,942 und einem F_1-Wert von 0,936. Da das System für verschiedene Layouts verschiedene Regeln benötigt, ist es auch ungeeignet, um eine große Menge unterschiedlicher Artikel erfolgreich zu klassifizieren. Außerdem wurden mit einem nicht gut ausgereiften Regelwerk, bei einer Menge von wissenschaftlichen Arbeiten, deutlich schlechtere Ergebnisse erzielt. Die Präzision lag bei 0,739, die Sensitivität bei 0,904, was einen F_1-Wert von 0,813 ergibt.

2.5. Labler

Das System *Labler* ist ein System um durch OCR erfassten Dokumenten, Text strukturiert zu extrahieren. Dabei wird nur die Struktur des Dokuments analysiert, ohne den Inhalt zu verarbeiten. Von Textzeilen werden z. B. die Position, Höhe und Koordinaten verarbeitet. Ebenso werden Wiederholungen von geometrischen Mustern innerhalb von Bereichen ausgewertet.

2.5.1. Funktion

Zu Beginn wird das Dokument in kleinstmögliche Blöcke aufgeteilt. Diese Blöcke werden untersucht, ob sich geometrische Muster wiederholen. Außerdem werden isolierte Blöcke gesucht, die zu Mustern passen, welche an anderen Stellen gefunden wurden. Dann werden die Blöcke zu Gruppen zusammengefasst, welche anschließend ebenfalls zusammengefasst werden, bis eine Baumstruktur entsteht. Die Elemente der Baumstruktur werden mit gespeicherten Mustern verglichen, um in einer ersten Stufe eine Aufteilung des Dokuments in eine Hierarchie logischer Elemente wie “Dokument”, “Dokumentenkörper” und “Abschnitt” zu erreichen. Anschließend erfolgt eine Klassifizierung in Typen logischer Struktur. Logische Struktur ist z. B. ein “Abschnitt”, welcher ein “besonderer Abschnitt”, wie “erster Abschnitt”, oder ein “normaler Abschnitt” sein kann[Sum98].

2.5.2. Bewertung

Labler erreicht bei der Aufteilung des Dokuments, in die verschiedenen Bereiche, eine Präzision von 0,823 und eine Sensitivität von 0,797, was einen F_1-Wert von

0,810 ergibt. Bei der Klassifizierung aller korrekt aufgeteilten Bereiche erzielt *Labler* eine Präzision von 0,809. Die von *Labler* erzielten Werte für die Klassifizierung in logische Strukturen sind akzeptabel. Im Vergleich zu den regelbasierenden Verfahren wie *PTX* in Kap. 2.4 ist *Labler* jedoch deutlich flexibler und damit eher geeignet den Text strukturiert aus einer großen Menge an wissenschaftlichen Arbeiten zu extrahieren.

3. Theoretische Analyse

3.1. Aufbau

Der grundsätzliche Gedanke bei der Entwicklung des Systems war es, nach und nach weitere Teile des zu verarbeitenden Artikels auszuschließen, um am Ende nur den wichtigen Fließtext zu erhalten. Dabei musste stets darauf geachtet werden, dass nicht fälschlicherweise Teile des Fließtextes aussortiert werden. Jede Klasse des Systems stellt Methoden bereit, die bestimmte Teile klassifizieren und aussortieren können.

3.1.1. Extraktorklassen

Die Klasse `PDFTextExtractor` ist der Hauptbestandteil des Systems. Die Klasse stellt alle verwendeten Datenstrukturen zur Verfügung. Außerdem wird hier das PDF-Dokument geladen. Alle Analyseklassen, die Klasse `PropertyExtractor` und die Ausgabeklassen werden von hier gestartet. In der Klasse `PropertyExtractor` werden sowohl die einzelnen Buchstaben als auch die zusammenhängenden Wörter des wissenschaftlichen Artikels, zusammen mit allen Eigenschaften wie Position, Schriftart, -größe etc. extrahiert. Die Klasse `WordPositions` wird benötigt, um fehlerhaft erkannte Wörter[1] zu finden und entsprechend zu verarbeiten.

3.1.2. Analyseklassen

Das System besteht im Wesentlichen aus drei wichtigen Analysewerkzeugen. Das erste Werkzeug ist der `ContentAnalyzer`, zur Analyse des Inhalts von Textblöcken. Hier werden Fußnoten, Bildbeschreibungen, Abstract, Titel, Quellenangaben und Grafiken aussortiert. Das zweite Werkzeug ist `FontPropertyDistributions`. Es dient zur Analyse sämtlicher Eigenschaften wie Schriftart und -größe. Außerdem werden Häufigkeitsverteilungen berechnet. Die Häufigkeit von Großbuchstaben und Zahlen wird zum Aussortieren von Grafiken und Tabellen verwendet. Die Klasse `GeometryAnalyzer` teilt die Seiten des Artikels in einzelne Textbereiche auf. Auch

[1] Einige der von *PDFBox* extrahierten Worte beinhalten Leerzeichen, Zeilenumbrüche, Sonderzeichen oder bestehen sowohl aus Buchstaben, als auch aus Ziffern.

wird hier die Dichte[2] von Textbereichen berechnet. Sind alle Analysewerkzeuge abgeschlossen, wird in der Klasse `GeometryAnalyzer` aus den nicht zusammenhängenden interessanten Textblöcken ein zusammenhängender Text erstellt.

3.1.3. Ausgabeklassen

Die Klassen `CharacterPagePrinter`, für die grafische Ausgabe einzelner Buchstaben und die Klasse `WordPagePrinter`, für die grafische Ausgabe ganzer Wörter, werden nur für die händische Analyse zur Visualisierung genutzt. Für die Ausgabe des Fließtextes wird die Klasse `XMLOutput` verwendet. Sie generiert die Ausgabe in Form einer XML-Datei. Diese Datei beinhaltet Titel, Autorenangaben, Abstract und jedes Kapitel des wissenschaftlichen Artikels.

3.2. Ablauf

Der Ablauf des Systems kann grob in drei Abschnitte unterschieden werden. Zuerst findet die Extraktion der Worte und Vorbereiten der Datenstrukturen statt. Anschließend folgt die Analyse des Textes, die den Text in interessant und nichtinteressant aufteilt. Final erfolgt die Ausgabe des Textes. Der Ablauf ist am Ende des Kapitels auch in einem Diagramm unter Abb. 3.2 zu sehen.

3.2.1. Vorbereitung und Wortextraktion

Zu Beginn wird der zu extrahierende wissenschaftliche Artikel, in Form eines PDF-Dokuments, in der Klasse `PDFTextExtractor` geladen. Anschließend werden die Buchstaben aus dem Dokument extrahiert. Dazu wird die Klasse `PropertyExtractor` initialisiert, um mit der Funktion `getText()` den Text jeder einzelnen PDF-Seite, in Form einzelner Buchstaben, zu extrahieren. Diese Buchstaben werden als `Character WithProperties` in eine Liste für jede Seite zusammengefasst. Es werden dabei eine eindeutige Nummer als Identifikator, Position, Farbe, Schriftart und -größe abgespeichert. Während der Extraktion der Buchstaben erfolgt ein erstes Zusammenfügen von Worten durch *PDFBox*. Diese Worte haben eindeutige Wortidentifikatoren. Diese Wortidentifikatoren werden ebenso in den einzelnen Buchstaben abgespeichert, dass eine Zuordnung von Buchstaben zu Worten erfolgen kann. Die meisten Worte werden korrekt erkannt und keine weitere Verarbeitung ist notwendig. Einige der gefundenen Worte beinhalten jedoch Leerzeichen und Sonderzeichen oder sowohl Buchstaben als auch Zahlen oder Zeilenumbrüche. Einige Worte beinhalten auch verschiedene Schrifteigenschaften. Deswegen werden anschließend in der Funktion `buildWords()` die gefundenen Wörter weiter verarbeitet. Dazu wird die

[2] Anzahl Wörter bezogen auf die Größe des Textbereiches.

Klasse `WordPositons` initialisiert und deren Funktion `calculateWords()` aufgerufen. Diese Funktion iteriert über alle Buchstaben einer Seite und ordnet sie anhand ihres Wortidentifikators, über die Funktion `addCharacter()`, in ein vorhandenes Wort ein. Die Funktion `addCharacter()` prüft die Eigenschaften des einzufügenden Buchstabens. Passen die Eigenschaften des Buchstaben zu denen des Wortes, wird der Buchstabe eingefügt. Passen die Eigenschaften jedoch nicht zusammen, z. B. wenn zwei verschiedene Schriftarten verwendet wurden, schlägt das Einfügen fehl und ein neues Wort wird erstellt. Alle Buchstaben eines von *PDFBox* gefundenen Wortes, die nicht in das ursprüngliche Wort eingefügt werden konnten, werden wenn möglich in das neue Wort eingefügt.

3.2.2. Analyseschritte

Nachdem die Datenstrukturen vorbereitet sind, wird die Klasse `GeometryAnalyzer` initialisiert. Jede Seite wird mit Hilfe der darin enthaltenen Funktion `makeGrid()` für die Analysen in einzelne Blöcke zerlegt. Die Ausdehnung der Wörter einer Seite werden zuerst erweitert. Dazu werden die minimalen Koordinaten verkleinert und die maximalen Koordinaten vergrößert. Die Wörter werden ausgedehnt, dass sie sich teilweise überschneiden und beim horizontalen Teilen mit Hilfe der Funktion `make HorizontalLines()` nicht nach jeder Zeile ein neuer Block beginnt. Die Worte werden vertikal auf die doppelte Schriftgröße ausgedehnt, horizontal um 5 Pixel. Mit diesen Werten wurden in Versuchen mit verschiedenen wissenschaftlichen Artikeln die besten Ergebnisse erzielt. Eine horizontale Ausdehnung von 5 Pixeln führt dazu, dass bei einem zweispaltigen Layout beide Spalten voneinander getrennt werden. Dieser Wert ist jedoch trotzdem noch groß genug, dass nicht fälschlicherweise ein einzeiliger Textblock vertikal geteilt wird. Bei einer vertikalen Ausdehnung auf die doppelte Schriftgröße werden Textbereiche, mit bis zu zweifachem Zeilenabstand, zu einem Block zusammengefasst. Der Abstand zwischen Textblöcken und Überschriften, zwischen Bildern und Beschreibungen, oder vor dem Literaturverzeichnis ist meist größer. Hier wird der Text korrekt geteilt. Die beim horizontalen Teilen entstandenen Blöcke werden anschließend mit der Funktion `makeVerticalLines()` vertikal geteilt. Dabei wird versucht, den Block zuerst in der Mitte aufzuteilen. Dies entspricht dem Teilen bei einem zweispaltigen Layout des Dokuments. Die bereits einmal horizontal und vertikal aufgeteilten Blöcke werden im dritten Schritt wieder horizontal geteilt. Blöcke, die nur aus wenigen Worten bestehen, werden mit den Nachbarblöcken zusammengelegt. Dies dient dazu, dass nicht zu viele kleine Fragmente entstehen, da für die weiteren Analysen möglichst große zusammenhängende Blöcke gefunden werden sollen.
Um im nächsten Schritt Text auszuschließen, werden in der Funktion `calculate UppercaseAndNumberDensities()` alle Worte auf Anzahl an Großbuchstaben und Zahlen untersucht und die Anzahl in einer `HashMap` abgelegt. Außerdem wird für den gesamten Artikel die durchschnittliche Anzahl an Großbuchstaben und Zahlen pro Wort berechnet. Da Tabellen und Grafiken überdurchschnittlich viele Großbuchsta-

ben und Zahlen aufweisen, werden in der Funktion `calculateNumbersAndCaseForArea()` der Klasse `GeometryAnalyzer` für jeden Bereich die Anzahl an Großbuchstaben und Zahlen pro Wort berechnet. Anhand dieser Werte werden Blöcke mit vergleichsweise hoher Anzahl an Großbuchstaben und Zahlen aussortiert. In verschiedenen Versuchen hat sich gezeigt, dass die besten Ergebnisse mit einem Faktor von 1,2 erzielt werden. Beinhalten Blöcke also mehr Zahlen und Großbuchstaben als der 1,2-fache Durchschnittswert des gesamten Textes, werden sie aussortiert. Die Worte dieser Blöcke werden mit einem *interest* von *false* markiert. Einmal aussortierte Blöcke werden von den nachfolgenden Analysen nicht mehr bewertet. Die Funktion findet die meisten Grafiken und Tabellen, jedoch können wegen dem hohen Schwellwert nicht alle Grafiken und Tabellen entfernt werden. Ein niedrigerer Schwellwert könnte aber zum falschen aussortieren von Blöcken führen, die viele Formeln beinhalten, da in diesen oft Zahlen und Großbuchstaben verwendet werden. Teilweise beinhalten die Textblöcke auch die Bildunterschriften, welche deshalb korrekt mit entfernt werden.
Für die nächsten Analysen wird die Klasse `ContentAnalyzer` initialisiert und mit deren Funktion `findCaptions()` die zuvor nicht entfernten Bildunterschriften entfernt. Da diese sich in der Schriftart oft nicht vom Fließtext unterscheiden, muss der Inhalt des Textes analysiert werden. Alle Textblöcke, deren *interest* nicht *false* ist, werden untersucht. Die Funktion iteriert über alle Wörter und prüft dabei, ob ein Wort, dass am weitesten links steht, "*Table*", "*Figure*" oder "*Algorithm*" ist. Es wird nicht geprüft, ob dieses Wort auch am oberen Rand des Textblocks steht. Wegen dem hohen Schwellwert in der ersten Analyse kann es passieren, dass nicht alle Grafiken und Tabellen erkannt werden. Da einige der Blöcke sowohl Grafiken als auch Bildunterschriften beinhalten, findet die Funktion `findCaptions()` auch diese. Da einige Blöcke Bezug nehmen auf Grafiken, Tabellen und Algorithmen, kommen die Schlüsselworte auch vor, obwohl der Textblock keine Bildunterschrift ist. Aus diesem Grund wird geprüft, ob nach dem gefundenen Schlüsselwort direkt eine Zahl und ein Doppelpunkt folgt. Nur wenn diese Bedingungen erfüllt werden, werden alle Worte des Bereichs mit einem *interest* von *false* markiert.
Die Funktion `makeGrid()` teilt den Text in mehrere große Textblöcke auf. Da in den ersten beiden Analyseschritten einige große Blöcke ausgefiltert wurden, wird die Funktion `makeGrid()` erneut aufgerufen, um kleinere Blöcke zu erhalten. Versuche mit mehreren wissenschaftlichen Artikeln haben gezeigt, dass einige der Fußnoten mit den vorangehenden Bereichen zusammengefasst werden. Dies geschieht, da Fußnoten oft aus wenigen Worten bestehen und kleine Textblöcke mit den Nachbarblöcken zusammengefasst werden. Für die vorangehenden Analyseschritte war es aber wichtig, dass die Textblöcke nicht zu klein sind. Die Funktion `findFootnotesAlgorithm()` in der Klasse `ContentAnalyzer` filtert aus dem Text sowohl Fußnoten als auch nicht gefundene Algorithmen heraus. Da Fußnoten mit einer hochgestellten Zahl oder einem Sonderzeichen beginnen und Algorithmen ebenso meistens mit dem Schlüsselwort "Algorithm", gefolgt von einer Zahl, beginnen, werden diese Bereiche vom gleichen Algorithmus gesucht. Ähnlich wie bereits in der Funktion `findCaptions()`, werden alle Textblöcke daraufhin untersucht, ob das Wort, das am wei-

testen links steht, eine Zahl, ein Sonderzeichen oder das Schlüsselwort “Algorithm” ist. Desweiteren wird in dieser Analyse geprüft, ob das Wort auch am weitesten oben im jeweiligen Block positioniert ist. Treffen die Bedingungen zu, werden die Worte wieder mit dem *interest* von *false* markiert und somit aussortiert.
Im nächsten Schritt werden in der Funktion `findReferences()` der Klasse `Content Analyzer` die Quellenangaben entfernt. Dazu werden wieder alle, noch nicht aussortierten Blöcke untersucht, ob sie mit dem Schlüsselwort “References” beginnen. Dazu werden die Seiten, beginnend bei der Letzten, rückwärts untersucht. Da Quellenangaben am Ende von wissenschaftlichen Artikeln stehen, werden auch alle nachfolgenden Blöcke mit einem *interest* von *false* markiert. Dabei ist es egal, ob die Quellenangaben in einem einspaltigen oder zweispaltigen Layout angegeben werden. In einigen Artikeln folgen nach den Quellenangaben Diagramme oder Tabellen zur Erklärung. Diese werden ebenfalls korrekt ausgefiltert, sollte dies nicht bereits vorher geschehen sein. Die Schriftart der Überschrift *References* wird in einer Variable für einen späteren Schritt abgespeichert.
Anschließend wird mit der Funktion `findAbstract()` der Klasse `ContentAnalyzer` die erste Seite des Artikels auf das Schlüsselwort “Abstract” untersucht. Der beinhaltende Block wird, wie in den vorangehenden Schritten, markiert und in den folgenden Analyseschritten nicht mehr betrachtet. In einer Variable wird, ebenso wie beim vorangehenden Schritt, die Schriftart der Überschrift *Abstract* abgespeichert. Die Worte des Abstract werden nach ihrer Position sortiert und der gesamte Block für die spätere Ausgabe in einer Variable gespeichert.
Da seit der ersten Suche nach Bildunterschriften die Blockgrößen verkleinert wurden, und eventuell nicht alle Grafiken entfernt wurden, wird erneut nach Bildunterschriften gesucht. Dazu wird die Funktion `findFigures()` im `ContentAnalyzer` aufgerufen. Diese Funktion durchsucht erneut die Blöcke. Befindet sich am linken oberen Rand eines Blockes das Schlüsselwort “Figures”, gefolgt von bis zu vier Zeichen und einem Doppelpunkt, wird auch dieser Block mit einem *interest* von *false* markiert.
Anschließend wird die Funktion `extractAuthorAndTitle()` aufgerufen um den oder die Autoren und den Titel des Artikels zu extrahieren. Die Funktion startet eine modifizierte Version der Funktion `makeGrid()` mit dem Namen `makeFirstPage Grid()`. Dabei wird die erste Seite des Artikels nur horizontal geteilt. Die vertikale Ausdehnung der Buchstaben wird auf $\frac{1}{3}$ der Schrifthöhe verringert. Mit diesem Wert wird ein mehrzeiliger Titel noch als ein zusammenhängender Block ausgegeben. Die nachfolgenden Angaben über die Autoren werden trotzdem als ein anderer Block ausgegeben. In dieser Funktion werden Blöcke mit wenigen Worten auch nicht mit ihren Nachbarblöcken zusammengefasst, wie es die Funktion `makeGrid()` macht. Die beiden Blöcke mit Titel und Autoren werden nach den Koordinaten sortiert und für die spätere Ausgabe in Variablen abgelegt. Da Titel und Autorenangaben nicht weiter analysiert werden müssen, werden auch diese Blöcke mit dem *interest* von *false* markiert.
Die meisten nicht erwünschten Blöcke wurden bereits aussortiert. In einigen Fällen sind aber noch Fragmente von unerwünschten Blöcken vorhanden. Diese Fragmente

sind meist verbliebene Bestandteile von Grafiken, Tabellen oder Fußnoten. Deshalb wird im nächsten Schritt die Funktion `findSmallerBlocks()` aufgerufen. Diese ruft die Funktion `makeGrid()` der Klasse `GeometryAnalyzer` erneut auf, um den Artikel wieder in größere Blöcke aufzuteilen. Die horizontale Ausdehnung der Worte bleibt bei 5 Pixeln, während die vertikale Ausdehnung auf etwa die dreifache Schriftgröße festgelegt wird. Kleine Blöcke werden in diesem Schritt allerdings nicht mit den Nachbarblöcken zusammengelegt, da einige Fragmente nur aus 1 oder 2 Worten bestehen können. Anschließend wird die Funktion `removeAreasWithOtherWidths ThanAverage()` der Klasse `GeometryAnalyzer` aufgerufen. Diese Funktion iteriert über alle Blöcke und misst die durchschnittliche horizontale Ausdehnung jeder Zeile eines Blockes. Zeilen innerhalb des Fließtextes haben meist in etwa die gleiche Breite, während bei den Fragmenten die Reihen oft große Unterschiede in den Breiten aufweisen. In diesem Schritt werden folgende Blöcke mit einem *interest* von *false* markiert: Blöcke, deren Breiten entweder größer als 150% der durchschnittlichen Breite sind und Blöcke, die nur eine sehr geringe Wortanzahl aufweisen und deren Breiten kleiner als 90% der durchschnittlichen Breiten sind. Der erste Fall tritt unter ein, wenn Grafiken oder Tabellen die gesamte Seitenbreite nutzen, während der Fließtext in einem zweispaltigen Layout angeordnet ist. Der zweite Fall tritt ein, wenn einzelne Zeilen einer Tabelle oder Kopf- und Fußzeilen gefunden werden.

3.2.3. Ausgabe

Sind die Analysen abgeschlossen, ist der Text bereit zur Ausgabe. Alle nicht aussortieren Textblöcke werden mit der Funktion `makeOnePage()` der Klasse `Geometry Analyzer` aneinandergehängt. Dazu wird jede Seite vertikal in der Mitte geteilt. Jeweils die rechte Hälfte wird an die linke Hälfte angefügt und das Ergebnis an die vorangehende Seite angehängt. Lässt sich eine Seite nicht vertikal teilen, wird die Seite als Gesamtes angehängt. In diesem Fall ist entweder das Layout des kompletten Artikels nicht zweispaltig, oder eine der beiden Spalten besteht nur aus aussortieren Blöcken.
Für die Ausgabe wird als erstes die Funktion `removeFootnoteSign()` aufgerufen. Diese Funktion entfernt die im Text vorkommenden Anmerkungsziffern[3]. Die Funktion iteriert über alle Worte im Fließtext. Die hochgestellten Anmerkungsziffern haben eine von der Zeile abweichende y-Koordinate. Werden einzelne Worte gefunden, deren y-Koordinaten von den anderen y-Koordinaten der Zeile abweichen, wird noch geprüft, ob diese Worte aus höchstens zwei Zeichen bestehen. Trifft dies zu, werden diese Zeichen aussortiert.
Als nächstes wird die Funktion `removeUnwantedGaps()` aufgerufen. Die Funktion entfernt unerwünschte Leerzeichen und Zeilenumbrüche. Dabei werden vor “.”, “,” und “-” alle *Whitespace*-Zeichen[4] entfernt, indem die Java-Funktion `replaceAll()` aufgerufen wird. Ebenso entfernt werden in dieser Funktion Quellenverweise, die in

[3] Das im Text hochgestellte Zeichen, welches eine Fußnote anzeigt, wird Anmerkungsziffer genannt.
[4] *Whitespace*-Zeichen sind Leerzeichen, Tabulatorzeichen und Zeilenumbrüche.

eckigen Klammern stehen.
Anschließend wird der zusammenhängende Fließtext aufgeteilt. Dazu wird die Funktion `splitTextToChapters()` aufgerufen. Diese Funktion iteriert über alle Zeilen des Fließtextes. Dabei wird untersucht, ob die Worte des Textes in der Schriftart der Überschrift *Abstract* oder in der Schriftart der Überschrift *References* geschrieben sind. Trifft dies zu, wird der Text als Teilüberschrift und Beginn eines Unterkapitels markiert. Der nachfolgende Text bis zur nächsten Teilüberschrift wird als Inhalt dieses Unterkapitels markiert. Die Funktion speichert außerdem Titel, Autoren, Abstract und jedes Kapitel des wissenschaftlichen Artikels in eine XML-Datei, deren Dateiname dem Dateinamen des PDF-Dokuments entspricht. Die Struktur der XML-Datei ist in Quelltext 3.1 zu sehen.

```
<paper>
  <title>Titel</title>
  <authors>Autoren</authors>
  <abstract>Abstract</abstract>
  <chapter>
    <header>Teilueberschrift</header>
    <content>Inhalt</content>
  </chapter>
  <chapter>
    <header>...</header>
    <content>...</content>
  </chapter>
  ...
</paper>
```

Quelltext 3.1: Struktur einer XML-Ausgabedatei

3.3. Datentypen und Datenstrukturen

Um die Daten zu organisieren, wurden einige Datentypen implementiert. Da jeder Buchstabe x- und y-Koordinaten besitzt, wurde z. B. ein Typ *Coordinate*, der die x- und y-Werte verwaltet, implementiert. Der Datentyp *Coordinate* implementiert die Klasse *Comparable*. Dabei gilt: $P(x_P|y_P) < Q(x_Q|y_Q)$, wenn

$$y_P < y_Q$$
oder
$$y_P = y_Q \wedge x_P < x_Q.$$

Im wesentlichen verwenden die Werkzeuge des Systems aber die drei Datentypen, die in den folgenden Abschnitten genannt werden.

3.3.1. CharacterWithProperties

Ein *CharacterWithProperties* beinhaltet alle Eigenschaften, die ein Buchstabe des Textes hat. Diese sind ein eindeutiger Identifikator *id* und eine *wordId*, die eine Zuordnung zu einem Wort darstellt. Eine Eigenschaft ist auch die Seite *page* auf welcher der Buchstabe zu finden ist. Alle Schrifteigenschaften (Schriftart *font*, Schriftgröße *fontSize*, Farbschema *colorScheme* und Farbe *color*) sind ebenso enthalten. Außerdem beinhaltet der Datentyp die Koordinaten *coordinate*, in Form des Datentyps *Coordinate* und natürlich den Buchstaben selbst als *String*. Der Datentyp implementiert die Klasse *Comparable*, indem die Koordinaten des Buchstabens miteinander verglichen werden.

3.3.2. WordWithProperties

Der Typ *WordWithProperties* enthält eine Liste von *CharacterWithProperties*, die zu einem Wort gehören. Ebenso beinhaltet er einen eindeutigen Identifikator *wordId* und die Seite des Wortes *page*. Außerdem sind in diesem Datentyp die minimalen Koordinaten *minX*, *minY* und die maximalen Koordinaten *maxX*, *maxY* der Buchstaben gespeichert. Mit der Funktion `addCharacter()` kann ein weiterer *CharacterWithProperties* zum Wort hinzugefügt werden, wenn Wortidentifikator, Farbe, Farbschema, Schriftart und -größe übereinstimmen. Mit der Funktion `getWord()` kann ein *String* der Buchstaben des Wortes, in korrekter Reihenfolge, ausgegeben werden. Auf die Schrifteigenschaften wie Farbe, Schriftart und -größe, kann zugegriffen werden, indem diese aus den beinhalteten *CharacterWithProperties* abgefragt werden. Der Datentyp implementiert die Klasse *Comparable* ebenso, indem die minimalen Koordinaten zweier Worte miteinander verglichen werden. Der Datentyp *WordWithProperties* wird in zwei *HashMaps* abgelegt. Eine *HashMap* dient dem einfachen Zugriff auf ein Wort, indem der Wortidentifikator als *Key* abgelegt wird. Die andere *HashMap* dient dem einfachen Zugriff auf die Worte einer Seite, indem die Seite als *Key* und eine Liste von *WordWithProperties* als *Value* abgelegt wird.

3.3.3. WordDimensionPair

Für die Benutzung durch die Analysewerkzeuge wurde der Datentyp *WordDimensionPair* implementiert. Dieser Datentyp beinhaltet nur minimale und maximale Koordinaten eines Wortes; den Wortidentifikator des dazugehörigen Wortes und dessen *interest*-Wert. Der Typ *WordDimensionPair* implementiert ebenso die Klasse *Comparable*, indem die minimalen Koordinaten miteinander verglichen werden. Dieser Datentyp wird verwendet, um die von der Funktion `makeGrid` berechnete Ausdehnung der Worte abzuspeichern. Der Datentyp wurde im vorgestellten System nicht mit dem Datentyp *WordWithProperties* zusammengefügt, da die durchschnittliche Suchzeit nach einem *Key* in einer *HashMap* ohnehin konstant ist[Orac].

3.4. Verwendete Algorithmen

Die Werkzeuge nutzen verschiedene Algorithmen um Textblöcke auszusortieren. Die Algorithmen *findCaptions()*, *findFootnotesAlgorithm()*, *findReferences()*, *findAbstract()* und *findFigures()* sind sich im Prinzip ähnlich, deswegen wird nur der Algorithmus *findCaptions()* vorgestellt. Die Unterschiede zwischen dem vorgestellten Algorithmus und den nicht Vorgestellten wird in Kap. 3.4.6 genau erläutert.

3.4.1. calculateWords()

Algorithmus 1 erstellt aus den *CharacterWithProperties* neue *WordWithProperties*. *PDFBox* kann Worte zusammenfügen, aber einige dieser Worte sind fehlerhaft. Diese Worte beinhalten verschiedene Schriftarten, Buchstaben und Zahlen oder erstrecken sich über mehrere Zeilen. Der Algorithmus probiert in der ersten *foreach*-Schleife jeden Buchstaben den ihm zugeordneten Wort hinzuzufügen. Klappt dies nicht, weil das Wort fehlerhaft ist, wird der Buchstabe in eine temporäre *HashMap* gespeichert. Sind alle Buchstaben einer Seite das erste mal untersucht, wird so lange eine *while*-Schleife durchlaufen, bis diese temporäre *HashMap* leer ist. Innerhalb dieser Schleife wird die Liste der nicht erfolgreich zugeordneten *CharacterWithProperties* zu jedem Wortidentifikator betrachtet. Aus dem ersten *CharacterWithProperties* wird ein neues Wort erstellt und alle weiteren *CharacterWithProperties* werden, wenn möglich, in dieses Wort eingefügt. Anschließend wird diese Liste, wenn sie leer ist, aus der temporären *HashMap* entfernt. Im schlechtesten Fall hat *PDFBox* alle n Buchstaben einer Seite einem einzigen Wort zugeordnet, tatsächlich aber ist jeder Buchstabe der Seite ein eigenes Wort. In diesem Fall wird in der ersten Schleife genau ein Buchstabe entfernt. Die anschließende *while*-Schleife wird zu Beginn eine Liste mit den $n-1$ verbleibenden Buchstaben beinhalten. Jeder dieser $n-1$ Buchstaben wird betrachtet werden und einer entfernt werden. Anschließend wird die Liste noch $n-2$ Buchstaben beinhalten, über die wieder iteriert werden muss. Somit werden also insgesamt n Schleifendurchläufe benötigt, bei denen n, dann $n-1$, dann $n-2$, etc. bis zu 1 Buchstaben betrachtet werden. Mit Hilfe der Gaußschen Summenformel[5] lässt sich somit eine Laufzeit $\mathcal{O}(n^2)$ errechnen. Dieser Fall wird jedoch in der Realität kaum auftreten, da sowohl der Algorithmus zum Finden von Worten von *PDFBox* gut funktioniert, als auch kein wissenschaftlicher Artikel aus Worten besteht, die jeweils nur aus einem einzigen Zeichen bestehen.

3.4.2. makeHorizontalLines()

Algorithmus 2 versucht eine Liste von *WordDimensionPairs* horizontal in Blöcke aufzuteilen. Es kann eine Anzahl an Versuchen angegeben werden, diese Anzahl ist

[5] $1+2+3+\cdots+n = \sum_{k=1}^{n} k = \frac{n(n+1)}{n} = \frac{n^2+n}{2}$

```
Data : HashMap, die alle CharacterWithProperties beinhaltet. Key ist der
       jeweilige Identifikator.
Result : HashMap, die alle WordWithProperties beinhaltet. Key ist jeweils der
         Wortidentifikator eines Wortes.
Ausgabe ⟵ Leere-HashMap<Long, Liste von WordWithProperties>;
foreach Seite ∈ Dokument do
    tempCharacters ⟵ Leere-HashMap<Long, Liste von
    CharacterWithProperties>;
    foreach CharacterWithProperties c auf Seite do
        if c.getWordID() ∉ Ausgabe then
            w ⟵ neues WordWithProperties;
            Ausgabe ⟵ Ausgabe ∪ w;
            w.addCharacter(c);
        else
            w ⟵ Ausgabe.get(c.getWordID());
            if !w.addCharacter(c) then
                tempCharacters ⟵ tempCharacters∪Paar aus w.getWordID()
                und w;
            end
        end
    end
    while tempCharacters ≠ ∅ do
        foreach Liste von CharacterwithProperties l ∈ tempCharacters do
            w ⟵ neues WordWithProperties;
            Ausgabe ⟵ Ausgabe∪ Paar aus w.getWordID() und w;
            w.addCharacter(l.first());
            l.remove(l.first());
            foreach CharacterwithProperties c ∈ l do
                if w.addCharacter(c) then
                    l.remove(c);
                end
            end
        end
        if l = ∅ then
            tempCharacters ⟵ tempCharacters \ l;
        end
    end
end
return Ausgabe;
```

Algorithmus 1 : Der Algorithmus *calculateWords()* fügt die von *PDFBox* gefundenen Worte als *WordWithProperties* zusammen. Dabei wird zuerst jeder Buchstabe in das zugeordnete Wort eingefügt. Ist dies nicht möglich, weil z. B. die Schriftart abweicht oder der Buchstabe an einer ganz anderen Position ist, wird ein neues Wort erstellt. Wurde zu einem gefundenen Wort schon ein neues Wort erstellt, wird versucht den Buchstaben in dieses Wort einzufügen.

die Differenz aus maximaler und minimaler Y-Koordinate. Dazu iteriert der Algorithmus jeweils über die Liste von *WordDimensionPairs*. Wenn keines der Worte auf der vorher berechneten Trennlinie liegt, werden alle Worte oberhalb dieser Linie in die Ausgabe-*HashMap* eingefügt. Ist die maximale Anzahl an Blöcken erreicht, werden die verbleibenden Worte mit dem *Key* 10.000[6] in die Ausgabe-*HashMap* eingefügt. Der Algorithmus hat für eine Eingabeliste der länge n eine Laufzeit von $\mathcal{O}(n)$, da die Anzahl der Iterationen über die Wortliste eine Konstante ist.

3.4.3. makeVerticalLines()

Algorithmus 3 versucht eine Liste von *WordDimensionPairs* nahe der Mitte vertikal zu teilen. Zuerst wird eine Trennlinie in der Mitte erstellt. Anschließend iteriert der Algorithmus über die Liste von *WordDimensionPairs*. Wenn keines der Worte auf der vorher berechneten Trennlinie liegt, werden alle Worte links dieser Linie in die Ausgabe-*HashMap* eingefügt. Alle verbleibenden Worte werden mit dem *Key* 10.000 in die Ausgabe-*HashMap* eingefügt. Die Trennlinie wird bei Misserfolg abwechselnd links und rechts der Mitte neu positioniert. Der Algorithmus hat eine Laufzeit von $\mathcal{O}(n)$, wobei n die Anzahl der Worte in der Eingabeliste ist, da die Anzahl der Versuche eine Konstante ist.

3.4.4. makeGrid()

Einer der wichtigsten Algorithmen des Systems ist Algorithmus 4. Dieser Algorithmus teilt mit Hilfe der Algorithmen *makeVerticalLines()* und *makeHorizontalLines()* jede Seite in einzelne Blöcke auf. Die Eingabe ist eine Liste von *WordWithProperties* einer Seite. Zu Beginn wird für jedes *WordWithProperties* ein *WordDimensionPair* erstellt. Jedes *WordDimensionPair* wird horizontal und vertikal ausgedehnt, wie in Kap. 3.2.2 beschrieben. Die Ausgabe ist eine *HashMap*. Jeder *Key* der *HashMap* entspricht dabei einem Block. Diesen zugeordneten sind Listen der beinhalteten *WordDimensionPairs*. Um aus jedem *WordWithProperties* ein *WordDimensionPair* zu erstellen, werden n Berechnungen durchgeführt. Da die Algorithmen *makeVerticalLines()* und *makeHorizontalLines()* jeweils eine Laufzeit von $\mathcal{O}(n)$ für eine Listenlänge von n haben, hat der *makeGrid*-Algorithmus eine Laufzeit von $\mathcal{O}(n^3)$. In Abb. 3.1 sind zwei Seiten eines wissenschaftlichen Artikels zu sehen, die vom *makeGrid*-Algorithmus in einzelne Blöcke aufgeteilt wurden. Dabei sieht man, dass auf beiden Seiten das zweispaltige Layout berücksichtigt wurde. Titel, Abstract, zwei Tabellen und mehrere Fußnoten wurden jeweils in einzelne Blöcke aufgeteilt.

[6] Der Wert 10.000 ist willkürlich gewählt. Die maximale y-Koordinate liegt bei etwa 800, alle verbleibenden Worte werden also in einen gemeinsamen Block nach dem letzten gefundenen Block eingefügt.

```
Data : Liste von WordDimensionPairs L, Blockgröße, Anzahl an Blöcken B
Result : HashMap, mit y-Koordinaten und WordDimensionPairs oberhalb der
         jeweiligen y-Koordinate
Ausgabe ⟵ leere Map<Integer, Liste von WordDimensionPairs>;
for i ← 1 to B do
    V ⟵ ∅;
    nichtGebrochen ⟵ true;
    zeile ⟵ i · Blockgröße;
    foreach word ∈ L do
        y_min ⟵ word.getYmin();
        y_max ⟵ word.getYmax();
        if y_min < zeile ∧ y_max > zeile then
            nichtGebrochen ⟵ false;
            break;
        end
        if y_max < zeile then
            V ∪ word;
        end
    end
    if nichtGebrochen then
        Ausgabe ⟵ Ausgabe∪ Paar aus Zeile und V;
        L ⟵ L \ V;
        V ⟵ ∅;
    end
end
Ausgabe ⟵ Ausgabe∪ Paar aus 10000 und L;
return Ausgabe;
```

Algorithmus 2 : Der *makeHorizontalLines()*-Algorithmus teilt eine Liste von *WordDimensionPairs* in horizontale Blöcke. Die Ausgabe ist eine *HashMap*. Jeder *Key* ist die maximale y-Koordinate eines Blocks. Diesem zugeordnete ist jeweils eine Liste von *WordDimensionPairs*, die oberhalb dieser y-Koordinate liegen.

```
Data : Liste von WordDimensionPairs L, Versuche, Breite
Result : HashMap, mit x-Koordinaten und WordDimensionPairs links der
         jeweiligen x-Koordinate
Ausgabe ⟵ leere HashMap<Integer, Liste>;
nichtGebrochen ⟵ true;
for i ← 0 to Versuche − 1 do
    if i mod 2 = 0 then
        spalte ⟵ Breite/2 + i;
    end
    spalte ⟵ Breite/2 − i;
    V ⟵ ∅;
    foreach word ∈ L do
        x_min ⟵ word.getXmin();
        x_max ⟵ word.getXmax();
        if x_min < spalte ∧ x_max > spalte then
            nichtGebrochen ⟵ false;
            break;
        end
        if x_max < spalte then
            V ∪ word;
        end
    end
    if nichtGebrochen then
        Ausgabe ⟵ Paar aus spalte und V;
        L ⟵ L \ V;
        V ⟵ ∅;
        break;
    end
end
Ausgabe ⟵ Ausgabe∪ Paar aus 10000 und L;
return Ausgabe;
```

Algorithmus 3 : Der *makeVerticalLines()*-Algorithmus versucht eine Liste von *WordDimensionPairs* in der Mitte vertikal zu teilen. Die Ausgabe ist eine *HashMap*. Jeder *Key* ist die maximale x-Koordinate eines Blocks, zugeordnet ist jeweils eine Liste von *WordDimensionPairs*, die links dieser x-Koordinate liegen.

```
Data : Liste von WordWithProperties L, minimal erlaubte Größe von Blöcken m
Result : HashMap, mit Listen von WordDimensionPairs
Ausgabe ⟵ leere HashMap<int, Liste von Listen von WordDimensionPairs>;
horizontaleMap1 ⟵ makeHorizontalLines(L);
foreach Key in horizontaleMap1 do
    Liste1 ⟵ horizontaleMap1.get(Key);
    if Liste1 < m then
        Liste1 ⟵ Liste1∪ Nachbarliste;
        horizontaleMap1 ⟵ horizontaleMap1\ Nachbarliste;
    end
    foreach Key in horizontaleMap1 do
        Liste2 ⟵ horizontaleMap1.get(Key);
        vertikaleMap1 ⟵ makeVerticalLines(Liste2);
        foreach Key in vertikaleMap1 do
            Liste3 ⟵ vertikaleMap1.get(Key);
            if Liste3 < m then
                Liste3 ⟵ Liste3∪ Nachbarliste;
                vertikaleMap1 ⟵ vertikaleMap1\ Nachbarliste;
            end
            foreach Key in vertikaleMap1 do
                Liste4 ⟵ vertikaleMap1.get(Key);
                horizontaleMap2 ⟵ makeHorizontalLines(Liste4);
                foreach Key in horizontaleMap2 do
                    Liste5 ⟵ horizontaleMap2.get(Key);
                    if Liste5 < m then
                        Liste5 ⟵ Liste5∪ Nachbarliste;
                        horizontaleMap2 ⟵ horizontaleMap2\ Nachbarliste;
                    end
                    foreach Key in horizontaleMap2 do
                        Liste6 ⟵ horizontaleMap2.get(Key);
                        Ausgabe ⟵ Liste6;
                    end
                end
            end
        end
    end
end
return Ausgabe;
```

Algorithmus 4 : Der *makeGrid()*-Algorithmus teilt mit Hilfe der Algorithmen *makeVerticalLines()* und *makeHorizontalLines()* jede Seite in einzelne Blöcke auf. Das Ergebnis ist eine *HashMap*, deren *Keys* jeweils einem Blöck entsprechen. Diesen zugeordnet sind Listen von *WordDimensionPairs*.

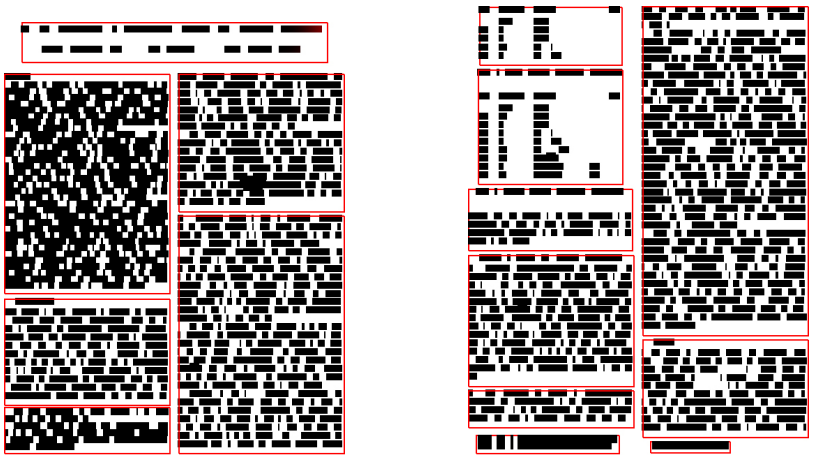


Abbildung 3.1.: Zu sehen sind zwei Seiten eines Artikels, mit Hilfe des MakeGrid-Algorithmus in Teilbereiche zerlegt wurden.

3.4.5. calculateUppercaseAndNumbers()

Der Algorithmus 5 untersucht Blöcke auf eine hohe Anzahl an Großbuchstaben und Zahlen. Dazu wird für jede Seite über jeden Block iteriert. Ist das Wort eine Zahl, wird ein Zähler erhöht. Beginnt das Wort mit einem Großbuchstaben, wird ein anderer Zähler erhöht. Anschließend werden die Durchschnittswerte, gemessen an der Anzahl der Worte des Blocks, berechnet. Sind beide Durchschnittswerte höher, als der 1,2-fache Durchschnittswert aller Blöcke, werden die Worte dieses Blocks aussortiert. Der Algorithmus muss über alle n Worte iterieren, um jeweils die Zähler zu erhöhen. Anschließend muss über einen Teil der n Worte iteriert werden, um den *interest* von *false* zu setzen. Vor dem eigentlichen Algorithmus muss noch einmal über alle Worte iteriert werden, um die Durchschnittswerte initial zu berechnen. Somit wird maximal dreimal über alle n Worte iteriert, somit wird eine Laufzeit von $\mathcal{O}(n)$ erreicht.

3.4.6. findCaptions()

Der Algorithmus 6 durchsucht alle Blöcke auf die Schlüsselworte “Table”, “Figure” und “Algorithm”. Dazu iteriert er über die Worte aller Blöcke. Wird eines der Schlüsselworte gefunden, werden die Koordinaten zwischengespeichert. Anschließend wird über alle Worte des Blocks erneut iteriert. Befindet sich in der gleichen Zeile noch ein Doppelpunkt rechts des Schlüsselwortes, wird erneut über den Block iteriert und jedes Wort mit dem *interest* von *false* markiert. Da maximal dreimal über alle Worte iteriert wird, ergibt sich eine Laufzeit von $\mathcal{O}(n)$.
Die Algorithmen *findAbstract()*, *findReferences() findFootnotesAlgorithm()*, und *findFigures()* sind diesem Algorithmus sehr ähnlich. Die Algorithmen *findAbstract()* und

Data : *HashMap* die Liste von Listen von *WordDimensionPair* beinhaltet, durchschnittliche Anzahl an Zahlen pro Block Z, durchschnittliche Anzahl an Großbuchstaben pro Block G
Result : *HashMap*, Blöcke mit hoher Anzahl von Zahlen und hoher Anzahl von Großbuchstaben sind mit *interest false* markiert

```
Ausgabe ⟵ Eingabe;
foreach Seite do
    foreach Liste auf Seite do
        if Liste ≠ ∅ then
            TempZahlen ⟵ 0;
            TempGroßbuchstaben ⟵ 0;
            Anzahl ⟵ 0;
            foreach WordDimensionPair w ∈ Liste do
                Anzahl ⟵ Anzahl + 1;
                if w ist Zahl then
                    TempZahlen ⟵ TempZahlen + 1;
                end
                if w beginnt mit Großbuchstabe then
                    TempGroßbuchstaben ⟵ TempGroßbuchstaben + 1;
                end
            end
            TempZahlen ⟵ TempZahlen / Anzahl;
            TempGroßbuchstaben ⟵ TempGroßbuchstaben / Anzahl;
            if Tempzahlen ≥ Z * 1.2 ∧ TempGroßbuchstaben ≥ G * 1.2 then
                foreach WordDimensionPair w ∈ Liste do
                    w.Interest ⟵ false;
                end
            end
        end
    end
end
return Ausgabe;
```

Algorithmus 5 : Die Funktion `calculateUppercaseAndNumbers()` entfernt Blöcke, deren Worte besonders viele Großbuchstaben und Zahlen beinhalten. Dazu wird für jeden Block die Anzahl an Großbuchstaben und Zahlen pro Block gezählt. Sind diese Werte höher als der jeweilige 1,2-fache Durchschnittswert aller Blöcke, werden die Worte in diesem Block mit dem *interest* von *false* markiert.

findReferences() durchsuchen die Blöcke nur nach den Schlüsselworten "Abstract", bzw. "References". Befinden sich die Schlüsselworte zu Beginn des Blockes, wird der Block aussortiert. Der Abstract wird nur auf der ersten Seite gesucht und anschließend auch zurückgegeben, die Quellenangaben werden zuerst auf der letzten Seite gesucht, werden sie dort nicht gefunden werden, werden die Seiten Rückwärts durchsucht. Wird das Schlüsselwort gefunden, werden alle nachfolgenden Blöcke ebenfalls aussortiert. Für beide Algorithmen ergibt sich ebenfalls eine Laufzeit von $\mathcal{O}(n)$. Der Algorithmus *findFigures()* durchsucht alle Blöcke nach dem Schlüsselwort "Figure". Blöcke, die das Wort beinhalten, werden mit der Java-Funktion *Collections.Sort()* in $\mathcal{O}(n \log n)$ [Oraa] sortiert. Anschließend wird der sortierte Block daraufhin untersucht, ob sich nach dem Schlüsselwort zuerst bis zu 4 beliebige Zeichen, gefolgt von einem Doppelpunkt befinden. Danach wird über alle Worte des Blockes iteriert, und diese mit dem *interest* von *false* markiert. Für diesen Algorithmus ergibt sich also eine Laufzeit von $\mathcal{O}(n \log n)$. Der Algorithmus *findFootnotesAlgorithm()* untersucht, ob jeder Block mit einem der Schlüsselworte "Algorithm", "Keywords" oder "Categories" beginnt. Wenn dies zutrifft, wird jedes Wort des Blocks mit dem *interest* von *false* markiert. Beginnt ein Block nicht mit einem der Schlüsselworte, wird überprüft, ob die Schriftart des ersten Wortes des Blocks kleiner ist, als die meistverwendete Schriftart. Wenn dies zutrifft, wird dies für jedes weitere Wort des Blocks überprüft. Anschließend wird jedes Wort mit dem *interest* von *false* markiert. Auch hier ergibt sich eine Laufzeit von $\mathcal{O}(n)$.

3.4.7. extractAuthorAndTitle()

Der Algorithmus 7 extrahiert Autoren und Titel eines wissenschaftlichen Artikels. Dazu wird zuerst die erste Seite des wissenschaftlichen Artikels durch eine modifizierte Version des in Kap. 3.4.4 dargestellten Algorithmus in Blöcke aufgeteilt. Bei dieser Version wird eine Seite nur horizontal in drei Bereiche aufgeteilt. Die vertikale Ausdehnung der Wörter ist in dieser Version geringer und kleine Blöcke werden nicht mit den Nachbarblöcken zusammengelegt. Die Ausgabe-*HashMap* dieser Funktion beinhaltet zwei Schlüssel, die den Blöcken Titel und Autoren entsprechen. Der Titel kann in eine Variable gespeichert werden, indem auf den ersten Schlüssel der Ausgabe zugegriffen wird. Auf die Autoren kann ebenso zugegriffen werden. Die anschließende Sortierung wird mit der Java-Funktion *Collections.Sort()* durchgeführt. Sie verwendet einen modifizierten *MergeSort*-Algorithmus und hat eine garantierte Laufzeit von $\mathcal{O}(n \log n)$. Somit ergibt sich für diesen Algorithmus die gleiche Laufzeit.

```
Data : HashMap, Key ist Seitenangabe Value ist Liste von Listen von
       WordDimensionPairs
Result : Die gleiche HashMap, Wörter aus Bildunterschriften sind mit interest
         von false markiert
Ausgabe ⟵ Eingabe;
foreach Key in Ausgabe do
    foreach Liste in Value do
        if Liste ≠ ∅ und interest = true then
            foreach WordDimensionPair w in Liste do
                if w ∈ { "Table", "Figure", "Algorithm" } then
                    start ⟵ true;
                    startWort ⟵ w;
                    x_start ⟵ w.getXmin();
                    y_start ⟵ w.getYmin();
                end
            end
            if start = true then
                x ⟵ ∞;
                zweitesWort ⟵"";
                foreach WordDimensionPair w in Liste do
                    if w ≠ startWort ∧ w.getYmin() = y_start ∧ w.getXmax()
                    > x_start ∧ w.getXmax() < x then
                        x ⟵ w.getXmax();
                        zweitesWort ⟵ w;
                    end
                end
                if zweitesWort = ":" then
                    foreach WordDimensionPair w in Liste do
                        w.interest ⟵ false;
                    end
                end
            end
        end
    end
end
return Ausgabe;
```

Algorithmus 6 : Der Algorithmus *findCaptions()* durchsucht den Text nach Bildunterschriften, die die Schlüsselworte "Table", "Figure" oder "Algorithm" beinhalten. Die Worte werden mit dem *interest* von *false* markiert, wenn auf das gefundene Schlüsselwort ein Doppelpunkt folgt.

Data : *Eingabe-HashMap* aller nicht aussortierten Worte
Result : *Titel* und *Autoren*
Titel ⟵“”;
Autoren ⟵“”;
TempMap ⟵ makeFirstPageGrid(ersteSeite aus *Eingabe*);
Titel ⟵ *TempMap*.get(0);
sortiere *Titel*;
Autoren ⟵ *TempMap*.get(1);
sortiere *Autoren*;
return *Titel*, *Autoren*;

Algorithmus 7 : Der *extractAuthorAndTitle()*-Algorithmus extrahiert Titel und Autoren aus der ersten Seite. Dazu wird ein modifizierter *makeGrid*-Algorithmus gestartet, der die Seite nur in horizontale Blöcke teilt. Der Schlüssel “0” entspricht dem obersten Block, also dem Titel. Der Schlüssel “1” entspricht dem darunterliegenden Block mit den Autorenangaben.

3.4.8. findSmallerBlocks()

Algorithmus 8 entfernt Blöcke aus dem Text, deren durchschnittliche Zeilenbreite sich deutlich vom Mittelwert unterscheiden. Dazu wird zunächst der *makeGrid*-Algorithmus erneut ausgeführt. Die Werte für die Ausdehnung der Worte wurden so geändert, dass die Blöcke insgesamt größer werden. Gefundene sehr kleine Blöcke werden aber nicht mit den Nachbarblöcken zusammengefügt. Jeder gefundene Block wird untersucht. Dazu wird zuerst die Liste mit der Java-Funktion *Collections.Sort()* sortiert. Anschließend wird eine *HashMap* erstellt. Diese *HashMap* soll jede y-Koordinate des Blocks und die zugehörigen Worte abspeichern. Dazu wird einmal jedes Wort des Blocks in die *HashMap* eingefügt. Anschließend wird jede Zeile des Blocks untersucht und die minimalen und maximalen x-Koordinaten gesucht. Die Differenz wird zwischengespeichert. Sind alle Zeilen abgearbeitet, wird der Mittelwert der Differenzen D_{zeilen} bestimmt und der Block zusammen mit diesem Mittelwert in eine weitere *HashMap* gespeichert. Dieser Mittelwert der Differenzen wird zu D_{ges} addiert. Nach dem Untersuchen aller Blöcke wird D_{ges} durch die Anzahl der Blöcke geteilt, um den Mittelwert zu berechnen. Anschließend wird für jeden Block D_{zeilen} mit D_{ges} verglichen. Ist D_{zeilen} größer als das 1,5-fache von D_{ges} oder kleiner als 90% von D_{ges} und beinhaltet in diesem Fall weniger als 50 Worte, wird der Block mit dem *interest* von *false* markiert und somit aussortiert. Dieser Algorithmus sortiert Bereiche aus, die in einem zweispaltigen Layout eines Artikels größer als eine Spalte sind. Dies trifft z. B. auf einige Bildunterschriften von breiten Grafiken oder Tabellen zu. Ebenso werden einzelne bisher nicht gefundene Tabellenzeilen von diesem Algorithmus erkannt und aussortiert. Bei n Worten hat der aufgerufene *makeGrid*-Algorithmus eine asymptotische Laufzeit von $\mathcal{O}(n^3)$. Das Sortieren jeder Liste benötigt zusammen $\mathcal{O}(n \log n)$. Da alle Schritte nacheinander durchgeführt werden, ergibt sich eine Laufzeit von $\mathcal{O}(n^3)$, bei n Worten.

3.4.9. makeOnePage()

Algorithmus 9 sortiert alle Worte, um sie für die Ausgabe vorzubereiten. Dazu wird für jede Seite die maximale Ausdehnung in x- und y-Richtung bestimmt, indem für jedes Wort dieser Seite die Koordinaten betrachtet werden. Anschließend wird mit der Funktion `makeVerticalMiddleLine()` eine modifizierte Version des *makeVerticalLines()*-Algorithmus aufgerufen. Diese Version teilt eine Seite wenn möglich vertikal in zwei Blöcke auf. Dazu wird auf der Seite bei der x-Koordinate $\frac{x_{max}-x_{min}}{2}$ eine Linie erstellt. Liegt keines der Worte auf der Linie werden die Worte links der Linie in einen Block eingeteilt. Rechts der Linie werden sie in den anderen Block eingeteilt. Ist eine Teilung nicht möglich, werden alle Worte dem gleichen Block zugeordnet. Der Block, bzw. die Blöcke werden in eine *HashMap* abgelegt. Dabei erhält ein linker Block auf Seite s den Schlüssel $key_l = s \cdot 10 + 1$, während der rechte Block den Schlüssel $key_r = s \cdot 10 + 2$ erhält. Sind alle Blöcke der Seiten in die *HashMap* eingefügt, wird aus den k Schlüsseln ein *TreeSet* erstellt. Die benötigte Zeit hierfür ist garantiert $\mathcal{O}(k \log k)$, da k-mal eine Einfügeoperation stattfindet[Orab]. Damit sind die Schlüssel sortiert und die Lesereihenfolge der Blöcke gewährt. Anschließend wird der Inhalt aller Blöcke mit der Funktion *Collections.Sort()* sortiert und in eine Ausgabeliste eingefügt. Die Funktion `makeVerticalMiddleLine()` iteriert einmal über alle Worte n. Die Sortierung aller Worte benötigt $\mathcal{O}(n \log n)$. Da für n Worte und k Schlüssel $k \leq n$ gilt, ergibt sich somit eine Laufzeit von $\mathcal{O}(n \log n)$.

3.4.10. removeFootnoteSign()

Algorithmus 10 entfernt aus dem Fließtext die Anmerkungsziffern. Dazu wird jedes Wort entsprechend der y-Koordinate in eine *HashMap* eingefügt. Anschließend wird für jede Liste von *WordDimensionPairs*, die einer y-Koordinate zugeordnet sind, überprüft, ob diese Liste aus höchstens 5 Worten besteht. Trifft dies zu, wird für jedes dieser Worte überprüft, ob es aus höchstens 3 Zeichen besteht. Trifft diese Bedingung auf alle Worte dieser Zeile zu, werden die Worte aus der Ausgabeliste entfernt. Die Funktion iteriert nur über alle n Worte. Anschließend iteriert sie über $k \leq n$ Schlüssel der *HashMap*. Für jedes der höchstens n Worte, muss das entsprechende Wort in der Liste in höchstens n Schritten gesucht werden. Somit ergibt sich eine Laufzeit von $\mathcal{O}(n^2)$.

Data : Liste von WordWithProperties L
Result : *HashMap*, mit Listen von *WordDimensionPairs*
$Ausgabe \longleftarrow$makeGrid(L);
$D_{ges} \longleftarrow 0$;
$DurchschnittMap \longleftarrow$ leere *HashMap*;
$AnzahlZeilen \longleftarrow 0$;
foreach $Key\ in\ Ausgabe$ **do**
  foreach $Liste\ in\ Value$ **do**
    if $Liste \neq \emptyset\ und\ interest = true$ **then**
      sortiere Liste;
      $ZeilenMap \longleftarrow$ *HashMap*<*Zeile*, Liste von $WordDimensionPairs$>
      foreach $WordDimensionPair\ w\ in\ Liste$ **do**
        füge w in $ZeilenMap$.Keys ein;
      end
      foreach $Zeile\ in\ ZeilenMap$ **do**
        foreach $WordDimensionPair\ w\ in\ Zeile$ **do**
          finde x_{min}, x_{max};
          $D_{zeilen} \longleftarrow D_{zeilen} + (x_{max} - x_{min})$;
        end
        $D_{zeilen} \longleftarrow \frac{D_{zeilen}}{AnzahlderZeilenimBlock}$;
        $DurchschnittMap \longleftarrow DurchschnittMap \cup$ Paar aus D_{zeilen} und Block;
        $D_{ges} \longleftarrow D_{ges} + D_{zeilen}$;
        $AnzahlZeilen \longleftarrow AnzahlZeilen + 1$;
      end
    end
  end
end
$D_{ges} \longleftarrow \frac{D_{ges}}{AnzahlZeilen}$;
foreach $Block\ in\ DurchschnittMap$ **do**
  if $(D_{zeilen} > \frac{3}{2} * D_{ges}) \vee (D_{zeilen} < \frac{9}{10} * D_{ges} \wedge Blockgröße < 50)$ **then**
    foreach $WordDimensionPair\ w\ in\ Block$ **do**
      $interest \longleftarrow$ false;
    end
  end
end
return *Ausgabe*

Algorithmus 8 : Der *findSmallerBlocks()*-Algorithmus findet Blöcke, deren Breite deutlich über dem Durchschnitt oder darunter liegen. Dazu wird für jeden Block die Breite jeder Zeile berechnet. Anschließend wird der Durchschnittswert für jeden Block gebildet. Anhand dieser Durchschnittswerte wird entschieden, ob der *interest* von *false* gesetzt wird.

Data : Liste aller interessanten *WordWithProperties*
Result : Nach Koordinaten sortierte Liste aller *WordDimensionPairs*
$Ausgabe \longleftarrow \emptyset$;
$Map \longleftarrow$ leere *HashMap*;
foreach *Seite* **do**
  $SeitenListe \longleftarrow \emptyset$;
  foreach $WordWithProperties\ w \in Eingabe$ **do**
    if $w.getPage() = Seite$ **then**
      berechne $x_{min}, y_{min}, x_{max}, y_{max}$;
      $SeitenListe = SeitenListe \cup w$;
    end
    $Temp \longleftarrow$makeVerticalMiddleLine($SeitenListe, x_{min}, y_{min}, x_{max}, y_{max}$);
    $Map \longleftarrow Map \cup$ Paar aus $Seite \cdot 10 + Temp.Key$ und $Temp.Value$;
  end
end
sortiere $Keys$ von Map;
foreach $Key\ in\ Map$ **do**
  sortiere zu Key gehörige *Liste*;
  $Ausgabe \longleftarrow Ausgabe \cup Liste$;
end
return *Ausgabe*;

Algorithmus 9 : Der *makeOnePage*-Algorithmus sortiert alle Worte, die nicht vorher durch die Werkzeuge aussortiert wurden. Dazu werden die Worte jeder Seite mit der Funktion `makeVerticalMiddleLine()` wenn möglich vertikal geteilt. Diese Funktion entspricht dem in Algorithmus 3 beschriebenen Algorithmus, der einzige Unterschied besteht darin, dass höchstens zwei Blöcke erstellt werden. Sobald eine vertikale Teilung durchgeführt wurde, werden alle verbleibenden Worte in den rechten Block eingeordnet. Anschließend werden die Worte der Blöcke sortiert und hintereinandergehängt.

```
Data : Liste der nach Koordinaten sortierten WordDimensionPairs
Result : sortierte Liste der WordDimensionPairs ohne Anmerkungsziffern
Ausgabe ⟵ Eingabe;
ZeilenMap ⟵ HashMap, Key ist y-Koordinate, Value ist Liste von
WordDimensionPairs;
foreach WordDimensionPair w ∈ Ausgabe do
    ZeilenMap ⟵ ZeilenMap∪ Paar aus w.getY() und w;
end
foreach Key in ZeilenMap do
    if Value beinhaltet höchstens 5 Worte then
        Temp ⟵ ∅;
        foreach WordDimensionPair w ∈ Value do
            if w besteht aus höchstens 3 Zeichen then
                Temp ⟵ Temp ∪ w;
            end
        end
        if Länge von Temp = Länge von Value then
            foreach WordDimensionPair w ∈ Value do
                w.interest ⟵ false;
            end
        end
    end
end
return Ausgabe;
```

Algorithmus 10 : Die Funktion `removeFootnoteSign()` entfernt alle im Text vorkommenden Fußnoten-Zeichen. Dazu wird jedes Wort basierend auf der y-Koordinate in eine *HashMap* eingefügt. Befinden sich in einer Zeile deutlich weniger Worte, als in den anderen, werden diese Worte daraufhin untersucht, ob alle nur aus einem oder zwei Zeichen bestehen. Trifft dies zu, werden diese Worte entfernt.

3.4.11. splitTextToChapters()

Algorithmus 11 teilt den durch Algorithmus 9 zusammengesetzten Text in einzelne Kapitel mit den zugehörigen Überschriften auf. Dazu vergleicht der Algorithmus die Schriftart jedes Wortes mit den Schriftarten der Überschriften der Abschnitte *Abstract* und *References*. Stimmt die Schriftart überein ist das gefundene Wort Teil einer Überschrift, ansonsten ist es Teil des Inhalts eines Kapitels. Wird ein Wort gefunden, das Teil des Inhalts ist, wird es an die temporäre Variable $Inhalt$ angefügt. Ist das Wort Teil einer Überschrift, wird geprüft, ob $Inhalt$ leer ist. Ist dies der Fall wird das Wort an die temporäre Variable $\ddot{U}berschrift$ angehängt. Ist dies nicht der Fall, muss das Wort Teil einer neuen Überschrift sein. Dann wird das Paar aus $\ddot{U}berschrift$ und $Inhalt$ in die Ausgabe-Datei geschrieben. Da der Algorithmus nur einmal alle Worte n auf die Schriftart überprüft, hat er eine asymptotische Laufzeit von $\mathcal{O}(n)$.

Data : Liste der nach Koordinaten sortierten *WordDimensionPairs*, Schriftarten S_A und S_R der Überschriften *Abstract* und *References*, $Titel$, $Autoren$, $Abstract$
Result : XML-Datei
schreibe $Titel$, $Autoren$ und $Abstract$ in XML;
$\ddot{U}berschrift \longleftarrow$“”;
$Inhalt \longleftarrow$“”;
foreach $WordDimensionPair\ w \in EingabeListe$ **do**
  if $w.getFont() = S_A \vee w.getFont() = S_R$ **then**
    if $Inhalt =$ “” **then**
      $\ddot{U}berschrift = \ddot{U}berschrift + w$;
    else
      schreibe $\ddot{U}berschrift$ und $Inhalt$ in XML;
      $\ddot{U}berschrift = w$;
      $Inhalt =$ “”;
    end
  else
    $Inhalt = Inhalt + w$;
  end
end
schreibe $\ddot{U}berschrift$ und $Inhalt$ in XML;

Algorithmus 11 : Die Funktion `splitTextToChapters()` teilt den sortierten Fließtext mit Teilüberschriften in einzelne Blöcke auf. Dazu wird die sortierte Liste aller Worte nach den Worten durchsucht, deren Schriftart der Schriftart der Schlüsselworte “Abstract” oder “References” entspricht. Die Ausgabe wird in eine XML-Datei gespeichert.

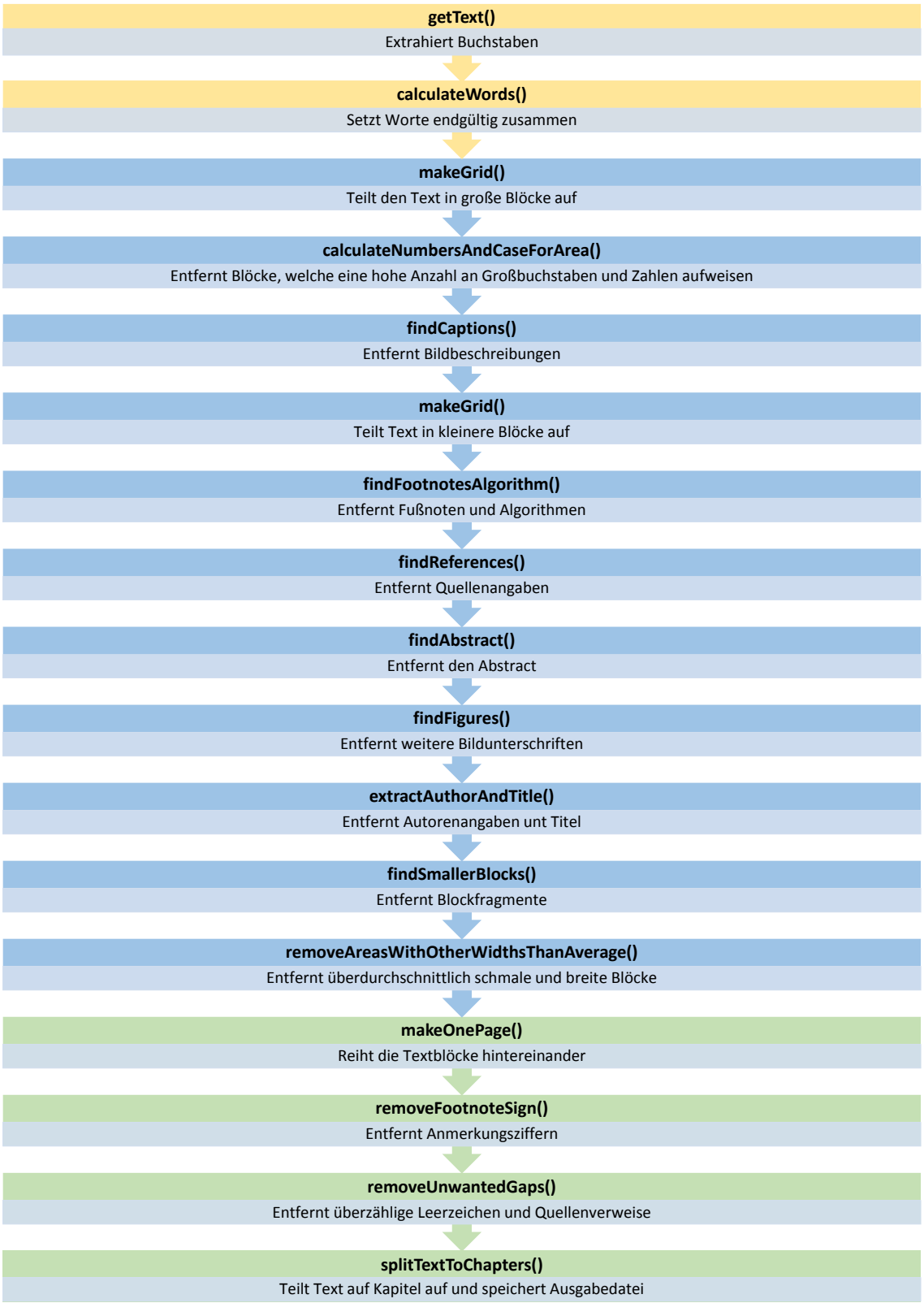


Abbildung 3.2.: Zu sehen ist der Ablauf des Systems. Die Initialisierung ist gelb, die Analyseschritte blau und die Ausgabe grün markiert.

4. Empirische Analyse

4.1. Qualität der Klassifizierung

Um die Qualität des hier vorgestellten Systems bewerten zu können, wurden zwei kleine Mengen an wissenschaftlichen Artikeln zusammengestellt und analysiert. Die erste Menge besteht aus den auf der Webseite des Lehrstuhls für Algorithmen und Datenstrukturen verfügbaren Publikationen[1]. Bis zum 19.04.2015 waren auf der Webseite 29 wissenschaftliche Artikel zum Download verfügbar. Die Artikel wurden in mehr als 20 verschiedenen Zeitschriften, Workshops, Konferenzen, etc. veröffentlicht, weshalb fast alle Artikel in anderen Layouts abgespeichert sind.
Die zweite Menge an wissenschaftlichen Artikeln besteht aus acht *PLoS Biology*-Artikeln aus vier verschiedenen Auflagen. Die Artikel unterscheiden sich im Layout untereinander kaum. Die Artikel wurden verwendet, um das in Kap. 2.1 vorgestellte System zu bewerten.

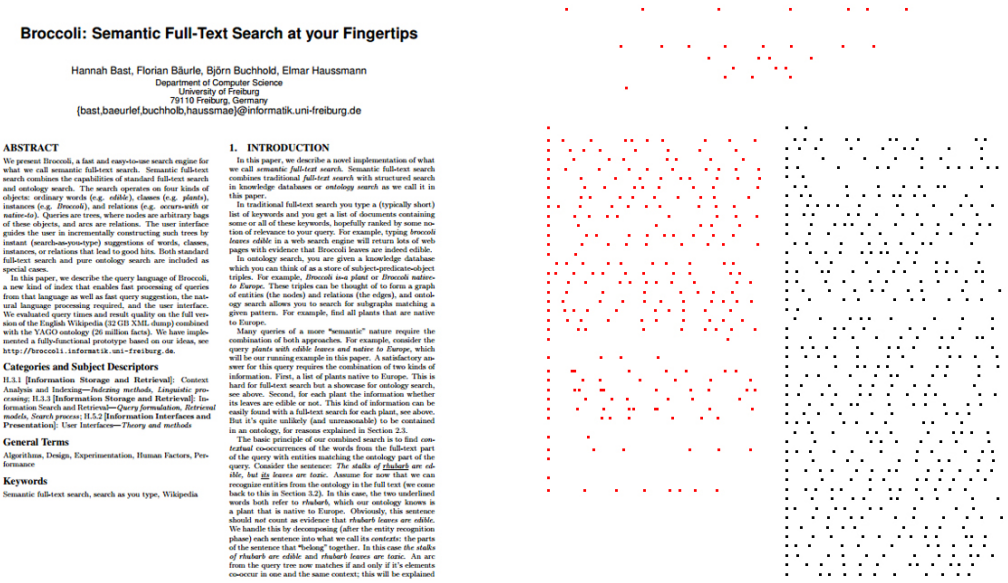
4.1.1. Methodologie

Da die Analysewerkzeuge des Systems aufeinander aufbauen und das System als Gesamtes funktionieren soll, wurde auf eine Bewertung der einzelnen Werkzeuge verzichtet. Um die Analyse bewerten zu können, muss die Liste der ausgegebenen Worte des Systems mit den tatsächlich interessanten Worten des wissenschaftlichen Artikels verglichen werden. Für diesen Zweck wurde die Klasse `MeasureQuality` erstellt. Diese Klasse ermöglicht es mit der Funktion `printPage()` alle Worte des wissenschaftlichen Artikels als Rechtecke in einer Grafikdatei abzuspeichern. Dabei wird ein Feld der Größe 3x3 Pixel an den Koordinaten des entsprechenden Wortes erstellt. Wird das Wort vom System als unwichtig erkannt und mit dem *interest* von *false* markiert, wird es in der Farbe Rot ausgegeben, andernfalls in der Farbe Schwarz. Anschließend wird die ausgegebene Grafikdatei mit dem originalen PDF-Dokument verglichen und eventuell falsch markierte Worte von Hand in einem Grafikprogramm korrekt eingefärbt. Die so bearbeitete Grafikdatei wird unter anderem Namen abgespeichert. In Abb. 4.1 ist eine Seite eines PDF-Dokuments mit der dazugehörigen Grafikdatei abgebildet. Zu sehen ist, dass das System die Seite korrekt erkannt hat.

[1] https://ad.informatik.uni-freiburg.de/

Titel, *Autoren*, *Abstract*[2] und die Abschnitte *Categories and Subject Descriptors*, *General Terms* und *Keywords* wurden korrekt vom System als nicht interessant markiert (rote Farbe). Der Abschnitt *Introduction* hingegen wurde als interessant markiert (schwarze Farbe). Anschließend können die unbearbeiteten Grafikdateien mit der Ausgabe des Systems mit den korrigierten Grafikdateien verglichen werden. Dieser Vergleich geschieht automatisch mit der Funktion `readPage()` der Klasse `MeasureQuality`, die für jede Seite eines PDF zwei Grafikdateien einliest und die Unterschiede berechnet. Hierbei werden die Worte in vier Klassen eingeteilt:

- Worte, die interessant sind und vom System als interessant markiert wurden: Klasse *richtig positiv (rp)*
- Worte, die uninteressant sind und vom System als uninteressant markiert wurden: Klasse *richtig negativ (rn)*
- Worte, die interessant sind, aber vom System als uninteressant markiert wurden: Klasse *falsch negativ (fn)*
- Worte, die uninteressant sind, aber vom System als interessant markiert wurden *falsch positiv (fp)*



Broccoli: Semantic Full-Text Search at your Fingertips

Hannah Bast, Florian Bäurle, Björn Buchhold, Elmar Haussmann
Department of Computer Science
University of Freiburg
79110 Freiburg, Germany
{bast,baeurlef,buchholb,haussmae}@informatik.uni-freiburg.de

ABSTRACT

We present Broccoli, a fast and easy-to-use search engine for what we call semantic full-text search. Semantic full-text search combines the capabilities of standard full-text search and ontology search. The search operates on four kinds of objects: ordinary words (e.g. *edible*), classes (e.g. *plants*), instances (e.g. *Broccoli*), and relations (e.g. *occurs-with* or *native-in*). Queries are trees, where nodes are arbitrary bags of these objects, and arcs are relations. The user interface guides the user in incrementally constructing such trees by instant (search-as-you-type) suggestions of words, classes, instances, or relations that lead to good hits. Both standard full-text search and pure ontology search are included as special cases.

In this paper, we describe the query language of Broccoli, a new kind of index that enables fast processing of queries from that language as well as fast query suggestion, the natural language processing required, and the user interface. We evaluated query times and result quality on the full version of the English Wikipedia (32 GB XML dump) combined with the YAGO ontology (26 million facts). We have implemented a fully-functional prototype based on our ideas, see `http://broccoli.informatik.uni-freiburg.de`.

Categories and Subject Descriptors

H.3.1 [Information Storage and Retrieval]: Content Analysis and Indexing—*Indexing methods*, *Linguistic processing*; H.3.3 [Information Storage and Retrieval]: Information Search and Retrieval—*Query formulation*, *Retrieval models*, *Search process*; H.5.2 [Information Interfaces and Presentation]: User Interfaces—*Theory and methods*

General Terms

Algorithms, Design, Experimentation, Human Factors, Performance

Keywords

Semantic full-text search, search as you type, Wikipedia

1. INTRODUCTION

In this paper, we describe a novel implementation of what we call *semantic full-text search*. Semantic full-text search combines traditional *full-text search* with structured search in knowledge databases or *ontology search* as we call it in this paper.

In traditional full-text search you type a (typically short) list of keywords and you get a list of documents containing some or all of these keywords, hopefully ranked by some notion of relevance to your query. For example, typing *broccoli leaves edible* in a web search engine will return lots of web pages with evidence that Broccoli leaves are indeed edible.

In ontology search, you are given a knowledge database which you can think of as a store of subject-predicate-object triples. For example, *Broccoli is-a plant* or *Broccoli natives-to Europe*. These triples can be thought of to form a graph of entities (the nodes) and relations (the edges), and ontology search allows you to search for subgraphs matching a given pattern. For example, find all plants that are native to Europe.

Many queries of a more "semantic" nature require the combination of both approaches. For example, consider the query *plants with edible leaves and native to Europe*, which will be our running example in this paper. A satisfactory answer for this query requires the combination of two kinds of information. First, a list of plants native to Europe. This is hard for full-text search but a showcase for ontology search, see above. Second, for each plant the information whether its leaves are edible or not. This kind of information can be easily found with a full-text search for each plant, see above. But it's quite unlikely (and unreasonable) to be contained in an ontology, for reasons explained in Section 2.3.

The basic principle of our combined search is to find *contextual* cooccurrences of the words from the full-text part of the query with entities matching the ontology part of the query. Consider the sentence: *The stalks of rhubarb are edible, but its leaves are toxic.* Assume for now that we can recognize entities from the ontology in the full text (we come back to this in Section 3.2). In this case, the two underlined words both refer to *rhubarb*, which our ontology knows is a plant that is native to Europe. Obviously, this sentence should *not* count as evidence that *rhubarb leaves are edible*. We handle this by decomposing (after the entity recognition phase) each sentence into what we call its *contexts*: the parts of the sentence that "belong" together. In this case the *the stalks of rhubarb are edible* and *rhubarb leaves are toxic*. An arc from the query tree now matches if and only if it's elements cooccur in one and the same context; this will be explained

Abbildung 4.1.: Zu sehen sind zwei Seiten eines wissenschaftlichen Artikels. Auf der linken Seite ist Seite 1 des Artikels als PDF dargestellt. Auf der rechten Seite ist die vom System ausgegebene Grafikdatei. Die Worte wurden korrekt in uninteressant (rot) und interessant (schwarz) eingeteilt.

[2] *Titel*, *Autoren* und *Abstract* werden vom System als nicht interessant markiert, aber in entsprechende Variablen gespeichert. Siehe Kap. 3.4.6 und Kap. 3.4.7.

4.1.2. Publikationen des Lehrstuhls für Algorithmen und Datenstrukturen

Die 29 PDF-Dokumente auf der Webseite des Lehrstuhls für Algorithmen und Datenstrukturen haben zusammen etwa 480.000 Worte auf 364 Seiten. In Tab. 4.1 sind die Ergebnisse für diese Artikel aufgelistet. Jedem wissenschaftlichen Artikel wurde eine laufende Nummer zugeordnet. Diese Nummern sind im Anhang in Kapitel Kap. A.1 aufgeschlüsselt. Im Durchschnitt konnte eine Präzision[3] von 0,99 erreicht werden, die Sensitivität[4] lag bei 0,978. Daraus lässt sich ein F_1-Wert[5] von 0,984 errechnen. Die wissenschaftlichen Artikel mit den Nummern 14 ($F_1 = 0,82$), 16 ($F_1 = 0,935$), 18 ($F_1 = 0,932$) und 23 ($F_1 = 0,938$) heben sich negativ von den anderen Artikeln ab.
Bei Artikel 16 wurde auf der ersten Seite am linken unteren Rand eine Copyrightangabe nicht entfernt. Auf der zweiten Seite wurden die drei Tabellen auf der rechten Seite nicht aussortiert, da für den `makeGrid()`-Algorithmus der Abstand zwischen den Textblöcken und den Tabellen zu gering war. In Artikel 18 entsprechen unter anderem die Bildunterschriften nicht dem gesuchten Muster. Das System sucht nach Blöcken, die z.B. das Wort "Figure" gefolgt von einer Zahl und einem Doppelpunkt beinhalten. In diesem Artikel wurde aber statt dem Doppelpunkt ein einfacher Punkt verwendet. Außerdem wurde Seite 9 fälschlicherweise komplett als uninteressant markiert. Dies liegt vor allem daran, dass die Abschnitte 5 und 6 des Artikels auf dieser Seite von vielen Zeilenumbrüchen unterbrochen werden. Der Algorithmus `findSmallerBlocks()` hat diese Abschnitte aussortiert, da hier die durchschnittliche Zeilenbreite gering war. Dieses Problem trat auch bei Artikel 23 auf. Eine der Seiten wurde vom `findSmallerBlocks()`-Algorithmus ebenso komplett verworfen. Am deutlichsten unterhalb dem Durchschnitt liegt das Ergebnis für den Artikel 14. Dies liegt daran, dass dieser Artikel auf deutsch geschrieben ist. Sämtliche Algorithmen, die nach Schlüsselworten suchen, scheitern an diesem Artikel. Außerdem tritt hier auch das Problem auf, dass lange Textbereiche mit vielen Zeilenumbrüchen fälschlicherweise aussortiert werden. Dies erklärt diese große Abweichung.
Einige der Ergebnisse weichen deutlich von den durchschnittlichen Ergebnissen ab. Da aber mit 19 Artikeln fast $\frac{2}{3}$ aller untersuchten Artikel einen F_1-Wert von mehr als 0,98 aufweisen und sogar fast die Hälfte (13 Artikel) einen F_1-Wert von mehr als 0,99 aufweisen zeigt dies, dass das System sehr gute Ergebnisse liefert. Einige der Artikel haben zweispaltige Layouts, andere haben nur eine Spalte. Einige Artikel haben Kopfzeilen, manche Artikel haben in den Bildbeschreibungen keine Doppelpunkte, wie eigentlich vom Programm untersucht. Diese sehr guten Ergebnisse sind somit unabhängig vom Layout des Artikels.

[3] Die Präzision (engl. *precision*) errechnet sich als $precision = \frac{richtigpositiv}{richtigpositiv+falschpositiv}$. Sie gibt an, wie viele der interessant markierten Worte tatsächlich interessant sind.

[4] Die Sensitivität (engl. *recall*) gibt an, wie viele der interessanten Worte auch tatsächlich als interessant markiert wurden. Sie wird berechnet durch $recall = \frac{richtigpositiv}{richtigpositiv+falschnegativ}$

[5] Der F_1-Wert ist ein Maß aus Präzision und Sensitivität. Er wird berechnet als $F_1 = 2 \cdot \frac{precision \cdot recall}{precision+recall}$.

4.1.3. Publikationen aus *PLoS Biology Articles*

Das in Kap. 2.1 vorgestellte System wurde entwickelt, um Dokumente aus dem *PLoS Biology Journal* vom Computer strukturiert erfassen zu können. Dieses System konnte nicht zum Laufen gebracht werden, weshalb keine strukturierte Textextraktion bei Artikeln aus der Menge der wissenschaftlichen Artikel vom Lehrstuhl für Algorithmen und Datenstrukturen durchgeführt werden konnte. Aus diesem Grund wurden einige von diesem System analysierten wissenschaftlichen Artikel durch das in dieser Arbeit vorgestellte System analysiert. Die Ergebnisse, welche in Tab. 4.2 zu sehen sind, sind nicht direkt vergleichbar, da das in Kap. 2.1 vorgestellte System die Textblöcke nicht bloß in interessant und uninteressant klassifiziert. Das hier vorgestellte System erreicht eine Präzision von 0,963 und eine Sensitivität von 0,942, was einen F_1-Wert von 0,952 ergibt. Diese Werte sind niedriger, als die der in Kap. 4.1.2 analysierten Menge. Dafür gibt es mehrere Gründe:

- Das Layout der Artikel sah es in den Bänden 1-4 (Artikel 30-35) vor, den Abstract nur durch Fettschrift hervorzuheben und nicht durch das Schlüsselwort "Abstract".
- Auf der zweiten Seite aller Artikel ist ein Abschnitt *Author Summary*, der nicht zum Fließtext gehört, aber in den Artikeln 30-35 genauso breit dargestellt wurde wie der Fließtext. Die einzige Hervorhebung ist die Überschrift und ein blauer Hintergrund, der vom hier dargestellten System nicht zu erkennen ist.
- Sämtliche Bildunterschriften unter Grafiken und Tabellen fangen zwar mit den Schlüsselworten "Figure" oder "Table" an, es folgt aber auf die Zahl kein Doppelpunkt, sondern ein einfacher Punkt.
- In allen Artikeln ist ein Abschnitt *Materials and Methods* zu finden. Dieser Abschnitt ist unterschiedlich lang und beinhaltet nur Informationen zu verwendeten Materialien und Methoden.
- Ebenso uninteressant ist der in allen Artikeln vorkommende Abschnitt *Supporting Information*. Dieser Abschnitt beinhaltet Webadressen zu Bildern, Tabellen und Videos und dazugehörige Bildunterschriften.
- Der Abschnitt *Acknowledgments*, am Ende der Artikel, wurde ebenso vom System meistens fälschlicherweise als interessant erkannt.

Trotz dieser Reihe an Punkten, die bei der Entwicklung des Systems nicht explizit beachtet wurden, sind die erreichten Ergebnisse sehr gut.

Tabelle 4.1.: Die Ergebnisse der Klassifizierung der Artikel von der Webseite des Lehrstuhls für Algorithmen und Datenstrukturen durch das System. In den Spalten sind die fortlaufende Artikelnummer, die Anzahl der Worte, Anzahl der richtig positiv (rp) und richtig negativ (rn) klassifizierten Worte. Anschließend folgen falsch negativ (fn) und falsch positiv (fp) klassifizierte Worte. Die letzten beiden Spalten beinhalten die Präzision und Sensitivität.

Nr.	**Worte**	**rp**	**rn**	**fn**	**fp**	**Präzision**	**Sensitivität**
1	15.068	13.682	44	1.332	10	0,997	0,999
2	13.485	11.790	227	1.186	281	0,981	0,977
3	9.801	8.287	0	707	808	1,000	0,911
4	43.194	39.170	4	3.844	177	1,000	0,996
5	19.839	18.203	3	1.632	1	1,000	1,000
6	18.504	16.272	0	2.023	208	1,000	0,987
7	11.588	10.075	273	1.145	96	0,974	0,991
8	13.144	10.791	189	1.455	709	0,983	0,938
9	20.283	18.935	12	928	409	0,999	0,979
10	9.177	8.425	0	719	33	1,000	0,996
11	15.820	14.805	15	956	44	0,999	0,997
12	20.092	17.224	34	1.354	1.480	0,998	0,921
13	10.759	10.158	62	539	0	0,994	1,000
14	7.586	5.102	504	272	1.708	0,910	0,749
15	4.252	3.773	29	450	0	0,992	1,000
16	2.052	1.614	220	214	4	0,880	0,998
17	15.825	14.907	167	722	29	0,989	0,998
18	13.867	10.623	624	1.713	907	0,945	0,921
19	7.695	7.343	148	202	2	0,980	1,000
20	12.837	11.348	126	1.316	47	0,989	0,996
21	3.006	2.509	19	478	0	0,992	1,000
22	15.253	12.853	380	1.694	327	0,971	0,975
23	13.577	10.583	125	1.609	1.261	0,988	0,894
24	4.789	3.588	16	938	247	0,996	0,936
25	87.935	80.649	581	6.544	161	0,993	0,998
26	23.520	20.014	48	2.991	467	0,998	0,977
27	18.751	17.484	387	808	72	0,978	0,996
28	4.917	4.296	49	451	121	0,989	0,973
29	22.043	20.323	0	1.589	131	1,000	0,994
$\sum$	**478.659**	**424.826**	**4.286**	**39.811**	**9.740**	**0,990**	**0,978**

Tabelle 4.2.: Die Ergebnisse der Klassifizierung der Artikel aus *PLoS Biology Articles* durch das System. In den Spalten sind die fortlaufende Artikelnummer, die Anzahl der Worte, Anzahl der richtig positiv (rp) und richtig negativ (rn) klassifizierten Worte, gefolgt von falsch negativ (fn) und falsch positiv (fp) klassifizierten Worte. Die letzten beiden Spalten beinhalten die Präzision und die Sensitivität.

Nr.	Worte	rp	rn	fn	fp	Präzision	Sensitivität
30	10.007	4.956	1.365	2.159	1.526	0,784	0,765
31	27.771	20.668	380	5.508	1.215	0,982	0,944
32	20.124	15.194	349	2.577	2.004	0,978	0,883
33	21.231	14.809	236	4.018	2.168	0,984	0,872
34	19.668	15.351	178	3.761	378	0,989	0,976
35	27.576	21.018	505	5.725	328	0,977	0,985
36	31.377	25.141	1.061	5.170	4	0,960	1,000
37	17.182	12.323	940	3.507	413	0,929	0,968
$\sum$	**174.936**	**129.460**	**5.014**	**32.425**	**8.036**	**0,963**	**0,942**

4.2. Analyse der Laufzeit

Da bei einer großen Menge an verarbeiteten wissenschaftlichen Artikeln auch die Laufzeit eine Rolle spielt, wurde während der Analyse der Qualität auch die Laufzeit analysiert. Dazu wurde während verschiedener Punkte des Systems die aktuelle Systemzeit gemessen und die Differenzen nach der Abarbeitung des Programms in eine Datei geschrieben. Anschließend wurden die Zeiten zusammengefasst.

4.2.1. Ergebnisse

Die Analysen der PDF-Dokumente wurden auf einem gewöhnlichen Desktop-PC durchgeführt. Die Hardware bestand aus einer 3,2GHz AMD Quadcore CPU und 8GB Ram. Für die 37 Dokumente mit zusammen 471 Seiten brauchte das System etwa 108 Sekunden. Dabei war die Initialisierung, bestehend aus der Extraktion der Buchstaben und dem Zusammenfügen der Worte, am zeitaufwendigsten. Etwa $\frac{3}{4}$ der Zeit benötigte das System dafür. Die Extraktion der Zeichen benötigte etwa 68 Sekunden (63,6%). Das Zusammenfügen der Wörter benötigte etwas mehr als 1 Sekunde (1,2%). Die drei Aufrufe der Funktion `makeGrid()` brauchten insgesamt etwa 21 Sekunden (19,8%). Die restlichen Analysewerkzeuge benötigten zusammen nicht einmal 5 Sekunden (4,6%). Das Schreiben der Ergebnisse in die XML-Datei war innerhalb von etwas mehr als 11 Sekunden (10,45%) abgeschlossen. Etwa 75% der Zeit benötigt das System für die Initialisierung und Ausgabe. Für eine deutliche Verringerung der Laufzeit müsste somit eine andere PDF-Bibliothek verwendet, oder die benötigten Funktionen in *PDFBox* umgeschrieben werden. Für die Analyse einer einzelnen Seite wurden im Schnitt etwa 230ms benötigt. Somit ist eine Analyse ad

hoc beim Betrachten eines Dokuments möglich. Ebenso möglich ist aber auch eine Analyse einer größeren Menge von wissenschaftlichen Artikeln, wie etwa die Referenzen einer Doktorarbeit, in kurzer Zeit. Eine Tabelle mit den zusammengefassten Laufzeiten befindet sich im Anhang in Kap. A.2.

5. Diskussion

5.1. Qualität der Extraktion

Obwohl insgesamt mehr als 95,8% der Worte richtig klassifiziert wurden, haben sich bei der Analyse in Kap. 4.1.2 und Kap. 4.1.3 einige Schwächen des vorgestellten Systems gezeigt. Viele der verwendeten Werkzeuge können noch durch Tests verfeinert werden, die im Endeffekt bei einigen wenigen Artikeln zu einem besseren Ergebnis führen. So etwa beim *findSmallerBlocks*-Algorithmus, der bei einigen Artikeln Blöcke aussortiert hat, die interessant waren und deshalb nicht aussortiert werden sollten. Allerdings hat der Algorithmus bei anderen Artikeln erfolgreich Blöcke aussortiert, die uninteressant waren. Im Folgenden werden die Punkte genannt, die Wahrscheinlich die größte Verbesserung der Ergebnisse bewirken.

5.1.1. Sprache

Der größte Schwachpunkt ist die nicht vorhandene Unterstützung von Sprachen. Da viele der verwendeten Werkzeuge die Blöcke nach Schlüsselworten untersuchen, scheitern sie, wenn z. B. statt dem Schlüsselwort "References" das Schlüsselwort "Literaturverzeichnis" verwendet wird. Eine Möglichkeit, für unterschiedliche Sprachen unterschiedliche Schlüsselworte zu hinterlegen, wäre z. B. eine XML-Datei anzulegen, die zu jedem möglichen Schlüsselwort die Übersetzungen in verschiedenen Sprachen beinhaltet. Dabei wäre es auch möglich, verschiedene Abkürzungen zu den Schlüsselworten abzuspeichern. Einige der wissenschaftlichen Artikel verwendeten z. B. statt "Table" die Abkürzung "Tab". Eine weitere Möglichkeit, die Schlüsselworte zu übersetzen wären auch Onlineübersetzer, die zur Laufzeit nach den Worten suchen. Hier wäre allerdings erstens eine Internetanbindung nötig und zweitens könnte die Laufzeit sich deutlich erhöhen. Zu Beginn der Arbeit wurde eine Analyse der Sprache mit [Shu10] durchgeführt. Die Qualität war gut, die Geschwindigkeit allerdings zu niedrig. Deshalb und wegen der sehr guten Ergebnisse bei englischen Artikeln, die die Mehrzahl der wissenschaftlichen Artikel ausmachen, wurde dies nicht weiter verfolgt.

5.1.2. Kopf- und Fußzeilen

Ein weiterer Punkt ist die Erkennung von Kopf- und Fußzeilen. In einigen Artikeln wurden diese erfolgreich erkannt und entfernt. Um diese jedoch bei allen wis-

senschaftlichen Artikeln erfolgreich zu entfernen, müsste ein Werkzeug genau nach diesen suchen. Dabei könnte, ähnlich wie in Kap. 2.3 beschrieben, am oberen und unteren Rand des Artikels nach sich wiederholenden Texten gesucht werden, da Kopf- und Fußzeilen meistens Wasserzeichen, Kapitelüberschriften, Autoren oder den Titel des Artikels beinhalten. Oft ist hier auch eine Angabe zur veröffentlichenden Zeitschrift, etc. vorhanden. Man könnte auch explizit nach dem vorher schon extrahierten Titel des wissenschaftlichen Artikels oder dessen Autoren suchen. Eine weitere Möglichkeit wäre es, am oberen Rand nach den gefundenen Teilüberschriften zu suchen.

5.1.3. Gescannte Artikel

Gar nicht betrachtet wurden wissenschaftliche Artikel, die nicht als PDF-Dokument erstellt wurden, sondern aus Büchern gescannt wurden. Die Seiten bestehen in diesen Artikeln aus Bildern, die sowohl Text als auch Abbildungen beinhalten. Diese Dokumente müssen zuerst mit einem OCR-Programm bearbeitet werden, um vom hier vorgestellten System analysiert werden zu können. Die Implementierung einer eigenen Texterkennung hätte den Rahmen dieser Arbeit gesprengt.

5.1.4. Erkennung des Abstract

Unter anderem wurde bei den in Kap. 4.1.3 untersuchten wissenschaftlichen Artikeln der Abstract nicht korrekt erkannt, da das gesuchte Schlüsselwort fehlte. Eine Möglichkeit dies zu umgehen, wäre es den Abstract wie gewohnt zu suchen. Kann der Abstract nicht gefunden werden, könnte der erste Block auf der ersten Seite eines wissenschaftlichen Artikels generell als Abstract markiert werden. Dies könnte aber bei einigen anderen Artikeln dann zu Fehlern führen, wenn der *makeGrid*-Algorithmus nicht optimal die Seite in Blöcke aufteilt.

Danksagung

An dieser Stelle möchte ich mich bei Frau Professor Dr. Hannah Bast für die Bereitstellung des Themas und die Übernahme des Erstgutachtens bedanken. Außerdem möchte ich mich bei Claudius Korzen für die Betreuung während der Bearbeitung und für die Tipps und Denkanstöße bedanken.
Bei Jan Jürgensen bedanke ich mich für die Korrekturlesesitzung. Die Zwischenfragen dabei haben deutlich zur Verständlichkeit einiger Passagen beigetragen.
Meike danke ich für das Korrekturlesen und das Verständnis für die etlichen Stunden, die ich "lieber" mit dieser Arbeit verbracht habe.
Zu guter Letzt möchte ich meinen Eltern für die Freiheiten beim Verfolgen meines Studiums und die immerwährende Unterstützung danken.

A. Anhang

A.1. Zuordnung der wissenschaftlichen Artikel zur fortlaufenden Nummerierung

Die folgenden Tabellen beinhalten die in Kap. 4.1.2 und Kap. 4.1.3 verwendeten fortlaufenden Nummern der analysierten wissenschaftlichen Artikel. Die Titel und Autoren können mit diesen Tabellen zugeordnet werden.

Nr	**Titel**	**Autoren**
1	Delay-Robustness of Transfer Patterns in Public Transportation Route Planning	Hannah Bast, Jonas Sternisko, and Sabine Storandt
2	Result Diversity for Multi-Modal Route Planning	Hannah Bast, Mirko Brodesser, and Sabine Storandt
3	Enabling E-Mobility: Facility Location for Battery Loading Stations	Sabine Storandt, Stefan Funke
4	Fast Construction of the HYB Index	Hannah Bast and Marjan Celikik
5	Flow-Based Guidebook Routing Author	Hannah Bast, Sabine Storandt
6	Polynomial-time Construction of Contraction Hierarchies for Multi-criteria Objectives	Stefan Funke, Sabine Storandt
7	An Index for Efficient Semantic Full-Text Search	Hannah Bast, Björn Buchhold
8	Broccoli: Semantic Full-Text Search at your Fingertips	Hannah Bast, Florian Bäurle, Björn Buchhold, Elmar Haussmann
9	Optimized Java Binary and Virtual Machine for Tiny Motes	Faisal Aslan, Luminous Fennell, Christian Schindelhauer, Peter Thiemann, Gidon Ernst, Elmar Haussmann, Stefan Rührup and Zastash A. Uzmi
10	More Informative Open Information Extraction via Simple Inference	Hannah Bast and Elmar Haussmann
11	Car or Public Transport-Two Worlds	Hannah Bast

12	Fast Routing in Very Large Public Transportation Networks using Transfer Patterns	Hannah Bast, Erik Carlsson, Arno Eigenwillig, Robert Geisberger, Chris Harrelson, Veselin Raychev and Fabien Viger
13	Open Information Extraction via Contextual Sentence Decomposition	Hannah Bast, Elmar Haussmann
14	Semantische Suche	Hannah Bast
15	A Case for Semantic Full-Text Search (position paper)	Hannah Bast, Florian Bäurle, Björn Buchhold, Elmar Haussmann
16	Efficient Two-Sided Error-Tolerant Search	Hannah Bast, Marjan Celikik
17	Contraction Hierarchies on Grid Graphs	Sabine Storandt
18	Rethinking Java Call Stack Design for Tiny Embedded Devices	Faisal Aslam, Ghufran Baig, Mubashir Adnan Qureshi, Zartash Afzal Uzmi, Luminous Fennell, Peter Thiemann, Christian Schindelhauer, Elmar Haussmann
19	Distributed Online Route Computation - Higher Throughput, more Privacy	Niklas Schnelle, Stefan Funke, Sabine Storandt
20	Fast Error-Tolerant Search on Very Large Texts	Marjan Celikik, Holger Bast
21	Semantic Full-text Search with Broccoli	Hannah Bast, Florian Bäurle, Björn Buchhold, Elmar Haußmann
22	Frequency-Based Search for Public Transit	Hannah Bast, Sabine Storandt
23	Real-Time Movement Visualization of Public Transit Data	Hannah Bast, Patrick Brosi, Sabine Storandt
24	TRAVIC: A Visualization Client for Public Transit Data	Hannah Bast, Patrick Brosi, Sabine Storandt
25	Efficient Fuzzy Search in Large Text Collections	Hannah Bast, Marjan Celikik
26	Efficient Index-Based Snippet Generation	Hannah Bast, Marjan Celikik
27	ForestMaps: A Computational Model and Visualization for Forest Utilization	Hannah Bast, Jonas Sternisko, and Sabine Storandt
28	Easy Access to the Freebase Dataset	Hannah Bast, Florian Bäurle, Björn Buchhold, Elmar Haußmann
29	The Icecite Research Paper Management System	Hannah Bast and Claudius Korzen

30	DNA Damage-Induced Bcl-x_L Deamidation Is Mediated by NHE-1 Antiport Regulated Intracellular pH	Rui Zhao, David Oxley, Trevor S. Smith, George A. Follows, Anthony R. Green, Denis R. Alexander
31	Impaired Genome Maintenance Suppresses the Growth Hormone-Insulin-Like Growth Factor 1 Axis in Mice with Cockayne Syndrome	Ingrid van der Pluijm, George A. Garinis, Renata M. C. Brandt, Theo G. M. F. Gorgels, Susan W. Wijnhoven, Karin E. M. Diderich, Jan de Wit, James R. Mitchell, Conny van Oostrom, Rudolf Beems, Laura J. Niedernhofer, Susana Velasco, Errol C. Friedberg, Kiyoji Tanaka, Harry van Steeg, Jan H. J. Hoeijmakers, Gijsbertus T. J. van der Horst
32	Distinct Cerebral Pathways for Object Identity and Number in Human Infants	Veronique Izard, Ghislaine Dehaene-Lambertz, Stanislas Dehaene
33	Amyloid as a Depot for the Formulation of Long-Acting Drugs	Samir K. Maji, David Schubert, Catherine Rivier, Soon Lee, Jean E. Rivier, Roland Riek
34	An Endoribonuclease Functionally Linked to Perinuclear mRNP Quality Control Associates with the Nuclear Pore Complexes	Michal Skruzy, Claudia Schneider, Attila Racz, Julan Weng, David Tollervey, Ed Hurt
35	Collective Cell Migration Drives Morphogenesis of the Kidney Nephron	Aleksandr Vasilyev, Yan Liu, Sudha Mudumana, Steve Mangos, Pui-Ying Lam, Arindam Majumdar, Jinhua Zhao, Kar-Lai Poon, Igor Kondrychyn, Vladimir Korzh, Iain A. Drummond
36	Poised Transcription Factories Prime Silent uPA Gene Prior to Activation	Carmelo Ferrai, Sheila Q. Xie, Paolo Luraghi, Davide Munari, Francisco Ramirez, Miguel R. Branco, Ana Pombo, Massimo P. Crippa
37	Ultradeep Sequencing of a Human Ultraconserved Region Reveals Somatic and Constitutional Genomic Instability	Anna De Grassi, Cinzia Segala, Fabio Iannelli, Sara Volorio, Lucio Bertario, Paolo Radice, Loris Bernard, Francesca D. Ciccarelli

A.2. Laufzeiten des Systems

In den folgenden Tabellen sind die zusammengefassten Laufzeiten der Werkzeuge des Systems aufgelistet. Alle Zeiten sind in Millisekunden. Die Spalten beinhalten die Gesamtlaufzeit (*Total*), die Dauer der Extraktion der Zeichen mit ihren Eigenschaften (*Extraktion*), das Zusammenfügen der Worte (*Worte*), die drei Aufrufe von `makeGrid()` (*Grid*), die Laufzeit der restlichen Analysewerkzeuge (*Analyse*), die Zeit für Ausgabe (*Output*) und interne Schritte zur Organisation der Listen und *HashMaps* (*Intern*).

Nr.	Total	Extraktion	Worte	Grid	Analyse	Output	Intern
1	2.280	1.600 (70,18%)	28 (1,23%)	449 (19,69%)	136 (5,96%)	54 (2,37%)	11 (0,48%)
2	2.191	1.506 (68,74%)	31 (1,41%)	421 (19,21%)	99 (4,52%)	128 (5,84%)	5 (0,23%)
3	2.027	1.383 (68,23%)	28 (1,38%)	437 (21,56%)	103 (5,08%)	68 (3,35%)	6 (0,30%)
4	5.742	3.073 (53,52%)	52 (0,91%)	1.066 (18,56%)	298 (5,19%)	1.222 (21,28%)	27 (0,47%)
5	2.370	1.548 (65,32%)	27 (1,14%)	498 (21,01%)	129 (5,44%)	160 (6,75%)	7 (0,30%)
6	2.640	1.637 (62,01%)	122 (4,62%)	603 (22,84%)	159 (6,02%)	109 (4,13%)	9 (0,34%)
7	3.124	2.065 (66,10%)	42 (1,34%)	668 (21,38%)	204 (6,53%)	134 (4,29%)	9 (0,29%)
8	2.772	1.873 (67,57%)	31 (1,12%)	600 (21,65%)	165 (5,95%)	86 (3,10%)	8 (0,29%)
9	2.561	1.903 (74,31%)	26 (1,02%)	436 (17,02%)	107 (4,18%)	81 (3,16%)	7 (0,27%)
10	1.470	1.164 (79,18%)	20 (1,36%)	186 (12,65%)	31 (2,11%)	65 (4,42%)	3 (0,20%)
11	2.403	1.663 (69,21%)	27 (1,12%)	449 (18,68%)	114 (4,74%)	141 (5,87%)	8 (0,33%)
12	2.148	1.520 (70,76%)	25 (1,16%)	426 (19,83%)	94 (4,38%)	75 (3,49%)	7 (0,33%)
13	2.039	1.391 (68,22%)	25 (1,23%)	365 (17,90%)	104 (5,10%)	147 (7,21%)	6 (0,29%)
14	2.385	1.844 (77,32%)	27 (1,13%)	359 (15,05%)	72 (3,02%)	78 (3,27%)	4 (0,17%)
15	1.525	1.156 (75,80%)	21 (1,38%)	237 (15,54%)	59 (3,87%)	49 (3,21%)	3 (0,20%)
16	1.424	1.086 (76,26%)	16 (1,12%)	167 (11,73%)	46 (3,23%)	106 (7,44%)	3 (0,21%)
17	1.837	1.309 (71,26%)	25 (1,36%)	330 (17,96%)	74 (4,03%)	92 (5,01%)	6 (0,33%)
18	2.834	1.932 (68,17%)	35 (1,24%)	657 (23,18%)	135 (4,76%)	64 (2,26%)	9 (0,32%)
19	1.628	1.194 (73,34%)	22 (1,35%)	249 (15,29%)	69 (4,24%)	89 (5,47%)	4 (0,25%)
20	2.562	1.600 (62,45%)	33 (1,29%)	654 (25,53%)	153 (5,97%)	112 (4,37%)	9 (0,35%)
21	1.465	1.158 (79,04%)	20 (1,37%)	136 (9,28%)	24 (1,64%)	123 (8,40%)	1 (0,07%)
22	2.873	1.999 (69,58%)	30 (1,04%)	564 (19,63%)	152 (5,29%)	115 (4,00%)	11 (0,38%)
23	2.786	1.933 (69,38%)	34 (1,22%)	580 (20,82%)	141 (5,06%)	85 (3,05%)	11 (0,39%)
24	1.616	1.239 (76,67%)	22 (1,36%)	244 (15,10%)	45 (2,78%)	62 (3,84%)	3 (0,19%)
25	11.791	4.290 (36,38%)	69 (0,59%)	1.950 (16,54%)	644 (5,46%)	4.809 (40,79%)	26 (0,22%)
26	4.581	2.788 (60,86%)	38 (0,83%)	836 (18,25%)	266 (5,81%)	634 (13,84%)	12 (0,26%)
27	2.150	1.489 (69,26%)	30 (1,40%)	435 (20,23%)	105 (4,88%)	82 (3,81%)	8 (0,37%)
28	1.842	1.446 (78,50%)	23 (1,25%)	271 (14,71%)	63 (3,42%)	35 (1,90%)	3 (0,16%)
29	2.260	1.733 (76,68%)	29 (1,28%)	349 (15,44%)	75 (3,32%)	67 (2,96%)	6 (0,27%)
30	4.159	2.486 (59,77%)	37 (0,89%)	971 (23,35%)	200 (4,81%)	451 (10,84%)	11 (0,26%)
31	4.046	2.566 (63,42%)	45 (1,11%)	970 (23,97%)	132 (3,26%)	320 (7,91%)	11 (0,27%)
32	3.375	2.073 (61,42%)	37 (1,10%)	811 (24,03%)	122 (3,61%)	324 (9,60%)	7 (0,21%)
33	3.444	2.173 (63,10%)	41 (1,19%)	886 (25,73%)	104 (3,02%)	231 (6,71%)	7 (0,20%)
34	2.790	1.859 (66,63%)	49 (1,76%)	645 (23,12%)	92 (3,30%)	138 (4,95%)	6 (0,22%)
35	3.497	2.202 (62,97%)	69 (1,97%)	849 (24,28%)	118 (3,37%)	248 (7,09%)	9 (0,26%)
36	3.767	2.391 (63,47%)	45 (1,19%)	866 (22,99%)	157 (4,17%)	298 (7,91%)	9 (0,24%)
37	3.289	2.216 (67,38%)	44 (1,34%)	707 (21,50%)	132 (4,01%)	181 (5,50%)	7 (0,21%)
$\sum$	**107.693**	**68.488 (63,60%)**	**1.325 (1,23%)**	**21.327 (19,80%)**	**4.923 (4,57%)**	**11.263 (10,46%)**	**299 (0,28%)**

A.3. Verwendung des Systems

Die Verwendung des Systems ist sehr einfach. Dazu muss nur die Datei *PDFTextExtractor.jar* gestartet werden. Als erstes Argument muss der Pfad der zu extrahierenden PDF-Datei angegeben werden. Automatisch wird eine XML-Datei erstellt, die den Dateinamen des PDF erweitert um die Endung *.xml* hat. Diese XML-Datei befindet sich in dem Verzeichnis, aus welchem der *PDFTextExtractor* gestartet wurde.

Wird das Argument *time* mit angegeben, wird in der Datei *time.txt* ein Eintrag angehängt, der die Laufzeit des Programms misst. Einträge in dieser Datei beinhalten zuerst den Dateinamen, anschließend die Gesamtlaufzeit (*TOTAL*) in Millisekunden. Es folgt die Initialisierung (*init*), welche anschließend noch einmal aufgeteilt ist in Initialisierung der Buchstaben (*init_prop*) und Zusammensetzen der Worte (*init_words*). Der nächste Eintrag steht für die drei Aufrufe von `makeGrid()` (*gridx3*). Es folgen die Messungen für interne Operationen (*intern*), Ausgabeoperationen (*output*) und der Laufzeit der Analysewerkzeuge (*analyse*). Alle Zeiten werden in Millisekunden jeweils ohne Trennzeichen ausgegeben.
Mit dem zusätzlichen Argument *paint* werden für jede Seite des Dokuments zwei gleiche Grafikdateien erstellt. Eine Grafikdatei ist um das Wort *calculated* erweitert. Diese Grafikdatei soll nicht bearbeitet werden. Die zweite Grafikdatei wird um das Wort *painted* erweitert. Diese Grafikdatei muss in einem Grafikprogramm bearbeitet werden. Interessante Worte sind dabei schwarz, uninteressante Worte in rot zu markieren. Die Worte in diesen Grafikdateien sind jeweils als 3x3 Pixel große Rechtecke dargestellt. Wird das Programm mit dem Argument *measurement* aufgerufen liest es entsprechend die Grafikdateien und vergleicht die Inhalte. Die Auswertung wird an die Textdatei *measurements.txt* angehängt. Einträge in dieser Datei beinhalten zuerst den Dateinamen. Anschließend kommen Einträge für die Gesamtzahl an Worten (*TOTAL*), richtig positiv (*RP*), falsch positiv (*FP*), richtig negativ (*RN*) und falsch negativ (*FN*) klassifizierte Worte.

A.4. Inhalte auf der beiligenden DVD

Im Ordner *bin* befindet sich das lauffähige Programm *PDFTextExtractor.jar*. Der Ordner *source* beinhaltet den Java-Quelltext des Systems. Im Ordner *results* befinden sich die Unterordner *ad*, *plos* und *time*. Der Ordner *ad* beinhaltet sowohl alle PDF-Dateien als auch alle Grafikdateien, welche für die Auswertung in Kap. 4.1.2 benutzt wurden. Außerdem sind darin die Datei *measurements.txt* und die XML-Ausgabedateien zu finden. Im Ordner *plos* sind die entsprechenden Dateien für die Auswertungen aus Kap. 4.1.3 zu finden. Der Unterordner *time* beinhaltet die Ergebnisse aus Kap. 4.2, bzw. Kap. A.2 in der Datei *time.txt*.

Literaturverzeichnis

[BBBH12] Bast, Hannah ; Bäurle, Florian ; Buchhold, Björn ; Haussmann, Elmar: Broccoli: Semantic Full-Text Search at your Fingertips. In: *CoRR* abs/1207.2615 (2012)

[BK13] Bast, Hannah ; Korzen, Claudius: The Icecite Research Paper Management System. In: Lin, Xuemin (Hrsg.) ; Manolopoulos, Yannis (Hrsg.) ; Srivastava, Divesh (Hrsg.) ; Huang, Guangyan (Hrsg.): *Web Information Systems Engineering - WISE 2013* Bd. 8181. Springer Berlin Heidelberg, 2013. – ISBN 978–3–642–41153–3, S. 396–409

[Bur98] Burges, Christopher J.: A tutorial on support vector machines for pattern recognition. In: *Data mining and knowledge discovery* 2 (1998), Nr. 2, S. 121–167

[CC13] Chen, Jianguo ; Chen, Hao: A Structured Information Extraction Algorithm for Scientific Papers based on Feature Rules Learning. In: *Journal of Software* 8 (2013), Nr. 1

[CL11] Chang, Chih-Chung ; Lin, Chih-Jen: LIBSVM: A library for support vector machines. In: *ACM Transactions on Intelligent Systems and Technology* 2 (2011), S. 27:1–27:27. – Software available at `http://www.csie.ntu.edu.tw/~cjlin/libsvm`

[DM06] Déjean, Hervé ; Meunier, Jean-Luc: A System for Converting PDF Documents into Structured XML Format. In: *Proceedings of the 7th International Conference on Document Analysis Systems*. Berlin, Heidelberg : Springer-Verlag, 2006 (DAS'06). – ISBN 3–540–32140–3, 978–3–540–32140–8, S. 129–140

[FD95] Feldman, Ronen ; Dagan, Ido: Knowledge Discovery in Textual Databases (KDT). In: *In Proceedings of the First International Conference on Knowledge Discovery and Data Mining (KDD-95*, AAAI Press, 1995, S. 112–117

[GMM03] Guha, R. ; McCool, Rob ; Miller, Eric: Semantic Search. In: *Proceedings of the 12th International Conference on World Wide Web*. New York, NY, USA : ACM, 2003 (WWW '03). – ISBN 1–58113–680–3, S. 700–709

[Hen01] Henke, Harold: *Electronic books and ePublishing : a practical guide for authors*. London, Berlin, Paris : Springer, 2001. – ISBN 1–85233–435–5

[HLT05] Hollingsworth, Bill ; Lewin, Ian ; Tidhar, Dan: Retrieving hierarchical text structure from typeset scientific articles–a prerequisite for e-science text mining. In: *Proc. of the 4th UK E-Science All Hands Meeting*, 2005, S. 267–273

[ISO08] ISO: Document management—Portable document format—Part 1: PDF 1.7 / International Organization for Standardization. Geneva, Switzerland, 2008 (32000-1:2008). – ISO

[KLN10] Kan, Min-Yen ; Luong, Minh-Thang ; Nguyen, Thuy D.: Logical Structure Recovery in Scholarly Articles with Rich Document Features. In: *Int. J. Digit. Library Syst.* 1 (2010), Oktober, Nr. 4, S. 1–23. – ISSN 1947–9077

[Meu05] Meunier, Jean-Luc: Optimized XY-Cut for Determining a Page Reading Order. In: *ICDAR* Bd. 5, 2005, S. 347–351

[Oraa] Oracle Corp. *Class Collections*. Internet: http://docs.oracle.com/javase/7/docs/api/java/util/Collections.html (letzter Zugriff 11.04.2015)

[Orab] Oracle Corp. *Class Collections*. Internet: http://docs.oracle.com/javase/7/docs/api/java/util/TreeSet.html (letzter Zugriff 11.04.2015)

[Orac] Oracle Corp. *Class HashMap<K,V>*. Internet: http://docs.oracle.com/javase/7/docs/api/java/util/HashMap.html (letzter Zugriff 06.04.2015)

[PDF] PDFlib. *PDFlib TET 4.4 - Text Extraction Toolkit*. Internet: http://www.pdflib.com/de/produkte/tet/ (letzter Zugriff 02.05.2015)

[RPHB12] Ramakrishnan, Cartic ; Patnia, Abhishek ; Hovy, Eduard ; Burns, Gully: Layout-Aware Text Extraction from Full-text PDF of Scientific Articles. In: *Source Code for Biology and Medicine* 7 (2012), Nr. 1, S. 7. – ISSN 1751–0473

[Shu10] Shuyo, Nakatani. *Language Detection Library for Java*. 2010

[Sum98] Summers, Kristen: *Automatic discovery of logical document structure*, Cornell University, Diss., 1998

[XER] XEROX Corp. *Rossinante Web Service*. Internet: https://open.xerox.com/Services/RossinanteWS (letzter Zugriff 02.05.2015)