Diplomarbeit

Benutzergeführtes Lernen von Dokument-Struktur-auszeichnungen aus Formatierungsmerkmalen

Christian Hüppe

Diplomarbeit
am Fachbereich Informatik
der Universität Dortmund

Februar 2003

Betreuer:

Prof. Dr. Katharina Morik
Dipl. Inform. Stefan Haustein

Ein Problem wird nicht im Computer gelöst, sondern in irgendeinem Kopf. Die ganze Apparatur dient nur dazu, diesen Kopf so weit zu drehen, dass er die Dinge richtig und vollständig sieht.

Charles Kettering, amerikan. Ing., 1876-1958

Danksagung

An dieser Stelle möchte ich mich ganz herzlich bei meinen beiden Betreuern, Prof. Dr. Katharina Morik und Dipl. Inform. Stefan Haustein, für ihrer zahlreichen Hilfestellungen bedanken. Durch ihre konstruktiven Ratschläge konnte diese Arbeit in eine erfolgreiche Richtung gelenkt werden.

Auch allen anderen Mitarbeitern und Hiwis des LS8 sei gedankt. Eure kleinen und großen Hilfen haben mir die Arbeit doch so manches mal erleichtert.

Abschließend möchte ich meinen Eltern danken, die mir durch ihre Unterstützung dieses interessante Studium ermöglichten.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1. Einleitung

Das erste Kapitel beschreibt die Aufgabenstellung, die dieser Diplomarbeit zu Grunde liegt. Anschließend werden verschiedene Lösungen der gestellten Aufgabe erörtert. Definitionen, der hierbei eingeführten Begriffe, folgen in den anschließenden Kapiteln.

1.1. Aufgabenbeschreibung

Moderne Textverarbeitungssysteme besitzen die Möglichkeit, statt physikalischer Textattribute wie Schriftgröße oder Schriftschnitt eine semantische Annotation vorzunehmen, also beispielsweise eine Überschrift für das System als solche erkennbar zu markieren. Diese semantische Textauszeichnung bietet eine Reihe von Vorteilen:

- **Automatische Erstellung von Inhalts- und Stichwortverzeichnissen:**
  Wurden Überschriften im Dokument als solche ausgezeichnet, können diese verwendet werden, um ein Inhaltsverzeichnis automatisch zu erstellen. Stichwortverzeichnisse werden auf dieselbe Art generiert. Voraussetzung ist auch hier, dass alle Stichwörter im Dokument gekennzeichnet wurden.
- **Einheitliche Formatierungsänderung:**
  Alle gleichen Strukturen werden einheitlich formatiert. Die Formatierung für das gesamte Dokument kann so nachträglich ohne großen Aufwand einheitlich geändert werden. Soll beispielsweise die Schriftgröße der Überschriften von 20 Punkte auf 24 Punkte erhöht werden, muss die Änderung nur an einer Stelle durchgeführt werden. Die Modifikation wirkt sich jedoch auf alle Überschriften gleich aus.
- **Sinnvolle Hypertext-Aufbereitung:**
  Eine sinnvolle Aufbereitung als Hypertext ist maschinell ohne semantische Auszeichnung nicht möglich. Verwendet man die Exportfunktionen moderner Textverarbeitungsprogramme, um ein Dokument in ein Hypertext-Dokument zu konvertieren, entstehen meist Dokumente, die nur optisch dem ursprünglichen Dokument ähneln. Beispielsweise werden Überschriften oft nicht als solche markiert, wie es in Hypertexten möglich ist, es wird ihnen lediglich eine bestimmte Schriftart und -größe zugeordnet.

Bei älteren Texten war solch eine semantische Textauszeichnung - abhängig vom Programm - oft nicht möglich, oder bei neueren Texten war dem Autor eine semantische

Auszeichnung zu kompliziert. In vielen Fällen wird aber eine solche Annotation nachträglich gewünscht, z.B. wenn diese Texte überarbeitet, weiterverwendet oder ins WWW gestellt werden sollen. Sicher ist es möglich, Texte manuell nachträglich semantisch auszuzeichnen. Viele dieser Texte besitzen jedoch eine starke implizite Strukturierung, bei der sich aus den verwendeten Schriftgrößen und weiteren Textattributen eine explizite semantische Strukturierung erstellen lässt. Diese Möglichkeit wird im Rahmen dieser Diplomarbeit untersucht.

Sollen Dokumentstrukturen ausgezeichnet werden, gibt es eine Reihe von Möglichkeiten dies zu tun:

Der sicherlich einfachste und zu gleich arbeitsaufwendigste Ansatz ist eine komplette manuelle Auszeichnung des Dokumentes. Ein Anwender konvertiert ein Dokument beispielsweise in ein ASCII-Format und bearbeitet das Dokument mit einem normalen Texteditor. Jede gewünschte Annotation muss dabei vom Anwender manuell vorgenommen werden. Unabhängig ob es sich dabei um Auszeichnungen von einzelnen Wörtern (*wortbezogene Auszeichnungen*) oder um Auszeichnungen von beliebig langem und beliebig geschachteltem Text (*geschachtelte Auszeichnungen*) handelt. Diese Vorgehensweise kann gerade bei langen Dokumenten sehr zeitaufwendig und unkomfortabel sein. Als Vorteil kann allerdings die hohe Korrektheit der Klassifikation genannt werden. Dies gilt vorallem dann, wenn der Autor selbst die Auszeichnungen vornimmt, da er sein Dokument meist am besten kennt.

Eine Lösung, die weniger Arbeitsaufwand für den Anwender bedeuten würde, wäre ein System, das Dokumentstrukturen automatisch Auszeichnungen zuordnet. Die unterschiedlichen Strukturen, wie Überschriften oder Aufzählungen, könnten mit Hilfe von zuvor definierten Regeln - beispielsweise mit Prolog - beschrieben werden. Ebenfalls ist es möglich Grammatiken oder reguläre Ausdrücke zu formulieren, mit denen einzelne Wörter oder beliebig geschachtelte Strukturen ausgezeichnet werden können. Der Vorteil bei diesem Ansatz wird gerade bei langen Dokumenten deutlich. Sind einmal die Regeln, Grammatiken und reguläre Ausdrücke erstellt, lassen sich auch lange Texte ohne zusätzliche Arbeit schnell verarbeiten. Es ergeben sich allerdings auch einige Nachteile:

- Für die Formulierung der benötigten Regeln wird zum einen Fachkenntnis benötigt, zum anderen sind Kenntnisse über die Eigenschaften, welche die verschiedenen Strukturen charakterisieren, bei der Erstellung der Regeln und Grammatiken notwendig.

- Eine weitere Schwäche dieses Ansatzes ergibt sich aus dem Problem, dass selbst Texte vom selbem Dokumenttyp[1] oft auf verschiedene Arten formatiert sind. Häufig verwenden ungeübte Autoren sogar verschiedene Formatierungen für gleiche Dokumentstrukturen. Dies führt zu dem Problem, dass die Regeln nicht auf allen Dokumenten angewandt werden können und darüberhinaus eventuell falsche Auszeichnungen vorgenommen werden.

  Das nachfolgende Beispiele demonstriert das Problem:

[1] z. B. Diplomarbeiten, Skripten o. Ä.

Beispiel 1:
Autor A schreibt ein Skript für eine Vorlesung. Bei der Erstellung des Dokumentes überlegt er sich, wie er Definitionen, Formeln und Beispiele am besten formatieren könnte. Schließlich entscheidet er sich für Beispiele die Schriftart `Typewriter` und für Definitionen die Schriftart *Italic* zu verwenden. Um Beispiele zusätzlich noch besser abzusetzen, werden sie gegenüber dem normalem Text eingerückt. Abbildung 1.1 zeigt einen Ausschnitt aus dem erstellten Skript*.

⋮

Wenn Parameter keine Objekte als Wert haben, sondern von einfachem Datentyp sind, wird der Wert direkt übergeben.

Definition 3.7: Wertübergabe *Sei in der Methodendeklaration ein Parameter v angegeben, dessen Typ ein einfacher Datentyp ist, sei im Methodenaufruf an entsprechender Stelle der Parameterliste eine Variable w angegeben, so wird der Wert von w kopiert und als Wert von v innerhalb der Methode eingetragen. Diese Parameterübergabe heißt Wertübergabe (englisch: call by value).*

Da bei der Wertübergabe kein Bezug zwischen den Variablen v und w hergestellt wird, sondern nur der Wert von w als Wert von v eingetragen wird, gibt es keine Referenz von v auf w. Folglich kann w nicht durch v verändert werden.

⋮

    Beispiel 3.3: Referenzzuweisung Nehmen wir an, eine Variable hat den Namen v, als Adresse für ihren Wert a175 und unter der Adresse a175 steht noch nichts. v soll den Wert einer anderen Variable w bekommen. Die Variable w hat als Adresse für ihren Wert a100. Der Wert von w sei ein Objekt, z.B. Utas blauer Ball. Dies Objekt ist im Speicher unter der Adress a200 zu finden. Im Speicherplatz 100 steht also „a200“. Nun soll v den Wert von w bekommen. Dazu wird die Adresse, die unter a100 zu finden ist, also a200, kopiert und die Kopie unter der Adresse a175 eingetragen. Die Beschreibung von Utas blauem Ball bleibt unverändert ab a200 stehen. Falls sich die Beschreibung von Utas blauem Ball ändert, so auch die Werte von v und w.

Name Adresse Wert
vorher: v a175 - w a100 a200
nachher: v a175 a200 w a100 a200

⋮

Abbildung 1.1.: Auszuge aus einem Skript vom Autor A

*Der Inhalt des Beispielskript ist aus [Morik 1998] entnommen.

Ein anderer Autor B hingegen formatiert Beispiele und Definitionen ganz anders. Abbildung 1.2 zeigt wie hier Definitionen eingerückt sind und die normale Schriftart »Roman« verwendet wurde. Beispiele benutzen, wie die Definitionen in Abbildung 1.1, die Schriftart *Italic* und sind hier allerdings nicht eingerückt.

⋮

Wenn wir aber eine gerichtete Assoziation haben, so geben wir nur entweder die Assoziation zwischen einer Studentin und den von ihr gehörten Vorlesungen an oder die Assoziation zwischen einem Professor und den von ihm gehaltenen Vorlesungen (s. Abbildung 4).

> **Definition 2.5: Assoziation** Eine Assoziation ist eine Verbindung von einer Klasse bzw. einem Objekt zu einer anderen Klasse bzw. einem anderen Objekt. Insbesondere kann eine Assoziation von n Objekten einer Klasse auf m Objekte einer anderen Klasse gerichtet sein.

Nun wissen wir was wir neben der Vererbungsbeziehung noch brauchen und es stellt sich die Frage, wie wir es darstellen wollen. Wir können Assoziationen als Eigenschaften und Methoden darstellen.

⋮

Nach diesen Präliminarien folgt das Wichtigste einer Klasse - der Klassenrumpf (ClassBody). Dies ist ein Block. Ein Block beginnt mit geschweifter Klammer und endet mit geschweifter Klammer. Meistens werden hier Methoden definiert. Dazu werden zunächst die Variablen deklariert und dann die Methode selbst.

Beispiel 3.1: In der Einleitung war von Utas blauem Ball die Rede. Dass er Uta gehört, soll bei Uta vermerkt werden. Allerdings können wir eine Klasse Besitz definieren und Bälle sind eine Unterklasse davon. Ausserdem haben Bälle immer eine Farbe. Ihre Methode besteht darin, einen Tritt in eine Ortsveränderung umzusetzen. Wir notieren den Tritt als zwei Zahlen im Koordinatensystem, die angeben, um wieviel der Ball sich auf der x- und um wieviel er sich auf der y-Achse verschieben soll.

Die Methoden erläutern wir später (Abschnitt 3.4). Hier erst einmal der Überblick über die Klasse Ball. Wir zeigen, dass die Klasse Ball gemäß der Grammatik (Tabelle 1) wohlgeformt ist.

⋮

Abbildung 1.2.: Auszuge aus einem Skript vom Autor B

Die Auszüge aus den Skripten der beiden Autoren zeigen, dass Regel für bestimmte Strukturen nicht für alle Dokumente benutzt werden können, auch wenn die Texte vom selben Typ sind.

Der zweite Lösungsansatz ist zwar weniger arbeitsaufwändig als der erste, allerdings sind die durch die Regelerstellung anfallenden Arbeitenschritte noch unzufriedenstellend.

Durch die Nachteile des zweiten Lösungsvorschlags ergibt sich die Motivation für einen dritten Ansatz: Die strukturcharakterisierenden Eigenschaften werden von einem System erkannt und zur Klassifikation der Dokumentstrukturen verwendet werden. D. h. zu Beginn bearbeitet das System ein beliebiges Dokument und extrahiert für jeden Typen der Dokumentstrukturen die jeweiligen Textattribute. Diese und vom Benutzer gegebene Beispielklassifikationen werden anschließend von einem Klassifizierungsalgorithmus, aus dem Bereich des Maschinellen Lernens, verarbeitet. Bei dieser Vorgehensweise entsteht eine Reihe von Vorteilen:

- geringer Arbeitsaufwand auch bei der Verarbeitung von langen Dokumenten
- es wird kein Fachwissen benötigt um komplizierte Regeln für Dokumentstrukturen aufzustellen
- unterschiedliche Formatierungsstile verschiedener Dokumente spielen keine Rolle, da Struktureigenschaften anhand von Beispielen gelernt werden

Von den genannten Vorschlägen erscheint der letzte Ansatz der vielversprechenste zu sein. Im Rahmen dieser Diplomarbeit wurde daher ein System entwickelt, dass Formatierungsmerkmale von Dokumentstrukturen verwendet, um damit, und unter zu Hilfenahme eines Klassifizierungsalgorithmuses, absatzbezogene Annotationen durchzuführen.

Im Bereich des maschinellen Lernens gibt es verschiedene Arten von Lernverfahren. ADT verwendet ein Attribut-Werte-Lernverfahren. Durch die gewählte Attribut-Werte Repräsentation der Daten ergibt es sich, dass grammatische Konzepte und Relationen nicht vorhanden sind. Da Relationen für die Erkennung von Absatzverschachtelungen wichtig sind, müssen die Relationen als Attribute dargestellt werden. Wie Kapitel 4 zeigt, können alle zweistelligen Relationen, bezüglich zweier aufeinanderfolgenden Paragraphen, durch vier Attribute approximiert werden. Inwieweit dieser Ansatz für geschachtelte Strukturen ausreichend ist und wo es Probleme geben kann, zeigen die später beschriebenen Versuche.

Ein zu nennender Vorteil, der sich durch die Verwendung der Attribut-Werte Repräsentation ergibt, ist, dass geschachtelte Strukturen beliebig viele innere Strukturen aufweisen können. Kapitel 4 erläutert wie der Beginn und das Ende der Strukturen erkannt wird. Ist dies geschehen, ist es irrelevant wie viele innere Elemente die Struktur aufweist. Genau hier liegt nun der Vorteil gegenüber regelbasierten Lernverfahren. Mithilfe von Logikausdrücken ist es sehr schwierig, beliebig viele innere Elemente zu beschreiben. Bei einer festen Anzahl der inneren Elemente wäre dies kein Problem. Da das System aber auf beliebige Dokumente angewendet werden soll, ist nicht bekannt, wie viele innere Elemente maximal vorkommen. Dieser Nachteil würde eine Einschränkung für den Anwender bedeuten.

Wortbezogene Annotationen bleiben in dieser Arbeit außen vor, da sich hier einige Probleme ergeben:

Weisen die Wörter in einem Dokument keine spezielle Formatierung auf, so können sie mit regulären Ausdrücken oder speziellen Grammatiken erkannt werden. Beispielsweise ist es möglich Uhrzeiten oder Datumsangaben durch reguläre Ausdrücke zu beschreiben. Da es jedoch sehr schwierig ist, Grammatiken und reguläre Ausdrücke maschinell zu lernen, wird dieser Ansatz hier nicht weiter verfolgt.

Sind die auszuzeichnenden Wörter hingegen speziell formatiert, können diese Wörter gerade aufgrund ihrer Formatierung erkannt werden. Das einzelne Wort ist hier also unwichtig, von Interesse ist seine Formatierung.

Der letzte Punkt, das Annotieren von Wörtern aufgrund ihrer Formatierung, ist prinzipiell realisierbar. Die Umsetzung ist allerdings aus zeitlichen Gründen nicht Teil dieser Arbeit; siehe dazu auch Kapitel 9.

Lediglich das erste Wort eines Absatzes wird bei der Klassifikation berücksichtigt. Kapitel 6 beschreibt genauer die daraus resultierenden Vorteile.

Anforderungen: Die Anforderungen an das zu entwickelte System lassen sich wie folgt stichpunktartig zusammenfassen:

1. Das System soll absatzbezogene und geschachtelte Auszeichnungen durchführen.
2. Die Klassifizierung von Struktureinheiten der Dokumente soll anhand von Textattributen erfolgen.
3. Dem Benutzer soll die Möglichkeit gegeben werden, interaktiv das Verfahren zu beeinflussen.
4. Mit dem System soll aktiv gearbeitet werden, die Klassifikation muss also sehr schnell ablaufen.
5. Das System soll ein von möglichst vielen Textverarbeitungsprogrammen unterstütztes Datenformat einlesen können.
6. Die bearbeiteten Texte sollen so abgespeichert werden können, dass sie neben absatzbezogenen und geschachtelten Annotationen keine visuellen Auszeichnungen mehr enthalten.
7. Der Benutzer soll die Möglichkeit erhalten die Metainformationen, welche die Dokumente annotieren, frei zu wählen.

Weiterhin sind folgende Randbedingungen wünschenswert:

1. Das Resultat einer Dokumentenbearbeitung sollte einfach zu begutachten sein. Ein Ausgabeformat wie XHTML würde die Darstellung in einem Web-Browser ermöglichen.
2. Ein plattformunabhäniges System brächte den Vorteil, dass auch Nutzer auf verschiedenen Rechnerarchitekturen das System ohne Anpassungsaufwand nutzen könnten.

1.2. Aufbau der Diplomarbeit

Dem Leser wurde bis zu diesem Punkt erläutert, welche Aufgabe innerhalb der Diplomarbeit bearbeitet werden soll und welche Lösungen für dieses Problem möglich sind. Zum Abschluss der Einleitung soll hier kurz der Aufbau der Diplomarbeit vorgestellt werden.

Das folgende Kapitel 2 gibt eine kurze Übersicht über verschiedene Typen und Sprachen, die es im Bereich der Markup-Sprachen gibt. Dies ist notwendig, da das im Rahmen der Diplomarbeit erstellte System Dokumente eines bestimmten Markup-Typs bearbeitet und in Dokumente eines anderen Typs umwandelt.

Das anschließende Kapitel 3 beschreibt andere, schon bestehende Systeme, denen ähnliche Aufgabenstellungen zu Grunde liegen. Im einzelnen sind dies die Systeme *WISDOM++*, *Slicing Book Technology* und *IP4W3*. Weiterhin werden Unterschiede und Gemeinsamkeiten in Bezug zum Diplomarbeitsthema erörtert.

Kapitel 4 gibt einen Überblick über das erstellte Programm ADT. Die verschiedenen Komponenten des System werden dabei im Einzelnen erläutert. Die Ganularität der Betrachtung geschieht allerdings nicht auf Implementierungsebene. Anfallende Probleme und Lösungen werden hier auf einer eher abstrakteren Ebene betrachtet.

Nachdem Kapitel 4 den Ablauf des Systems ADT näher gebracht hat, zeigt Kapitel 5 eine Anwendung des Systems am Beispiel. Am Ende dieses Kapitels sollte es dem Benutzer möglich sein, den praktischen Nutzen und die Benutzbarkeit des Programmsystems zu beurteilen.

Das folgende Kapitel 6 stellt mehrere Klassifikationsverfahren vor. Dabei wird erläutert, wie der Algorithmus ID3 von Quinlan [Quinlan 1983] arbeitet. ID3 liegt dem C4.5 Algorithmus [Quinlan 1993] zu Grunde, welcher wiederum die Vorlage des hier verwendeten J4.8 Algorithmuses [Witten und Frank 2000] darstellt. Anschließend werden die Versuche und Ergebnisse beschrieben, die durchgeführt wurden, um das System zu testen.

Kapitel 7 beschreibt die Evaluierung des ADT-Systems. Dabei wird untersucht, wie viele vom Benutzer gegebene Beispiele zu welchem Ergebnissen führen. Abschließend folgt eine Untersuchung der Laufzeit des Systems bezüglich verschiedener Algorithmen.

Das anschließende Kapitel 8 erläutert wie schon Kapitel 4 das Gesamtsystem. Der Fokus liegt hier allerdings auf Implementierungsebene. Die gegebenen Erklärungen behandeln einige wichtige Klassen und Methoden von ADT. Dieses Kapitel ist interessant für Anwender die das System verändern oder erweitern wollen.

Das folgende Kapitel überprüft, ob die gesetzten Ziele der Diplomarbeit erreicht wurden. Weiterhin diskutiert dieses Kapitel einige Verbesserungen und Erweiterungen, die an ADT durchgeführt werden können.

Der Anhang beschreibt kleine Erweiterungen innerhalb von ADT. Weiterhin wird erläutert, wie ADT auf verschiedenen Systemen installiert und gestartet wird.

2. Markup-Sprachen

Ursprünglich verstand man unter *Markup* das Markieren eines Dokuments durch einen Typographen, um dem Setzer mitzuteilen, wie das Dokument formatiert werden soll. Dies geschah in der Regel durch kleine handgeschrieben Notizen. Mit Einführung des Computers konnten diese Anmerkungen nun elektronisch kodiert werden. Die hierzu entwickelten Sprachen nannte man in Anlehnung an die alten Systeme *Markup-Sprachen*.

Auch heute noch enthalten Texte Steuerzeichen oder Makros, die Anweisung für die Formatierung geben. Zwar stellen die modernen WYSIWYG-Programme die Steuerzeichen nicht mehr auf dem Bildschirm dar, sondern können dank grafischer Oberfläche Attribute wie Fettdruck, Unterstreichungen und verschiedene Zeichensätze unmittelbar anzeigen, die Fixierung auf das Layout des Dokumentes herrscht jedoch noch vor [Behme und Mintert 2000].

Abbildung 2.1 zeigt einen Überblick über verschiedene Typen von Markup-Sprachen. Die folgenden Abschnitte erklären im Einzelnen die unterschiedlichen Sprachen und erläutern wodurch sie sich unterscheiden.

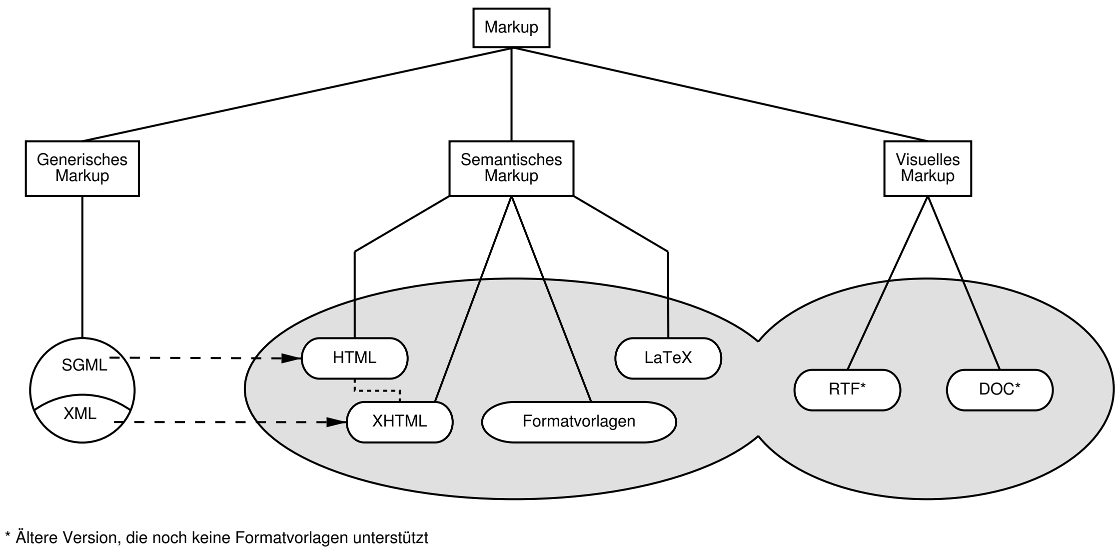

Abbildung 2.1.: Zusammenhang zwischen den verschiedenen Markup-Sprachen

2.1. Markup-Typen

In der Literatur werden unterschiedliche Typen von Markup-Sprachen behandelt. Überwiegend wird dabei eine Unterteilung in *generisches* und in *visuelles* Markup vorgenommen [Bryan 1988]. Nimmt man eine, wie in Abbildung 2.1 gezeigt, feinere Unterteilung vor, so lassen sich die Markup-Sprachen weiterhin in eine dritte Gruppe, die so genannten *semantischen* Markup-Sprachen, einteilen.

2.1.1. Generisches Markup

Generisches Markup ergänzt den Text um semantische Informationen, welche die Komponenten eines Dokuments nach logischen Gesichtspunkten beschreiben, zum Beispiel als Absatz, Überschrift oder Fußnote. Die Vorteile liegen auf der Hand. Bei der üblichen Vorgehensweise würde ein Verfasser beispielsweise eine Überschrift in einer größeren Schriftart setzen, ein Leser kann diesen Text durchaus als Überschrift erkennen, der Computer weiß jedoch nichts davon. Das Dokument enthält in diesem Fall nur Information darüber, wie ein Text darzustellen ist, aber nicht, welche Funktion er erfüllen soll.

An dieser Stelle setzt das generische Markup an. Die Markierungen sagen etwas über die Art der markierten Stelle aus. Dadurch ergibt sich der Vorteil, dass die Struktur des Dokumentes bei der Speicherung nicht verloren geht. Die Zuordnung einer bestimmten Darstellung zu einer bestimmten Klasse ist anwendungsspezifisch eindeutig möglich: Bei einem Ausdruck wären alle Verweise auf andere Kapitel kursiv gedruckt, bei der Darstellung im Web wären sie zusätzlich verlinkt.

Bei generischen Auszeichnungen können mit *wortbezogenen Annotationen* einzelne Worte ausgezeichnet werden. Sollen Satzteile annotiert oder Strukturen verschachtelt werden, verwendet man *geschachtelte Auszeichnungen*. Beispiele für wortbezogene und geschachtelte Annotationen finden sich im Beispiel 2.

Formatierungs- und Darstellungsanweisungen gibt es hier nicht. SGML und XML verwenden beispielsweise keinerlei Formatierungsanweisungen und fallen, wie in Abbildung 2.1 zu sehen, unter die generischen Auszeichnungssprachen. Die Darstellung übernimmt bei SGML die Formatierungssprache DSSSL[1] und bei XML die Formatierungssprache XSL[2]. Es liegt also eine Trennung zwischen Inhalt und Aussehen vor.

[1]**D**ocument **S**tyle and **S**emantic **S**pecification **L**anguage (ISO 10179:1996). Eine Sprache, die sowohl Formatierungs- als auch Transformationsmöglichkeiten beinhaltet und hauptsächlich für die Verarbeitung von SGML-Dokumenten eingesetzt wird [Mintert 2002].

[2]XSL steht für E**x**tensible **S**tyling **L**anguage und entstand 1997 aus DSSSL.

Beispiel 2:

```
<titel>
     Vorlesungsankündigungen
</titel>
<absatz>
     Die Lehrveranstaltung
     <veranstaltung>
          Künstliche Intelligenz
     </veranstaltung>
     wird im
     <termin>
          Wintersemester 2002/2003
     </termin>
     angeboten...
</absatz>
```

2.1.2. Semantisches Markup

Das semantische Markup dient ähnlich wie das generische Markup der Strukturierung von Dokumenten. Die Einschränkung, keine Formatierungs- und Darstellungsanweisungen zu verwenden, gibt es hier nicht. Neben der Markierung von verschiedenen Abschnitten, beispielsweise als Überschrift oder Zitat, können in einem Text zusätzliche Formatierungsangaben, wie kursiv, fett oder unterschiedliche Schriftarten, verwendet werden.

Beispiel 3 zeigt einen mit LaTeX formatierten Text. Neben Textstrukturen die als Sektionen markiert sind wurden Formatierungsanweisungen (hier bold face) verwendet, die den Text nicht nach logischen Gesichtpunkten strukturieren.

Beispiel 3:

```
\section{Vorlesungsankündigungen}
\subsection{Künstliche Intelligenz}
Die Lehrveranstaltung {\bf Künstliche Intelligenz} wird im
Wintersemester {\bf 2002/2003} angeboten...
```

2.1.3. Visuelles Markup

Visuelles Markup ergänzt den Text um Formatierungsanweisungen, die das Aussehen des Dokuments genau festlegen. Die Anweisungen haben einen direkten Effekt auf das

Erscheinungsbild des Textes [Bryan 1988]. Es handelt sich also um Auszeichnungen, die der reinen Layoutformatierung dienen.

Die Anweisungen, geben zum Beispiel an, ob ein Text kursiv, unterstrichen oder in einer bestimmten Größe dargestellt werden soll [Kobert 1999]. Anwender wollen mit solchen Formatierungen oft bestimmten Textabschnitten spezielle Funktionen, wie Überschriften oder Schlüsselwörter, zuordnen. Die Auszeichnungen enthalten allerdings keine Informationen über den Inhalt des Textes, es werden also keine Informationen geben, ob es sich bei einen Textstück beispielsweise um eine Definition, eine Überschrift oder eine ähnliche Struktur handelt.

Wie ein mit visuellem Markup formatierter Text aussieht, zeigt das Beispiel 4.

Bei den Dateiformaten der klassischen Textverarbeitungsprogramme, die noch keine Formatvorlagen unterstützen, wird diese Art der Formatierung meist angewandt, so auch beim in Abschnitt 2.2.7 beschriebenen RTF-Format.

Eine nachträgliche Änderung des Textes erfordert meist große Umstrukturierungen des Markups, eine maschinelle Verarbeitung ist so gut wie nicht möglich. Die größte Gefahr des visuellen Markups ist jedoch, dass durch unkontrolliert eingesetzte Effekte der eigentliche Inhalt in den Hintergrund gerückt wird, und so das einheitliche Bild, das ein zusammenhängender Text haben soll, verloren geht. Typographisches Design ist ein Handwerk, das erlernt werden muss. Ungeübte Autoren machen oft gravierende Formatierungsfehler. Fälschlicherweise glauben viele Laien, dass Buchdruck-Design vor allem eine Frage der Ästhetik ist, wenn das Schriftstück vom künstlerischen Standpunkt aus schön aussieht, dann ist es schon gut »designed«. Da Schriftstücke jedoch gelesen und nicht in einem Museum aufgehängt werden, sind die leichtere Lesbarkeit und bessere Verständlichkeit wichtiger als das schöne Aussehen [Knappen et al. 1995].

Ein Spezialfall des visuellen Markups ist das Konzept des »What you see is what you get« (WYSIWYG). Der Benutzer eines WYSIWYG-Editors sieht das Dokument an dem er arbeitet ständig in seiner endgültigen Form. Dies scheint auf den ersten Blick vorteilhaft zu ein. Allerdings ergibt sich das Problem, dass der Autor darauf achten muss, dass gleiche Textelemente wie z. B. Definitionen optisch immer gleich aussehen. Gerade bei sehr langen Dokumenten erweist sich dies allerdings als ein Problem. Mit WYSIWYG Entwurfsystemen erzeugen Autoren im Allgemeinen ästhetisch schöne, aber schlecht strukturierte Schriftstücke [Knappen et al. 1995].

Beispiel 4:

```
{\b0\i0 Vorlesungsank\'fcndigungen
\par }\pard\plain \ql \li0\ri0\widctlpar\aspalpha \aspnum
\faauto\adjustright\rin0\lin0\itap0 \fs24\lang1031\langfe1031
\cgrid\langnp1031\langfenp1031 {\f1 Die Lehrveranstaltung }
{\b\f1 K\'fcnstliche Intelligenz}{\f1 wird im }
{\b\f1 Wintersemester 2002/2003}
{\f1 angeboten... \par }
```

2.2. Markup-Sprachen

Zu den im vorherigen Abschnitt beschiebenen Typen von Markup-Sprachen gibt es eine Reihe von Sprachen die im Folgenden beschrieben werden.

Im Bereich des semantischen und des visuellen Markups lassen sich nicht alle Sprachen exakt einem Typ zu ordnen. Wie auch Abbildung 2.1 zeigt, überschneiden sich die Mengen der Sprachen. Die Grenzen sind hier fließend. Mit dem RTF-Format lassen sich z. B. nicht nur Formatierungsangaben kodieren, es werden in einem Dokument auch semantische Informationen, wie der Name des Autors und die Organisation für die er tätig ist, gespeichert.

2.2.1. SGML

Geschichte Die grundlegende Idee von SGML, nämlich das Konzept des *generic coding*, ist bereits über dreißig Jahre alt. William Tunnicliffe von der Graphic Communications Associaion (GCA) machte im September 1967 in seinem Vortrag, *»The seperation of information content of documents from their format«* [Goldfarb 1996], den Vorschlag, den Informationsgehalt eines Dokumentes von seiner äußeren Form zu trennen [Behme und Mintert 2000]: Auf dieser Idee aufbauend entwickelten Charles Goldfarb, Edward Mosher und Raymond Lorie 1969 bei IBM die *Gerneralized Markup Language (GML)*. GML enthielt erstmals das Konzept eines formal definierten Dokumenttyps mit einer verschachtelten Struktur.

Auf GML basierend wurde Ende der siebziger Jahre - zunächst vom American National Standard Institute (ANSI) ausgehend, später auch in Abstimmung u.a. mit der International Organization for Standardization (ISO) - mit der Entwicklung einer standardisierten Textbeschreibungssprache begonnen. Diese Arbeit endete 1986 mit der Veröffentlichung der *Standard Generalized Markup Language* im ISO-Standard 8879. Diesem folgte 1988 noch eine Änderung, die lediglich ergänzenden Charakter hatte und inhaltliche Ungenauigkeiten beseitigen sollte.

SGML in der Praxis Mit SGML wird eine Syntax bereitgestellt, mit der die Struktur von Dokumenten eines Typs beschrieben werden kann. Ein Dokument wird dabei nur strukturell und nicht typographisch beschrieben d.h. es wird unabhängig von der weiteren Verwendung und Darstellung durch die verschiedenen Textelemente charakterisiert.

Die Strukturdefinition eines Dokumententyps erfolgt in der *Document Type Definition (DTD)*. Diese enthält Informationen über den Aufbau eines Dokumentes des zu spezifizierenden Typs, d.h. in der DTD wird festgelegt, welche Elemente ein solches Dokument enthalten darf und/oder enthalten muss und in welcher Reihenfolge diese erscheinen dürfen.

Im konkreten Dokument, *Instanz* genannt, wird die Zugehörigkeit des Textes zu bestimmten - in der DTD definierten - Strukturelementen durch Start- und End-Tags gekennzeichnet. Die Tags beschreiben also nur um welche Art von Information es sich handelt, nicht wie diese dargestellt werden soll.

Beispiel 5:

```
<titel>
     Dieses ist der Titel
</titel>
```

Eine Instanz enthält außer dem eigentlichen Text mit den entsprechenden Kennzeichnungen (Markup) noch eine Referenz auf die zugehörige DTD. Ein SGML-Parser überprüft anhand der DTD die syntaktische Korrektheit und Vollständigkeit des Dokumentes.

Welches Aussehen die verschiedenen Elemente eines Textes letztendlich erhalten, ist von den system- und medienspezifischen Hilfsmitteln abhängig. Zunächst bringt ein Konverter das SGML-Dokument in eine syntaktische Form, die das jeweilige Formatierungsprogramm versteht. Dieser Formatierer (z.B. TeX) bereitet das Dokument dann entsprechend zur Ausgabe auf.

2.2.2. XML

XML (Extensible Markup Language) ist eine Entwicklung des W3-Consortium[3] (W3C), welches Normen festlegt und die Entwicklung neuer Standards im Internet vorantreibt. So befasst sich das W3C nicht nur mit XML, auch HTML, SGML oder VRML, um nur einige Beispiele zu nennen, werden hier in der Entwicklung koordiniert [Kobert 1999].

XML ist genau wie SGML ein System zur Markierung von Informationen und zur Steuerung ihrer Struktur, da XML eine deutlich abgespeckte Version von SGML ist, die noch genug von dessen Funktionalität bietet, um nützlich zu sein, gleichzeitig aber alle optionalen Features entfernt, die SGML für den normalen Gebrauch zu komplex machen [Lobin 2001].

XML ist also ein SGML-Profil. Wie in Abbildung 2.1 zu sehen, ist XML eine Teilmenge von SGML mit der sich ebenfalls neue Sprachen definieren lassen. Dies macht XML wie auch schon SGML zu einer Metasprache[4].

2.2.3. HTML

HTML bedeutet *Hyper Text Markup Language*. Es handelt sich hierbei um eine Sprache die 1989 von Tim Berners-Lee mit Hilfe von SGML definiert wurde. HTML ist also eine SGML-Anwendung [Lobin 2001].

[3] `http://www.w3c.org`

[4] Mit einer Metasprache lassen sich nicht direkt Dokumente erstellen, sondern verschiedene Sprachen (Auszeichnungssprachen) definieren, mit denen dann Dokumente erstellt werden können. Als Beispiele für Metasprachen sind XML und SGML zu nennen.

HTML ist die Sprache des World Wide Web. Nahezu alle Webseiten verwenden HTML um Erscheinungsbild und Funktion der Seite zu beschreiben. HTML ist dementsprechend auch eine Auszeichnungssprache. Da HTML im Gegensatz zu XML keine Metasprache ist, erlaubt es HTML nicht neue Befehle zu definieren.

2.2.4. XHTML

Anfang 2000 hat das W3-Consortium HTML 4.0 als eine Anwendung von XML 1.0 umgesetzt [Behme und Mintert 2000] und unter dem Namen XHTML veröffentlicht.

ADT verwendet XHTML als Zielformat zur Ausgabe des strukturierten Dokumentes. Der Grund hierfür ist einfach, dass XHTML von jedem modernen Webbrowser gelesen werden kann.

2.2.5. Formatvorlagen

Schreibt man längere Texte, ist es meist schwierig einen einheitlichen Stil beizubehalten. Zu Beginn eines Textes formatiert man bestimmte Textelemente wie Zitate oder Beispiele auf eine bestimmte Art und Weise. Will man einige Seiten später das selbe Element entsprechend formatieren, kann man sich meist nicht mehr daran erinnern. Im Textverarbeitungsprogramm *Word* gibt es für dieses Problem eine Lösung. Die so genannten Stylesheets. Sie erlauben es für bestimmte Textelemente Namen zu vergeben und diese mit einem bestimmten Format zu verknüpfen. Dadurch ist es dann auch möglich die Formatierung nachträglich ohne großen Aufwand zu ändern.

2.2.6. LaTeX

Im Mai 1977 begann Donald Knuth von der Standford University mit der Entwicklung des Textverarbeitungssystem TeX und METAFONT. Einer der größten Vorteile den TeX bietet, ist dass sich der Autor auf den Inhalt seines Dokuments konzentrieren kann, die Formatierung steht außen vor. Er kann die meisten Dokumente auf einfache Art und Weise mit Hilfe weniger Befehle erstellen. Um Einzelheiten der Gestaltung braucht er sich dabei nicht zu kümmern. Diese wurden von Designern in Layoutvorlagen (Styles) festgelegt. Eine Erweiterung mit der sich auch Literaturverzeichnisse, Querverweise und Inhaltsverzeichnisse erzeugen lassen, stellt das zu Beginn der achtziger Jahre von Leslie Lamport entwickelte, LaTeX dar [Goossens et al. 2000]. Insbesondere im wissenschaftlichen Bereich ist LaTeX sehr verbreitet.

LaTeX gestattet die Erstellung und Verwendung von selbst definierten Befehlen, die ähnlich wie Formatvorlagen verwendet werden können.

2.2.7. RTF

Die aus dem Bereich des visuellen Markup am häufigsten verwendete Sprache ist das, Mitte der 80er Jahre, von Microsoft spezifizierte Rich Text Format (RTF). Es enthält ausschließlich[5] typographische und visuelle Auszeichnungen, jedoch keine Metadaten, die Informationen über einzelne Textausschnitte liefern. RTF Dateien dienen der Kodierung von formatierten Texten und Graphiken, um einen einfachen Austausch zwischen verschiedenen Programmen, Rechnerarchitekturen und Ausgabegeräten zu gewährleisten. Die meisten aktuellen Textverarbeitungprogramme sind in der Lage RTF zu lesen und ebenso zu generieren.

Die Formatierung des Textes wird mit Hilfe folgender Elemente ausgeführt:

- **control words:** In der aktuellen RTF Spezifikation 1.7, vom August 2001, gibt es ca. 1375 control words, wobei die maximale Länge eines Kontrollwortes 32 Zeichen beträgt. Kontrollwörter werden zur Textformatierung verwendet. Außerdem können sie ebenfalls Informationen für die Verarbeitung in Programmen enthalten. Ein Kontrollwort wird stets durch einen Backslash (\) eingeleitet. Das Ende eines Kontrolwortes wird durch einen *Delimiter* (Begrenzer) definiert. Das können Leerzeichen, Zahlen (positiv oder negativ) oder jedes andere Zeichen, das kein Buchstabe ist, sein. Begrenzt eine Zahl ein Kontrollwort so wird diese als sein Wert aufgefasst.

- **control symbols:** Ein Steuersymbol beginnt mit einem Backslash (\) gefolgt von einem weiteren nicht-alphabetischen Zeichen. Es sind nur wenige Steuerzeichen vereinbart, die jedoch durch einen RTF-Scanner übergangen werden können. Steuersymbole besitzen gegenüber Kontrollwörtern keinen Delimiter [Born 2001].

- **groups:** Eine Gruppe schließt eine Menge von Kontrollwörtern und einen Teil des zu formatierenden Textes in Geschweifteklammern ({}) ein. Formatierungen die innerhalb einer Gruppe definiert werden, haben außerhalb der Gruppe keine Gültigkeit. Auf der anderen Seite werden hingegen Formatierungen an Subgruppen vererbt.

- **destination:** Destinations definieren einen zusammenhängenden Text der im formatierten Dokument an anderer Stelle auftaucht. Beispielsweise werden Einträge im Inhaltsverzeichnis eines Dokumentes durch destinations festgelegt. Der entsprechende Text erscheint dann nicht an der Stelle an der er im RTF-Dokument steht, meist am Beginn von Kapitel oder Abschnitten, sondern im Inhaltsverzeichnis. Destinations werden mit »*« eingeleitet und bilden stets eine Gruppe.

  Microsoft führte Destinations erst nach der Veröffentlichung der ersten RTF Spezifikation im März 1987 ein. Bis heute wurden ständig neue destinations hinzugefügt, was dazu führte, dass einige RTF-Reader unbekannte destinations ignorieren. Auch ADT verfährt auf diese Weise, da durch sie keine nützlichen Informationen zur Lösung der Aufgabe gewonnen werden können.

Abbildung 2.2 zeigt die Syntax einer RTF Datei. Es ist zuerkennen, dass eine RTF Datei zuerst einmal aus vier Elementen besteht. Dies sind zum einen die Literale »{« und »}«

[5] genau genommen gibt es, wie schon in Abschnitt 2.2 erwähnt, kleine Ausnahmen. Außerdem werden die, erst in neueren Versionen verfügbaren, Formatvorlagen ausgeschlossen.

und zum anderen die Gruppen *header* und *document*. Die beiden letztgenannten können nun weiterabgeleitet werden. Ein RTF Parser leitet die Knoten des Syntaxbaumes bis zu seinen Blättern, den terminalen Symbolen, ab. Aus Platzgründen konnte hier nicht der komplette Syntaxbaum aufgezeichnet werden.

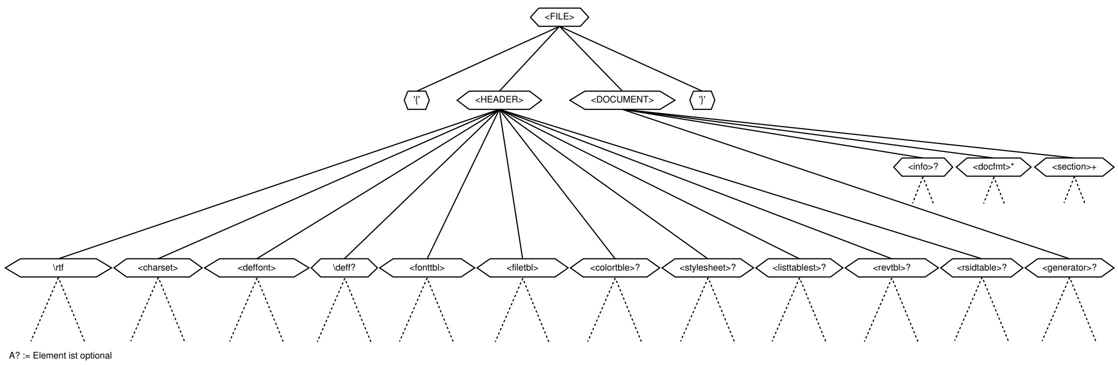

Abbildung 2.2.: RTF Syntaxbaum [ric 2001]

2.2.8. Word - DOC

Dokumente die mit Textverarbeitungsprogrammen erstellt wurden, sind in den wenigsten Fällen im ASCII-Code gespeichert. Nahezu jedes Textverarbeitungsprogramm besitzt sein eigenes Format, und leider legen nur sehr wenige Firmen die interne Datenstruktur offen. Daher ist es nicht leicht Informationen über diese Formate zu bekommen.

Das sehr weit verbreitete Textverarbeitungsprogramm *Word*, der Firma *Microsoft*, weist genau die genannten Probleme auf:

- es benutzt ein gemischtes ASCII-/Binärformat zur Abspeicherung der erstellten Textdateien
- es ist keine Spezifikation zum DOC-Format öffentlich verfügbar

Aufgrund des Binärformates und der fehlenden Formatbeschreibungen ist eine Verarbeitung von Dokumenten die mit *Word* erstellt wurden nur sehr schwer möglich.

In älteren *Word*-Versionen war es noch nicht möglich Formatvorlagen zu definieren, ein erstelltes Dokument konnte nur mit reinen Formatierungsattributen wie Schriftgröße und -art formatiert werden. Strukturierende Metainformationen konnten nicht eingefügt werden. Daher fällt die Formatierung, die alte *Word*-Versionen vornehmen, unter das visuelle Markup. Werden hingegen Formatvorlagen verwendet, können beliebige Strukturen ausgezeichnet werden. Damit fällt das DOC-Format nicht mehr unter die reinen visuellen Auszeichnungssprachen, sondern kann auch zu den semantischen Markup-Sprachen gezählt werden.

3. Abgrenzung zu bestehenden Systemen

Dieses Kapitel stellt drei verschiedene Systeme vor, die auf verschiedene Art und Weise strukturierte Dokumente erzeugen bzw. nutzen. Zuerst wird in Abschnitt 3.1 das an der Universität von Bari entwickelte System WISDOM++ vorgestellt. Das Programm scannt Dokumente ein und wandelt es in ein strukturiertes XML-Dokument um. Abschnitt 3.2 führt in die Slicing Book Technologie ein. Die an der Universität Koblenz-Landau entwickelte Technik bietet die Möglichkeit vorhandene Texte aufzuarbeiten und um Metainformationen zu ergänzen, wodurch die Dokumente in zusammenhängende Bereiche geteilt werden. Anschließend, in Abschnitt 3.3, wird das von Stefan Mintert entwickelte System IP4W3 vorgestellt. Das Programm ermöglicht eine komfortable Suchfunktion innerhalb verschiedener Kategorien.

3.1. WISDOM++

An der Universität Bari wurde ein System entwickelt, mit dem es möglich ist wissenschaftliche Arbeiten, die als Ausdruck auf Papier vorliegen, in ein für das Internet geeignetes Format, z.B. HTML/XML, zu transformieren [Altamura et al. 2000]. Die Umwandlung in HTML/XML ist dabei aus mehreren Gründen sinnvoll:

- schnelle Verfügbarkeit im Netz gegenüber einem gescannten Bild
- Suche nach Begriffen und Verlinkung von zusammengehörigen Abschnitten ist möglich
- XML-Dokumente besitzen auf Grund ihrer DTD[1] eine logische Struktur

Man könnte nun einwänden, dass auch normale OCR-Systeme[2] in der Lage sind HTML zu erstellen. Dabei tritt allerdings das Problem auf, dass das Aussehen der so entstehenden Dokumente meist nicht mehr dem Original ähnelt. WISDOM++[3] versucht hingegen ein möglichst originalgetreues Dokument zu erstellen. Der Grund, warum normale OCR-Systeme dies nicht bieten können, ist nach [Wang et al. 1999], dass diese Systeme keine

[1]Festlegungen über Aufbau, Struktur und gegenseitige Beziehungen von Elementen, Attributen und Entities von XML- oder SGML-Dokumenten

[2]**O**ptical **C**haracter **R**ecognition - Optische Schrifterkennung, der Prozess der Analyse auf Papier gedruckter Zeichen, um ihre Form durch die Erkennung von Mustern dunkler und heller Bereiche zu bestimmen

[3]Desweiteren als WISDOM bezeichnet.

Analyse der Dokumente durchführen, in der die Funktion der einzelnen Textteile untersucht werden. Um Texte in XML zu transformieren ist es nötig, Wissen über das Layout und die Struktur von Texten zu erlangen.

Das WISDOM-System arbeitet in fünf Schritten:

1. **Dokumenten-Analyse**
   Analysiert die hierarchische Layout-Struktur.

2. **Dokument-Klassifizierung**
   Erkennt die Dokumentenklasse, damit das richtige Stylesheet und die richtige DTD für das momentan bearbeitete Dokument verwendet wird.

3. **Dokument-Verständnis**
   Extrahiert den Text und definiert damit den eigentlichen Inhalt des XML-Files

4. **Text-Erkennung**
   siehe Punkt 3.

5. **Transformation in XML-Format**
   Das endgültige XML-Dokument wird hier aufgrund der vorherigen Analysen geschrieben.

Desweiteren wird nun die grobe Arbeitsweise des Systems vorgestellt:

Soll ein Paper, das nur als Ausdruck vorliegt, in ein für das Internet geeignetes Format gebracht werden, muss es zuerst einmal optisch gescannt werden. WISDOM übernimmt diesen Schritt, scannt also zuerst das zu bearbeitende Dokument mit 300 dpi ein, korrigiert eine eventuelle Verdrehung und speichert das Bild anschließend im Grafikformat TIFF[4] ab.

Im folgenden Schritt wird das gescannte Dokument mit Hilfe einer Variante des Run Length Smoothing Algorithmuses (RLSA) [Wong et al. 1982] in rechteckige Blöcke eingeteilt, welche entweder Text oder Grafik enthalten können. Mit Hilfe eines Entscheidungsbaumes werden die einzelnen Blöcke automatisch klassifiziert. Die Klassen die hierbei von WISDOM benutzt werden sind beispielsweise *Textblock, horizontale Linie, vertikale Linie, Bild* und *Grafiken*. Pro Seite entstehen so normalerweise weniger als 100 dieser Blöcke. Um den erwähnten Entscheidungsbaum zu erhalten müssen zuerst vom Administrator für jede Klasse ein Menge von Trainingsbeispielen gelernt werden. WISDOM verwendet hierfür eine verbesserte Variante des 1993 von Ross Quinlan [Quinlan 1993] vorgestellten C4.5 Algorithmuses.

Die anschließend durchgeführte Layoutanalyse erlaubt es eine Struktur innerhalb des Dokumentenbildes zu erkennen. Sie gruppiert die Blöcke zu einer Menge von so genannten Rahmen. Eine ideale Layoutanalyse erzeugt eine Menge von Rahmen. Jeder dieser

[4] **T**ag **I**mage **F**ile **F**ormat - Ein herstellerübergreifendes Format, das von Aldus, HP und Microsoft definiert wurde. Mittlerweile unterstützen viele Hersteller, insbesondere im Scannerbreich, dieses Format. Es lassen sich monochrome oder farbige Bilder als Bitmap-Grafik ablegen [Born 1995].

Rahmen entspricht einer logischen Komponente wie zum Beispiel *Titel* oder *Autor* einer wissenschaftlichen Arbeit. In WISDOM wird eine neue in C++ geschriebene Version des Layoutanalyse-Systems LEX [Esposito et al. 1995] verwendet. Dieses System fasst die einzelnen Rahmen zusammen, so dass zusammengehörige Einheiten, wie z. B. Paragraphen, Abschnitte oder Abbildungen, entstehen. Abbildung 3.1 zeigt die einzelnen Schritte die bei der Bearbeitung eines Dokumentes anfallen.

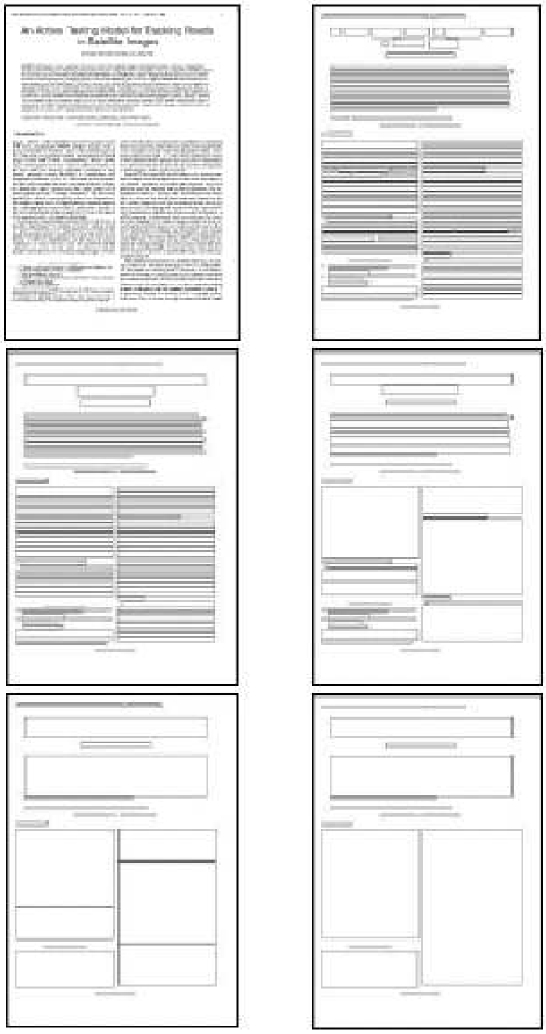

Abbildung 3.1.: Ein gescanntes Dokument (oben links) und die Erkennung von einfachen Blöcken, Zeilen und Rahmen [Altamura et al. 2001]

Die zuvor gefundene Layoutstruktur muss nun in eine logische Struktur abgebildet werden. Logische Struktur bedeutet hierbei, dass zum Beispiel der Absender bzw. Empfänger eines Briefes oder der Autor einer wissenschaftlichen Arbeit auch als solcher erkannt wird. In WISDOM geschieht dies mit Hilfe von Algorithmen des maschinellen Lernens[5].

[5] Detaillierte Beschreibungen zum induktiven Lernen von Dokumentklassifikationen finden sich in [Esposito et al. 2000]

Nachdem nun auch die logische Struktur des Dokuments bekannt ist, kann WISDOM eine HTML/XML Version des Dokumentes erstellen, welche im günstigsten Fall optisch identisch zum Originaldokument ist.

3.1.1. Vergleich von WISDOM und ADT

Das WISDOM-System betrachtet bei der Erstellung von XML-Dokumenten einzelne zusammengehörige, kleine Blöcke von Zeichen und Wörter die zu größeren Blöcken zusammengefasst werden. Am Ende der Zusammenfassungen entsprechen die einzelnen Blöcke, wie z. B. Überschriften oder Zusammenfassungen. WISDOM arbeitet also wie auch ADT auf Paragraphenebene. Die Klassifikation der Absätze geschieht bei beiden System mit Hilfe eines C4.5 Algorithmuses. Klassifiziert wird dabei nicht in Hinblick auf den Inhalt, sondern in Hinblick auf die physikalische Darstellung des Dokumentes. Der Unterschied besteht in der Art der Eingabedaten: Auf der einen Seite klassifiziert WISDOM Bildausschnitte (Erscheinungsformen) und ADT klassifiziert auf der anderen Seite Formatierungsanweisungen aus RTF-Dokumenten.

3.2. Slicing Books

Ein an der Universität Koblenz-Landau entwickelte Technik, Slicing Book Technology [Dahn 2001a], bietet die Möglichkeit Lerninhalte speziell darzustellen und zur Verfügung zu stellen. Insbesondere Fernuniversitäten könnten von dieser Technik profitieren. Gerade sie benötigen Literatur mit hoher inhaltlicher Qualität, da ihre Studenten ihr Wissen zum größten Teil aus Literatur und nicht wie Studenten an herkömmlichen Universitäten in Vorlesungen bzw. im Frontalunterricht erlangen [Dahn 2001a] [Valerius et al. 2001].

Ein Großteil der Literatur soll den Studenten über das Internet zur Verfügung gestellt werden. Da immer mehr virtuelle Schulen entstehen, tritt das Problem auf, dass eine große Menge virtueller Bücher erstellt werden muss. Die Slicing Book Technologie eröffnet die Möglichkeit existierende Bücher in eine elektronische Form zu konvertieren. Darüberhinaus versucht diese Technik dem Lernenden genau die Informationen zu liefern, die er benötigt. D. h. es werden je nach Wissensstand des Benutzers verschiedene Lerninhalte zu ein und dem selben Thema zur Verfügung gestellt. Die Slicing Book Technology zerlegt hierzu strukturierte Dokumente in so genannte Lernobjekte. Ein Lernobjekt kann sehr klein sein, zum Beispiel ein einzelnes Beispiel oder eine einzelne Tabelle. Die aufbereiteten Inhalte werden anschließend auf einem Webserver abgelegt, so dass sie immer verfügbar und aktualisierbar sind.

Einen Vorteil den die Slicing Books gegenüber anderen Büchern bieten möchten ist, dass der Benutzer auf der einen Seite die Möglichkeit hat auf eine große Fülle von Daten zuzugreifen, auf der anderen Seite hingegen werden ihm vom System nur die Inhalte angeboten, die er benötigt. Der Benutzer profitiert also von allen Vorteilen die ein normales elektronisches Buch liefert, wie z. B. Suchfunktionen und Verknüpfung von Inhalten und Begriffen durch Hyperlinks. Zusätzlich wird die Möglichkeit gegeben ein persönliches

Dokument mit allen, aufgrund des Wissenstandes des Benutzers, benötigten Inhalten zu erstellen und diese auszudrucken. Um dies Dokument anzufertigen macht das System, wie schon erwähnt, Annahmen[6] über den Wissenstand des Benutzers, weiterhin werden Vorlieben und Interessen des Benutzers einbezogen. Je nachdem wie diese Analyse ausfällt werden dem Anwender unterschiedliche Inhalte vorgeschlagen.

Im folgenden Abschnitt wird die Funktionsweise der Slicing Books kurz beschrieben. Eine ausführlichere Beschreibung findet sich in [Dahn 2001a], [Dahn 2001b] und in [Dahn et al.]. Für die Slicing Book Technologie eignen sich alle gut strukturierten Dokumente. Der erste Schritt bei der Erstellung eines Slicing Books aus einem existierenden Dokument ist es diese Struktur sichtbar zu machen. Das Dokument wird in so genannte *Slices*[7] zerlegt. Kapitel, Abschnitte, Beispiele, Definitionen oder Übungen sind nur ein paar Beispiele für Slices. Je feiner diese Strukturierung ist, desto flexibler ist später die Erstellung von Dokumenten für den Benutzer.

Damit die Slices genutzt werden können, müssen sie mit Hilfe von Metadaten beschrieben und ergänzt werden. Die Metadaten werden zum Teil automatisch bei der Erstellung des Slicing-Prozesses aus dem Text extrahiert und zur Beschreibung einer Slice benutzt. Ein Beispiel sind die durch den Autor markierten Schlüsselwörter. Um noch weitere Schlüsselwörter zu extrahieren werden Indexierungstechniken angewandt, die meist bei der Indexierung von Webseiten Anwendung finden.

Die automatische Verarbeitung eines Buches zum Slicing Book erfolgt leider nicht zuverlässig [Dahn 2001a]. Um möglichst kleine Slices zu erhalten, ist eine weitere manuelle Bearbeitung durch Fachpersonal nötig. Dies geschieht mit der so genannten Autorenumgebung. Hiermit kann noch einmal die Struktur des Sliced Books überprüft werden und darüberhinaus hat der Benutzer die Möglichkeit Metadaten hinzuzufügen und zu überprüfen.

Folgende Metadaten müssen hier nach [Dahn 2001a] behandelt werden:

- Verweise die vorausgesetztes Wissen beschreiben
- Verweise zu anderen Quellen, wie z. B. Multimediamaterialien oder Quellen im World Wide Web
- Verweise zu anderen Slices die nötig sind um der Slice einen Sinn zu geben
- Schlüsselwörter die den Inhalt der Slices beschreiben
- den Typ einer Slice

Ein fertiges Slicing Book steht zu demonstrationszwecken auf der Homepage der Firma Slicing Information Technology GmbH[8] zur Verfügung.

[6]Die Annahmen basieren entweder auf der Selbsteinschätzung der Benutzer, werden durch standardisierte Tests oder durch eine dritte Person erstellt.
[7]engl. Scheiben
[8]`http://www.slicing-infotech.de`

3.2.1. Vergleich der Slicing Book Technologie und ADT

Die Slicing Book Technologie versucht Dokumente um Metadaten zu erweitern, welche beispielsweise den Inhalt eines Textabschnitts beschreibt oder zusammengehörige Textfragmente kennzeichnen. Die hierbei vorgenommen Annotationen können absatz- oder wortbezogen sein. Im Gegensatz zu WISDOM oder ADT beziehen sich die Annotationen auf den Inhalt des Dokumentes. Die Erstellung der Auszeichnungen erfolgt nach [Dahn 2001a] maschinell sowie manuell. Wie die maschinelle Verarbeitung genau durchgeführt wird, bleibt allerdings offen.

3.3. IP4W3

IP4W3 (Intelligent Publishing for the World Wide Web) [Mintert 1999] ist ein von Stefan Mintert entwickeltes System, das im Rahmen seiner Diplomarbeit, am Lehrstuhl für Künstliche Intelligenz[9] der Universität Dortmund, entstand. IP4W3 bereitet SGML/XML-kodierte Dokumente für das World Wide Web auf und gestattet dem Leser, in für ihn angenehme Weise, auf die Inhalte zuzugreifen.

Um lange Texte zu lesen, bevorzugen die meisten Menschen die ausgedruckte Form. Ein Vorteil bei elektronischen Dokumenten ist allerdings, dass hier die Möglichkeit besteht die Texte zu durchsuchen. In der Regel liefert eine übliche Volltextsuche allerdings zu viele und meist unerwünschte Ergebnisse. Alle die bereits mit solchen Systemen gearbeitet haben kennen das Problem, man sucht nach dem Programmiersprachenbefehl `print` und das System findet Rezepte für Aachener Printen und Texte über Print-Medien. Das Problem besteht darin, dass man zwar den Suchbegriff angeben kann, nicht jedoch die Suchkategorie (hier: Begriff `print`, Kategorie: »Befehle von Programmiersprachen«) [Mintert 1999]. Die Ursache hierfür ist, dass die meisten Texte keine Metainformation, wie die Kategorien, enthalten. Mit IP4W3 hat der Benutzer die Möglichkeit, in zuvor um Metadaten ergänzten Dokumenten, nach Stichworten, in einer von mehreren Kategorien, zu suchen. Dadurch kann die häufig ungenaue Volltextsuche entscheident verbessert werden.

Als Ergebnis einer Suche liefert IP4W3 nicht nur die Texte zurück, in denen der Suchbegriff existiert und sich in der richtigen Kategorie befindet. Das Ergebnis wird darüberhinaus noch von der Stelle, an der sich der gefundene Begriff befindet, abhängig gemacht. Um die Suchfunktion noch komfortabler zu gestalten, hat Stefan Mintert die Suche so entwickelt, dass als Ergebnis einer Suche nicht nur das gefundene Wort zurückgeliefert wird, sondern eine bestimmte Umgebung des Wortes. Die Umgebung ist dabei so beschaffen, dass der Text darin verständlich bleibt.

Zu guter Letzt hat der Benutzer noch die Möglichkeit gefundene Textausschnitte in einer Art Warenkorb zu sammeln und kann sich so sein »eigenes« Dokument erstellen.

Um die oben genannten Merkmale dem Benutzer zur Verfügung stellen zu können, ist es nötig vorhandene Texte mit Metainformationen aufzubereiten. IP4W3 sieht hierfür

[9] `http://www-ai.cs.uni-dortmund.de/DOKUMENTE/mintert_99a.pdf`

ein eigenes Tool vor, um den Administrator, der die Texte aufbereitet, zu unterstützen und anzuleiten. Damit der Administrator nicht für jeden Text den selben Aufwand treiben muss, arbeitet das System mit Dokumententypen (z.B. Lehrbuch, Handbuch, Bedienungsanleitung, Brief, Artikel usw.). So können viele gleichartige Dokumente mit möglichst geringem Aufwand publiziert werden.

3.3.1. Vergleich von IP4W3 und ADT

Das IP4W3-System verwendet zuvor ausgezeichnete Dokumente. Das Hauptaugenmerk liegt hier auf der Suche innerhalb bestimmter Kategorien. Die vom System benötigten Dokumente werden normalerweise von einem zugehörigen Werkzeug bearbeitet. Die Annotation der Dokumente erfolgt allerdings nicht maschinell. Um den Aufwand der manuellen Auszeichnung der Dokumente zu verringern, ist es sinnvoll die Annotationen mit dem ADT-System automatisch zu erstellen.

4. Das Gesamtsystem

Dieses Kapitel beschreibt die einzelnen Phasen des ADT-Systems. Erläutert wird dabei, in welchen Schritten welche Aufgaben bearbeitet werden.

Detaillierte Beschreibungen zur Implementierung finden sich im Kapitel 8. Die hier folgenden Beschreibungen dienen erst einmal dazu, dem Leser einen Überblick über den Ablauf und die Funktionsweise des Systems zu verschaffen.

4.1. Komponenten des Systems

Die folgende Abbildung 4.1 zeigt alle Phasen bzw. Einheiten des Systems.

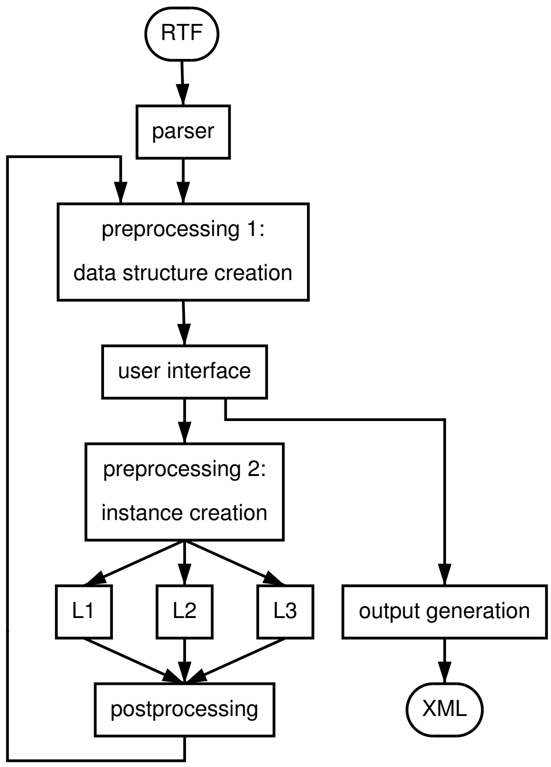

Abbildung 4.1.: Phasen und Ablauf des ADT-Systems

Um aus RTF-Dokumenten strukturierte XML-Dokumente zu generieren müssen die Daten, die sich durch die RTF-Texte ergeben, mehrfach aufgearbeitet werden, bevor die durch Anwender gegebenen Beispielklassifikationen von einem Lernalgorithmus genutzt werden können. Die Aufbereitung der Daten erfolgt dabei in den beiden »Preprocessing«-Schritten *data structure creation* und *instance creation*.

Im Diagramm ist zu erkennen, dass im System eine Schleife realisiert wurde, durch die alle Phasen vom »Preprocessing 1« bis zum »Postprocessing« beliebig wiederholt werden können. Den sich hierdurch ergebenden Vorteil und weitere Details beschreiben die folgenden Abschnitte.

4.1.1. Parser

Ein Parser ist ein Programm, das einen Text liest und ihn mit einer vorgegebenen Grammatik vergleicht. Die einzelnen Textteile werden den grammatikalischen Einheiten zugeordnet. Hierbei teilt man die Arbeit in zwei Phasen auf. Zuerst übernimmt ein so genannter Scanner (o. Tokenizer) die Erkennung der einzelnen Worte. Danach nimmt der Parser eine Gruppierung gemäß der gegebenen Grammatik vor. Der Scanner teilt zunächst den einzulesenden String in einer zuvor definierten Weise in Teilstrings (Tokens) auf. Anschließend werden die einzelnen Tokens klassifiziert.

Bei der Realisierung von ADT wurde ein *Top-Down-Parser* implementiert um die zu verarbeitenden RTF-Dateien zu analysieren. Generell lassen sich Parser noch in eine zweite Klasse unterteilen, in die so genannten *Bottom-Up-Parser* [Aho et al. 1989]. Die Namensgebung spiegelt die Richtung wider, in der die Knoten des Parse-Baums konstruiert werden.

In ADT übernimmt die Klasse `Tokenizer.java` die Aufgabe des Scanners. Die Aufteilung der Tokens erfolgt hier in die Klassen *controlword*, *documenttext*, *documentend*, *groupstart* und *groupend*. Der Parser hat nun die Möglichkeit die einzelnen Tokens gemäß der Grammatik zu verarbeiten. Um die Verarbeitung der Tokens zu vereinfachen, wurde im Rahmen der Diplomarbeit ein Scanner mit einem Lookahead von zwei Tokens implementiert. Damit ist es dem Parser möglich, bevor er das nächste Token vom Scanner anfordert, nachzuschauen, welches das nächste bzw. übernächste Element sein wird. Abbildung 4.2 zeigt den groben Ablauf des Parsings innerhalb von ADT: Zu Beginn liest das System ein RTF Dokument ein, der Scanner zerlegt die Eingabe in einzelne Tokens, die wiederum vom Parser angefordert werden. Der Parser seinerseits erzeugt für jeden Paragraphen des Dokuments Einheiten die in einem Vektor gespeichert werden. Eine detaillierte Beschreibung dieses Vorgangs wird im Kapitel 8 gegeben.

4.1.2. Preprocessing 1: Datenstrukturerzeugung

Im ersten Preprocessing[1]-Schritt werden die einzelnen Paragraphen, des durch den Parser eingelesenen Dokuments, in eine geeignete Datenstruktur überführt.

Um die Menge der zu speichernden und zu verarbeitenden Daten möglichst gering zu halten, ist es sinnvoll, zuerst einmal alle Daten zu entfernen, die keine Informationen zur Lösung des Klassifikationproblems bieten.

[1] eng. Vorverarbeitung

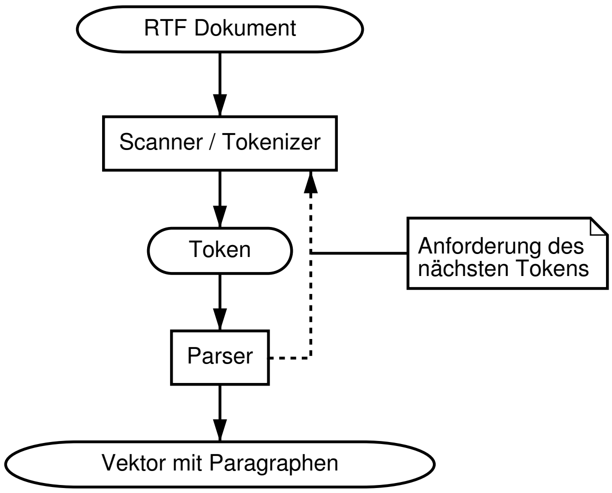

Abbildung 4.2.: Der ADT Parser

Zu diesen Daten gehört zum einen die in Abschnitt 2.2.7 beschriebene Header-Gruppe einer RTF-Datei. Das die dort gegebenen Informationen unbeachtet bleiben können, wird in Abschnitt 8.1.3 ausführlich erläutert.

Zum anderen wird im ersten Preprocessing-Schritt eine Datenbereinigung durchgeführt.

Allgemein geht es bei der Datenbereinigung um das Entfernen fehlerhafter oder irrelevanter Daten aus einer Datenmenge. Die Daten die aus einem Dokument gewonnen werden, sind zwar in der Regel nicht fehlerhaft, es finden sich allerdings eine Reihe von Daten die keine Informationen zur Lösung des Problems beitragen.

In RTF-Dokumenten werden beispielsweise meist vor Beginn eines neuen Paragraphen alle Absatzformatierungen, mit dem Befehl `\pard`, auf die Standardeinstellung zurückgestellt. Folglich müssen die Absatzformatierungen anschließend wieder neu gesetzt werden. Dies führt dazu, dass es eine große Menge an Kontrollwörtern gibt, die in allen Paragraphen vorkommen. Daher liefern sie keine Informationen mit denen bestimmte Abschnitte im Dokument beschrieben werden könnten.

Der Datenbereinigungsvorgang stellt die Menge B aller irrelevanten Kontrollwörter auf:

Sei $A := \{A_1, A_2, \ldots, A_m\}$ die Menge aller Paragraphen, $B := \{k_1, k_2, \ldots, k_n\}$ die Menge aller Kontrolwörter und $kw(A_x); x \in [1, m]$, $kw : A \mapsto \wp(B)$, dann ist

$$B = \bigcap_{i=1}^{n} kw(A_i)$$

die Menge aller Kontrollwörter, die in allen Paragraphen von A vorkommen.

Die nach der Datenbereinigung als relevant erkannten Steuerwörter werden bezüglich eines Paragraphen, den sie formatieren, gespeichert. Wichtig für die weitere Verarbeitung im zweiten Preprocessing-Schritt, der Instanzerzeugung, ist hierbei, dass die Anzahl der von einem Kontrollwort beeinflussten Zeichen eines Paragraphen, errechnet und ebenfalls gespeichert wird. Zu beachten ist hierbei, dass im RTF-Format Gruppen geschachtelt werden können. Dadurch bezieht sich ein Steuerwort einer Gruppe auch auf alle Subgruppen, sofern es nicht von einem anderen überschrieben wird.

4.1.3. User Interface

Das User Interface[2] ist eine sehr wichtige Komponente innerhalb von ADT, wodurch ein gut zu bedienendes System geboten werden soll. Wichtig ist hierbei, dass der Anwender ein Dokument möglichst einfach »durchlaufen« kann. Auch durch lange Texte soll eine schnelle Navigation geboten werden. Auf den ersten Blick scheinen diese Forderungen keinen Einfluss auf die Klassifikation der Paragraphen zu bewirken. Da der Erfolg der Klassifikation allerdings eng mit der Auswahl der Beispielklassifikationen zusammenhängt, ist es äußerst wichtig, dass der Anwender möglichst gute Beispieleklassifikationen zusammenstellt. Vielleicht noch entscheidender für eine korrekte Klassifikation ist, dass fehlerhafte Klassifikationen schnell erkannt werden.

Der Benutzerschnittstelle kann also eine entscheidende Rolle zu geordnet werden. Um so besser es hier gelingt die Intelligenz des Benutzers zu nutzen, indem ihm das Dokument und fehlerhafte Klassifikationen möglichst optimal dargestellt werden, um so besser ist wahrscheinlich auch die vom Benutzer zusammenstellte Beispielmenge. Diese wirkt sich entscheidend auf das Ergebnis der Klassifikation aus.

Die Abbildung 4.3 soll dem Leser einen ersten Eindruck der Benutzerschnittstelle des ADT-Systems geben. Wie leicht Anwender innerhalb von ADT Beispielklassifikationen zusammenstellen können, wird im Kapitel 5 erläutert.

4.1.4. Preprocessing 2: Instanzerzeugung

Im zweiten Preprocessing-Schritt werden Instanzen[3] gebildet die von Lernalgorithmen genutzt werden, um einen Klassifizierer zu erzeugen, mit dessen Hilfe noch nicht klassifizierte Paragraphen einer bestimmten Klasse zu geordnet werden können.

Um die Instanzen zu bilden sind mehrere Schritte nötig:

Zu Beginn wird die Gesamtmenge aller relevanten Steuerwörter, die innerhalb des Dokumentes benutzt werden, gebildet. D.h. alle Kontrollwörter die nicht ausschließlich in der Header-Gruppe vorhanden sind und nicht bei der Datenbereinigung als irrelevant erkannt wurden, bilden diese Menge.

Im zweiten Schritt wird für alle, während der Datenstrukturerzeugung, gefundenen Steuerwörter überprüft, ob sie in den einzelnen Paragraphen vorhanden sind oder nicht. Dementsprechend kann für jeden Paragraphen eine Instanz erzeugt werden, bei der angegeben wird, ob ein bestimmtes Kontrollwort vorhanden ist oder nicht.

[2] engl. Benutzerschnittstelle

[3] Eine Instanz ist ein unabhängiges Beispiel für das zu erlernende Konzept. Jede Instanz wird durch die Werte von Attributen charakterisiert, die unterschiedliche Aspekte der Instanz beschreiben.

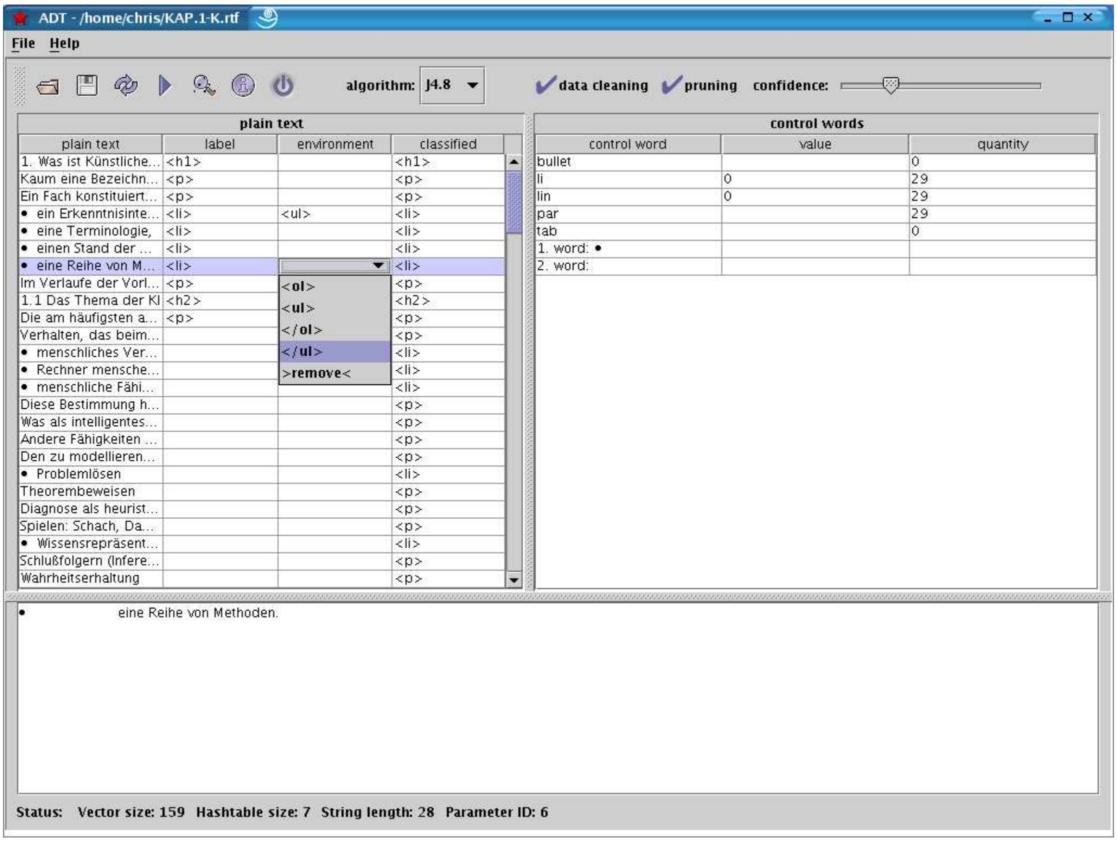

Abbildung 4.3.: Die ADT-Benutzerschnittstelle erlaubt eine einfache Bedienung und einen guten Überblick über das Dokument und Fehler die bei der automatischen Klassifikation entstanden sind.

Das Ergebnis der Klassifikation kann entscheidend verbessert werden, wenn weitere Attribute erzeugt und den Instanzen hinzugefügt werden. Dies ist notwendig, da ADT Attribut-Werte-Lernverfahren verwendet. Die daher nicht vorhandenen Relationen müssen approximiert werden.

Hierfür wird Paragraph n mit Paragraph $n+1$ verglichen. Dadurch können bei der Instanze des $n+1$ Paragraphen zusätzliche Attribute mit vier verschiedenen Attributwerten ergänzt werden. Der Vergleich der benachbarten Absätze ermöglicht es den Anfang oder das Ende einer bestimmten Struktur, wie z. B. von Aufzählungen, zu erkennen. Wie dies genau funktioniert zeigt das Beispiel 7 und Tabelle 4.2.

Eine Betrachtung der direkten Nachbarabsätze ist ausreichend um die Grenzen von Strukturen zu erkennen. D.h. Anfänge und Beendigungen von Strukturen sind dadurch gekennzeichnet, dass sich genau da, wo sie auftreten die Attributwerte des aktuell betrachteten Paragraphen gegenüber seinem direkten Vorgänger geändert haben. Der Vergleich zwischen direkt benachbarten Paragraphen ist also ausreichend.

Tabelle 4.1 zeigt eine Möglichkeit Relationen zu approximieren, die auch im Rahmen der

Diplomarbeit verwendet wurde:

Kodierung	entspricht
`ff`	siehe Punkt 1
`ft`	siehe Punkt 2
`tf`	siehe Punkt 3
`tt`	siehe Punkt 4

1. Ein Attribut ist weder im aktuell betrachteten Paragraphen, noch in seinem direkten Vorgänger vorhanden.
2. Ein Attribut ist im aktuell betrachteten Paragraphen nicht, jedoch in seinem direkten Vorgänger vorhanden.
3. Ein Attribut ist im aktuellen betrachteten Paragraphen vorhanden, jedoch nicht im Vorgängerparagraphen.
4. Ein Attribut ist im aktuell betrachteten Paragraphen und in seinem direkten Vorgänger vorhanden.

Tabelle 4.1.: Attributcodierung in ADT

Mithilfe der zusätzlichen Attribute kann die Veränderung von aufeinanderfolgenden Paragraphen festgestellt werden. Dadurch können verschiedene Dokumentstrukturen besser erkannt werden. Dies gilt besonders für den Beginn und die Beendigung von gleichen, aufeinanderfolgenden Strukturen, die eine Gruppe bilden. Dazu zählen beispielsweise Aufzählungen, wie sie im folgenden Beispiel 6 gezeigt werden.

Beispiel 6:

...Ein Fach konstituiert sich mindestens durch

- ein Erkenntnisinteresse, also ein Thema,
- eine Terminologie,
- einen Stand der Diskussion, also eine Menge von Annahmen, auf die sich ein Kreis von Kollegen geeinigt hat, und durch
- eine Reihe von Methoden.

Im Verlaufe der Vorlesung werden Sie die einzelnen Forschungsthemen, die Terminologie und die dahinter stehenden Annahmen und die wesentlichen Methoden der künstlichen Intelligenz, von jetzt ab kurz „KI“ genannt, kennenlernen...

Durch die ergänzenden Attribute, die eine Veränderung der ursprünglichen Attribute beschreiben, kann ein Klassifizierer nun auch Gruppen von Strukturen erkennen. Ohne die neuen Attribute ergibt sich folgende exemplarische Darstellung der Attribute für den Text aus Beispiel 6 (siehe Tabelle 4.2).

Paragraph	Attribute				
	`tab`	`bullet`	`li`	`lin`	`par`
1	`f`	`f`	`t`	`t`	`t`
2	`t`	`t`	`t`	`t`	`t`
3	`t`	`t`	`t`	`t`	`t`
4	`t`	`t`	`t`	`t`	`t`
5	`t`	`t`	`t`	`t`	`t`
6	`f`	`f`	`t`	`t`	`t`

Tabelle 4.2.: Attributcodierung ohne zusätzlich erzeugte Attribute

Ein Klassifizierer kann mit diesen Attributen zwischen normalem Text und Aufzählungen unterscheiden. Bei welchem Paragraphen die Aufzählung beginnt bzw. endet kann er allerdings nicht erkennen, da bezüglich der Attribute kein Unterschied zwischen dem zweiten, dritten und vierten Paragraphen besteht.

Mithilfe der ergänzenden Attribute ist es nun möglich, den Beginn und die Beendigung der Aufzählung zu identifizieren. Die Tabelle 4.3 enthält doppelt so viele Attribute wie Tabelle 4.2. Zu jedem Attribut der ersten Tabelle gibt es ein weiteres ergänzendes Attribut, welches mit dem jeweils selben Namen und der Endung »_C«, für *change* ergänzt wurde.

Das ein Attribut `tab` in einem Paragraphen beispielsweise vorhanden ist, im vorherigen Paragraphen allerdings nicht vorhanden war, wird durch `tf` kodiert.

Beispiel 7:

> Werden alle Paragraphen mit ihrem Vorgängern verglichen, unterscheidet sich der Paragraph zwei von Paragraph drei und vier dadurch, dass ihm, bezüglich seines Vorgängers, einige Attribute hinzugefügt wurden. Somit könnte ein Klassifizierer den Beginn der Aufzählungsgruppe erkennen. Das Ende der Gruppe ist durch den Wegfall einiger Attribute im ersten, der Aufzählungsgruppe folgendem, Paragraphen gekennzeichnet.

Neben den Steuerwörtern ohne Parameter gibt es in der RTF-Spezifikation auch Kontrollwörter die Zahlenwerte als Parameter verwenden. Die Instanzen sollten für diese Steuerwörter nicht ihr Vor- oder Nichtvorkommen speichern, sondern den Parameterwert des jeweiligen Steuerwortes.

Entsprechend dem oben beschriebenen Vorgehen ist es auch hier sinnvoll, Veränderungen zwischen Paragraphen, bezüglich ihrer Parameterwerte, zu analysieren. Die Instanzen können somit um ein Attribut pro Kontrollwort, mit Parameter, erweitert werden. Die ergänzenden Attribute beschreiben dann, ob ein Parameter sich vergrößert oder verkleinert hat oder vorher nicht vorhanden war. Im Kapitel 6 wird an einem Versuch gezeigt,

Para-graph	Attribute									
	tab	tab_C	bullet	bullet_C	li	li_C	lin	lin_C	par	par_C
1	f	ff	f	tf	t	ff	t	tf	t	tf
2	t	tf	t	tf	t	tt	t	tt	t	tt
3	t	tt	t	tt	t	tt	t	tt	t	tt
4	t	tt	t	tt	t	tt	t	tt	t	tt
5	t	tt	t	tt	t	tt	t	tt	t	tt
6	f	ft	f	ft	t	tt	t	tt	t	tt

Tabelle 4.3.: Attributcodierung mit zusätzlich erzeugten Attributen

dass die Kenntnis über eine Änderung von Attributen sehr wichtig ist, um beispielsweise verschachtelte Strukturen zu erkennen.

Um neben den beschriebenen Möglichkeiten noch zusätzliche Attribute durch vergleichen von Paragraphen zu erzeugen, erweitert ADT die Instanzen noch um weitere Attribute. In einigen Dokumenten kommt es vor, dass bestimmte Absätze dadurch charakterisiert werden können, dass sie stets mit dem selben Wort oder den selben Worten beginnen. Definitionen in Vorlesungsskripten oder Schulbüchern beginnen meist mit dem Wort »Definition«. Das erste Wort stellt hier also ein wichtiges Merkmal da. Um es bei der Klassifikation zu verwenden, werden die Instanzen der Paragraphen jeweils um das Attribut erweitert, das jeweils das erste Wort des jeweiligen Paragraphen als Attributwert enthält.

Sollte unter den ersten Wörtern eine Zahl enthalten sein, kann untersucht werden, ob die Ziffern der Zahl durch Punkte getrennt sind. Verschiedene Hierarchien von Überschriften, also Hauptüberschriften und Unterüberschriften können so erkannt werden.

Das beschriebene Vorgehen wird für alle Paragraphen durchgeführt. Die vom Anwender klassifizierten Paragraphen erhalten als zusätzliches Attribut den Namen ihrer zugehörigen Klasse und bilden damit eine Instanzmenge, mit deren Hilfe ein Klassifizierer gebildet wird. Alle vom Anwender nicht klassifizierten Paragraphen enthalten dieses Attribut nicht, da es sich um das Zielattribut handelt, welches vom Klassifizierungsalgorithmus bestimmt werden soll.

4.1.5. Lernalgorithmen

Die während des zweiten Preprocessing-Schrittes erzeugte Instanzmenge wird nun verwendet, um einen Klassifizierer zu erzeugen. Auf diesen Schritt wird in Kapitel 8 ausführlich eingegangen. Verwendet man zum Beispiel eine Variante des C4.5 Algorithmuses, so erstellt der Algorithmus einen Entscheidungsbaum, mit dessen Hilfe die Paragraphen ihren zugehörigen Klassen zugeordnet werden können.

Wurde der Klassifizierer erfolgreich erstellt, wird er verwendet um alle Paragraphen des Dokumentes zu klassifizieren.

Ist dieser Vorgang abgeschlossen, kann mit dem erneuten Aufbereiten der Daten, dem so genannten Postprocessing[4], fortgefahren werden.

4.1.6. Postprocessing

Im Postprocessing-Schritt wird zuerst einmal die zuvor vom Lernalgorithmus erstellte Klassifikation mit der des Anwenders verglichen. Kommt es zu Widersprüchen werden diese in der Benutzeroberfläche angezeigt. Diese Verifikation ist wichtig, da dem Anwender so fehlerhafte Klassifikationen angezeigt werden können. Die Auswahl von zusätzlichen Beispielen, die der Benutzer für eine erneute Klassifikation zusammenstellt, ist mit hoher Wahrscheinlichkeit besser für eine erfolgreiche Klassifikation geeignet, als eine Auswahl, die er ohne Hinweise auf Fehler erstellt hätte.

Die Vorgänge »manuelle Klassifikation«, »Klassifikation durch einen Algorithmus« und »Ergebnisdarstellung« werden so oft wiederholt, bis der Anwender mit dem Resultat zufrieden ist oder keine weitere Verbesserung am Ergebnis erzielt wird.

Der Benutzer erhält also die Möglichkeit, aktiv in den Lernvorgang einzugreifen und damit die Qualität der Klassifikation entscheidend zu verbessern.

4.1.7. XML Ausgabe

Um das Dokument beispielsweise im XML-Format auszugeben, werden die Klassennamen der einzelnen Paragraphen als Tags verwendet. Der Text eines Paragraphen wird dabei, wie bei XML üblich, von einem öffnenden und einem schließenden Tag umschlossen. Da die XML-Spezifikation vorsieht, dass alle geöffneten Tags auch geschlossen werden, ist bei Strukturen, die eine Gruppe bilden, wie z. B. eine ungeordnete Liste, darauf zu achten, dass sie korrekt von Tags eingeschlossen werden.

Neben XML können an dieser Stelle auch andere Ausgabeformate wie LaTeX-Code erstellt werden. Da XML allerdings sehr flexibel weiterverarbeitet werden kann, wird im Rahmen der Diplomarbeit nur dieses Ausgabeformat verwendet. Die Ergänzung anderer Formate kann als leicht realisierbare Erweiterung angesehen werden.

[4] engl. Nachverarbeitung

5. Anwendung am Beispiel

Dieses Kapitel beschreibt die Anwendung von ADT am Beispiel. Das vorgestellte Exempel wurde während der Entwicklungszeit in mehreren Durchläufen getestet. Dem Leser soll hier ein Eindruck des laufenden Programmsystem gegeben werden, so weit das in gedruckter Form möglich ist.

Die folgenden Beschreibungen erläutern die Schritte, die notwendig sind, um aus einem im RTF-Format vorliegenden Dokument, ein mit Strukturinformationen ausgezeichnetes Dokument zu erstellen. Im Überblick sind dies:

Schritt 1: Dokument einlesen

> Um ein Dokument bearbeiten zu können, muss es ins ADT-System geladen werden.

Schritt 2: Manuelle Auszeichnung von Paragraphen und Umgebungen

> Damit die Paragraphen klassifiziert werden können, müssen einige von ihnen ausgezeichnet werden.

Schritt 3: Automatische Klassifikation

> Die in Schritt 2 nicht ausgezeichnete Paragraphen werden von ADT automatisch klassifiziert.

Schritt 4: Ergebnisse speichern oder erneute Auszeichnung

> Entspricht die Auszeichnung den Wünschen des Benutzers, kann das Dokument im HTML-Format gespeichert werden. Bei fehlerhaften Annotationen können, um das Ergebnis der Annotation zu verbessern, weitere Paragraphen manuell annotiert werden (weiter ab Schritt 2).

Schritt 5: Entscheidungsbaum ansehen

> Um nachvollziehen zu können, wie bestimmte Ergebnisse zustande gekommen sind, kann der in Schritt 3 erstellte Entscheidungsbaum betrachtet und geprüft werden.

5.1. Schritt 1: Dokument einlesen

Nachdem Start von ADT präsentiert sich das System in Form eines großen Fensters. Prinzipiell ist dies in drei unterschiedliche Bereiche, auf die im weiteren Verlauf weiter eingegangen wird, unterteilt. Damit ein Dokument mit reinem visuellen Markup in ein mit Strukturinformationen ausgezeichnetes Dokument umgewandelt werden kann, muss zuerst einmal das Originaldokument geöffnet werden. Der Text muss hierfür im RTF-Format vorliegen. Da viele Dokumente, gerade in Bereichen außerhalb der Informatik, mit WYSIWYG-Editoren wie *Microsoft Word*, *Star Office* usw. erstellt werden, liegen viele Dokumente in Formaten vor, die gerade den Programmen entstammen, mit denen sie erstellt worden sind. Das RTF-Format soll den Austausch von Dokumenten zwischen verschiedenen Programmen ermöglichen, daher wird es von nahezu allen[1] gängigen Textverarbeitungsprogrammen unterstützt. Sollte der Benutzer also einen Text bearbeiten wollen, der nicht im RTF-Format vorliegt, muss er zuerst einmal mit Hilfe des Erstellungsprogramms in RTF konvertiert werden[2].

Liegt das Dokument nun im RTF-Format vor, kann der Benutzer, wie in den meisten modernen Programmen, über den Menüpunkt *Datei* oder etwas komfortabler über die Werkzeugleiste (Abbildung 5.1), einen Datei-Dialog öffnen und das entsprechende Dokument auswählen.

1	=	Dokument öffnen
2	=	Dokument speichern
3	=	Alle Auszeichnungen löschen (reset)
4	=	Klassifikationsprozess starten
5	=	J4.8-Entscheidungsbaum ansehen
6	=	Attributinformationen aus-/einschalten
7	=	ADT beenden
8	=	Algorithmenauswahl
9	=	data cleaning aus-/einschalten
10	=	Pruning aus-/einschalten
11	=	Einstellung des Confidence-Wertes

Abbildung 5.1.: Die Werkzeugliste, mit ihrer Hilfe können die meisten Funktionen gesteuert werden.

Abbildung 5.2 zeigt das mit ADT geöffnete erste Kapitel des KI-Vorlesungsskripts [Morik 1997].

Im linken oberen Fensterbereich wird der komplette Inhalt des Dokumentes dargestellt. Da ADT auf Paragraphenebene arbeitet ist das Dokument diesbezüglich aufgeteilt. Jede Zeile die hier angezeigt wird entspricht einem einzelnen Abschnitt im Dokument. Um Missverständnisse zu vermeiden, sollte hinzugefügt werden, dass sich die hier dargestellten Paragraphen aus dem RTF-Dokument ergeben. Es muss nicht immer ein logischer

[1] dem Autor ist keine Ausnahme bekannt.
[2] meist geschieht dies indem man das Dokument öffnet und unter einem anderen Dateiformat (hier RTF) abspeichert.

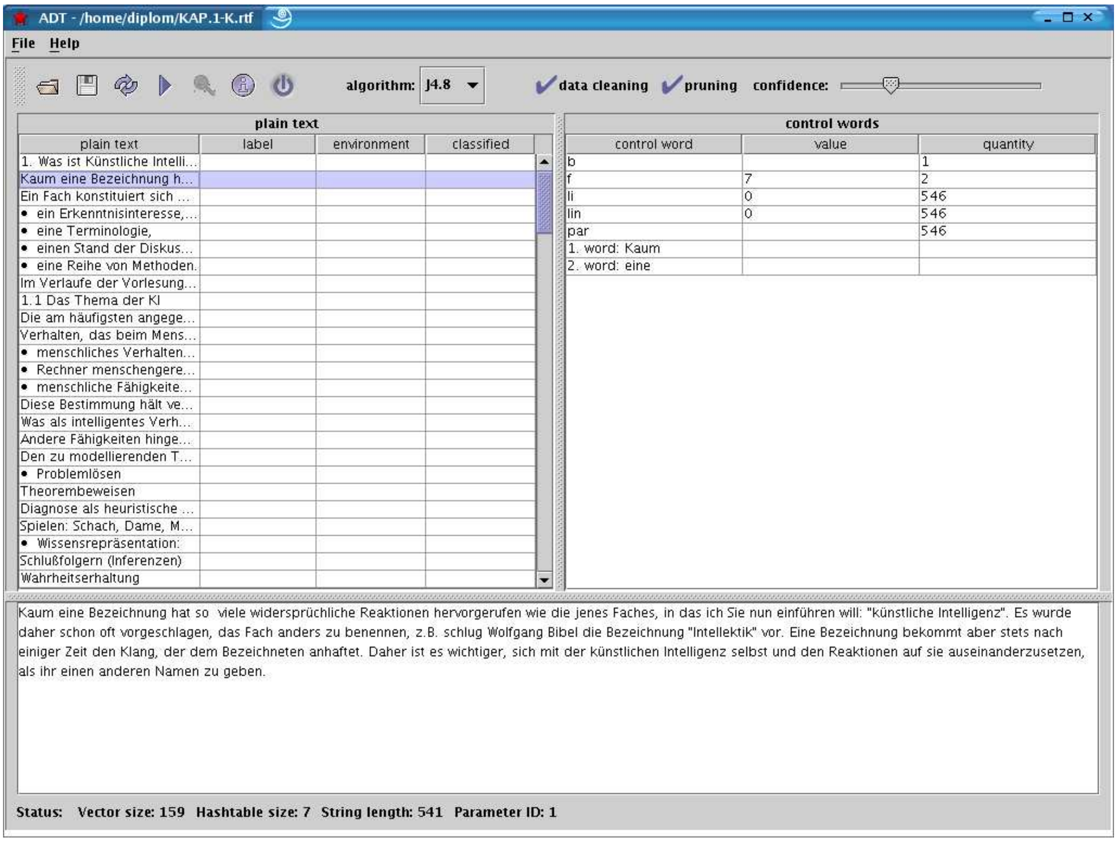

Abbildung 5.2.: ADT nachdem ein Dokument geöffnet wurde

Paragraph sein, der sich aus dem Inhalt des Textes ergibt. In RTF-Dokumenten wird das Schlüsselwort `par` oft verwendet um einen Zeilenumbruch durchzuführen. Beispielsweise tritt dies bei Aufzählungen auf.

Eine Zeile bietet natürlich viel zu wenig Platz um einen Paragraphen, der in gedruckter Form mehrere Zeilen oder vielleicht sogar große Teile einer Seite in Anspruch nimmt, komplett darzustellen. Das ist hier auch gar nicht nötig und nicht gewollt. Die Darstellung ermöglicht dem Benutzer eine schnelle Navigation durch das Dokument, langes scrollen ist hier nicht mehr nötig. Bearbeitet der Autor seinen eigenen Text, ist es für ihn meist nicht sehr schwer, anhand der ersten Wörter zu erkennen, um welchen Paragraphen es sich handelt.

Sollte dennoch ein kompletter Paragraph gefragt sein, kann dieser im unteren länglichen Fenster des Programms angezeigt werden. Der Benutzer selektiert einfach einen Paragraphen in der zuvor beschriebenen Auflistung aller Paragraphen und der vollständige Inhalt wird angezeigt.

Der dritte größere Fensterabschnitt rechts oben unterstützt den Benutzer bei der Klassifikation. Ähnlichkeiten von Paragraphen oder Paragraphenarten lassen sich hier erkennen. Für die normale Benutzung von ADT ist dieser Abschnitt eigentlich nicht von großer Bedeutung. Bei genaueren Analysen von Textabschnitten ist er aber sehr hilfreich. Weitere Details diesbezüglich werden in späteren Abschnitten gegeben.

5.2. Schritt 2: Manuelle Auszeichnung von Paragraphen und Umgebungen

Hat der Benutzer erst einmal ein Dokument geöffnet, können einzelne Paragraphen und Umgebungen ausgezeichnet werden. Die Auszeichnung selbst wird wiederum im oberen Fensterbereich vorgenommen. Wie in Abschnitt 5.1 beschrieben, werden hier zuerst einmal die Paragraphen, in der ersten Spalte, aufgeführt. In der zweiten Spalte hat man die Möglichkeit mittels Mausklick ein Auswahlmenü (engl. combobox) zu öffnen. Der Benutzer klickt also den auszuzeichnenden Paragraphen an und kann im Auswahlmenü das Label für den Absatz bestimmen. Abbildung 5.3 zeigt den Auswahlvorgang und mögliche Labels die zur Auswahl stehen.

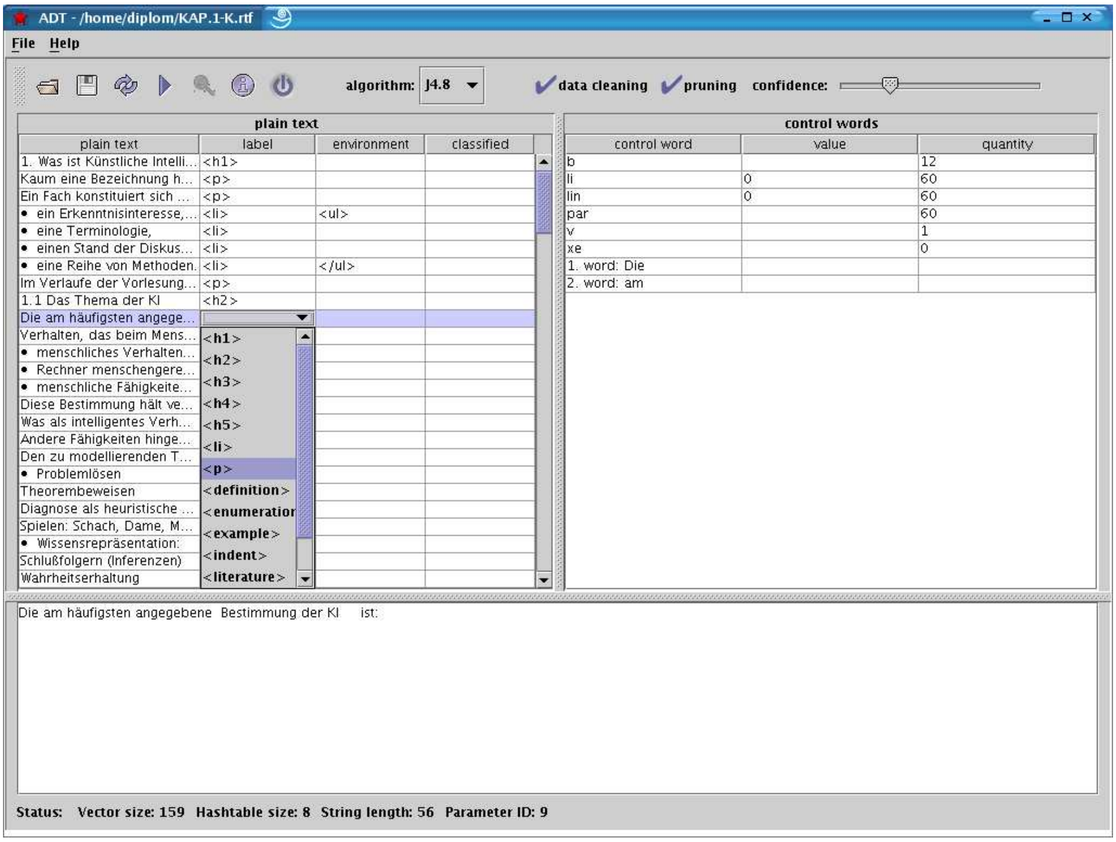

Abbildung 5.3.: Die oberen Paragraphen wurden manuell ausgezeichnet. Das geöffnete Menü zeigt eine Auswahl möglicher Labels.

Sollte eine Klasse gewünscht werden, welche unter den vorgegebenen Marken nicht vorhanden ist, besteht die Möglichkeit, über den Menüpunkt »neu« einen Dialog zu öffnen, indem beliebige Label definiert werden können.

Enthält ein Dokument verschiedene Umgebungen, wie zum Beispiel Listen mit Aufzählungspunkten, Aufzählungen mit Bezifferung oder einfache Einrückungen, wie sie von Versen oder Reimen bekannt sind, werden auch für diese Umgebungen einige Beispiele gegeben. Im Prinzip funktioniert die Auszeichnung der Umgebungen auf die selbe Art und Weise wie die Auszeichnungen der Paragraphen. In Abbildung 5.3 ist eine manuell ausgezeichnete Umgebung zu sehen. Es handelt sich hier um eine ungeordnete Liste[3]. Der Anfang der Umgebung wird mit `ul` bezeichnet, wogegen das Ende mit `\ul` gekennzeichnet wird.

Um eine möglichst gute Klassifikation durch das System zu erhalten, sollten zu einem bestimmten Paragraphentyp mehrere Auszeichnungen erfolgen. Kapitel 7 zeigt, dass vier Beispiele je Klasse ausreichend sind um ein gutes Ergebnis zu erhalten.

Zum Ende dieses Abschnittes sollte noch erwähnt werden, dass es nicht möglich ist einem Paragraphen zwei verschiedene Label zu zuteilen. Ein Paragraph kann nur genau einer Klasse angehören. Man kann aber selbstverständlich einem Paragraphen zusätzlich zu einem Label auch eine Umgebung zuordnen. Darüberhinaus können Paragraphen mit mehreren Umgebungsauszeichnungen annotiert werden. Dieser Fall tritt auf, wenn verschiedene Umgebungen ineinander geschachtelt werden.

5.3. Schritt 3: Automatische Klassifikation

Hat der Benutzer die in den vorherigen Abschnitten beschiebenen Schritte durchgeführt, kann die automatische Klassifikation gestartet werden. Der Benutzer muss hierfür nicht verschiedenen Tools benutzen oder komplizierte Einstellungen vornehmen. Die normalerweise meist nötige Vorverarbeitung (preprocessing) wird komplett von ADT erledigt. Durch einen einfachen Klick auf den Startknopf (siehe Abbildung 5.1, Knopf 4) in der Werkzeugleiste, wird der Klassifikationsprozess gestartet. Das Aufstellen eines Entscheidungsbaumes und die Klassifikation aller Paragraphen des Dokumentes dauert nur einen Augenblick. Das Ergebnis wird anschließend in der Spalte *classified* angezeigt. Sollte es zu Differenzen zwischen einer durch den Benutzer vorgenommenen Klassifikation und einer durch das System ermittelten Klassenzugehörigkeit kommen, wird diese durch eine Rotfärbung des Labels angezeigt.

Die Abbildung 5.4 zeigt die erste Klassifikation des ersten Kapitels des KI-Vorlesungsskripts [Morik 1997]. Meist kommt es vor, dass nach nur einer Klassifikation einigen Paragraphen noch falsch annotiert sind. Ein zweiter Klassifikationslauf mit weiteren Beispielen schafft allerdings meist Besserung. Um mit möglichst wenig Aufwand ein Dokument mit Strukturinformationen zu ergänzen, hat es sich bewährt, zuerst nur ein paar Beispiele zu geben, danach zu prüfen welche Paragraphen schon korrekt erkannt werden und dann ergänzende Beispiele zu geben, anstatt mit einer großen Menge von Beispielen zu starten.

[3] Die Bezeichnung `ul` stammt aus dem HTML-Bereich und steht für *unordered list*. Die einzelnen Aufzählungen werden durch einen schwarzen Punkt gekennzeichnet.

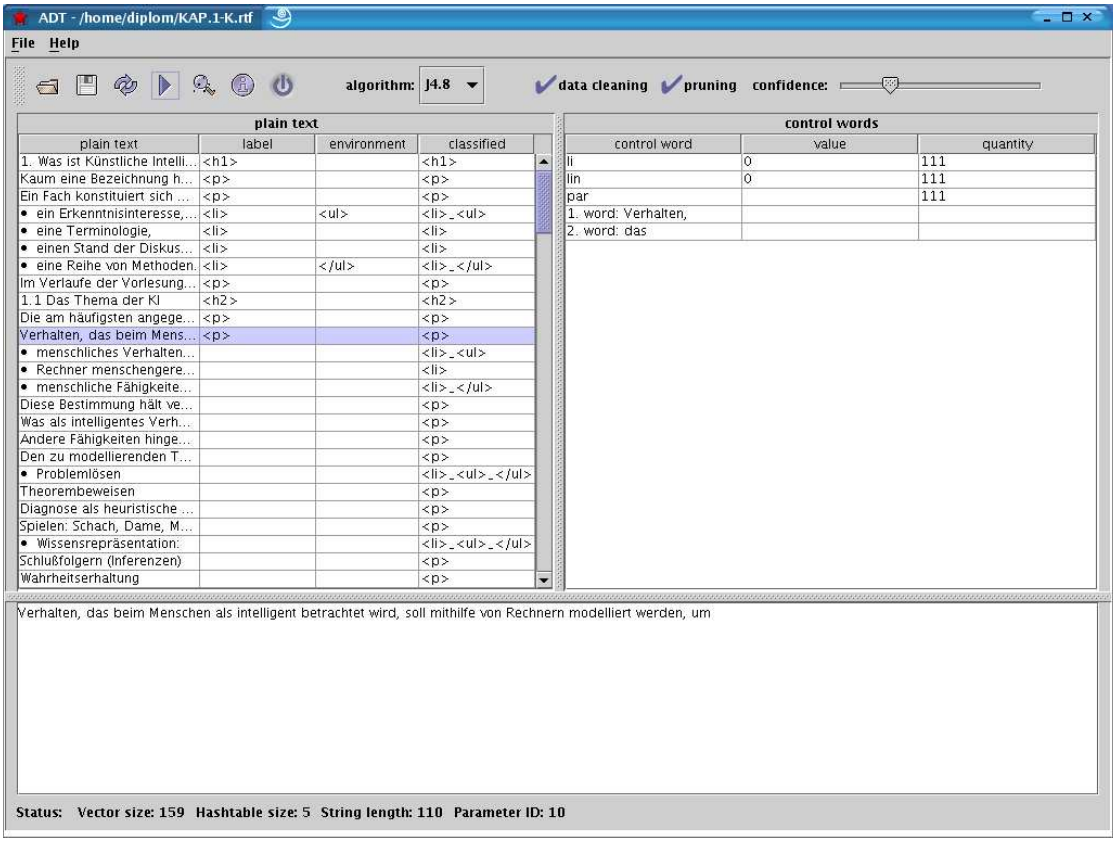

Abbildung 5.4.: Die automatische Klassifikation der Paragraphen wird in der Spalte *classified* angezeigt.

5.4. Schritt 4: Ergebnisse speichern

Wenn das Ergebnis der Klassifikation den Wünschen des Benutzers entspricht, kann das Dokument abgespeichert werden. Hierfür ist in der Werkzeugleiste (Abbildung 5.1, Punkt 2) das Diskettensymbol zu wählen. Zur Speicherung des strukturierten Dokumentes ist standardmäßig das HTML-Format vorgegeben. Abbildung 5.5 zeigt einen Auszug aus einem mit ADT erzeugten Text. Neben der Speicherung des Textes im HTML-Format stehen noch weitere Formate zur Verfügung. Die meisten dienten bei der Entwicklung von ADT Testzwecken, so lassen sich beispielsweise alle im Dokument vorkommenden Attribute speichern, weiterhin besteht die Möglichkeit, Formate für das Modellierungssystem MOBAL[4] zu erzeugen. Da diese Formate in Hinblick auf diese Diplomarbeit nicht weiter von Bedeutung sind, werden sie hier nicht näher erklärt.

[4] Aufgabe des Systems ist, den manuellen Prozess der Wissensakquisition durch Kontrolle der Wissensbasis auf Inkonsistenz zu unterstützen und so dem Benutzer eine Prüfung der Wissensbasis zu ermöglichen. Weiterhin werden in MOBAL, durch die verschiedenen eingebundenen Lernverfahren, dem Anwender nicht bekannte Zusammenhänge aufgezeigt.
`ftp://ftp.gmd.de/gmd/mlt/Mobal/Mobal4.2b09.tar.gz`

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0//EN//">
<html>
<h1>1. Was ist Künstliche Intelligenz</h1>
<p>Kaum eine Bezeichnung hat so  viele widersprüchliche Reaktionen
hervorgerufen wie die jenes Faches, in das ich Sie nun einführen will:
"künstliche Intelligenz". Es wurde daher schon oft vorgeschlagen, das Fach
anders zu benennen, z.B. schlug Wolfgang Bibel die Bezeichnung "Intellektik"
vor. Eine Bezeichnung bekommt aber stets nach einiger Zeit den Klang, der dem
Bezeichneten anhaftet. Daher ist es wichtiger, sich mit der künstlichen
Intelligenz selbst und den Reaktionen auf sie auseinanderzusetzen, als ihr
einen anderen Namen zu geben.</p>
<p>Ein Fach konstituiert sich mindestens durch</p>
<ul>
<li>ein Erkenntnisinteresse, also ein Thema,</li>
<li>eine Terminologie,</li>
<li>einen Stand der Diskussion, also eine Menge von Annahmen, auf die sich
ein Kreis von  Kollegen geeinigt hat, und durch</li>
<li>eine Reihe von Methoden.</li>
</ul>
<p>Im Verlaufe der Vorlesung werden Sie die einzelnen Forschungsthemen, die
Terminologie und die dahinter stehenden Annahmen und die wesentlichen Methoden
der künstlichen Intelligenz, von jetzt ab kurz "KI" genannt,
kennenlernen. Dabei werde ich stets die Entwicklung der fachlichen Diskussion
zum Thema skizzieren. Vorab aber soll das globale Thema der KI umrissen werden
(1.1). Dann wird die generelle Herangehensweise der KI diskutiert, indem drei
verschiedene Ansätze genannt werden (1.2). Schließlich werde ich auf zwei
wesentliche Kritiken an der KI eingehen: die Kritik an Newells "Physical
Symbol System Hypothesis"(1.3.1) und Searles Beispiel des chinesischen Zimmers
(1.3.2).</p>
```

Abbildung 5.5.: Ein mit Hilfe von ADT annotiertes Dokument

Ein Format, welches allerdings erwähnt werden sollte, ist das *ARFF-Format*. ARFF ist das in der Lernumgebung WEKA verwendete Dateiformat. Interessant ist die Möglichkeit, manuell ausgezeichneten Paragraphen im ARFF-Format zu speichern in so fern, dass die Daten mit Hilfe der WEKA-Umgebung verarbeitet werden können. Dem Benutzer steht so die Möglichkeit offen, andere Einstellungen an den in ADT verfügbaren Algorithmen vorzunehmen.

ftp://ftp.gmd.de/gmd/mlt/Mobal/UserGuide41b9.ps.gz

5.5. Schritt 5: Entscheidungsbaum ansehen

Manchmal werden Paragraphen falschen Klassen zugeordnet. Zeichnet der Benutzer dann weitere Absätze aus, um mehr Beispiele für eine bestimmte Kategorie zu geben, kann die fehlerhafte Klassifikation meist korrigiert werden. In manchen Umständen führen auch mehr Beispiele zu keinem besseren Ergebnis. In diesem Fall ist es oft aufschlussreich etwas darüber zu erfahren, warum bestimmte Ergebnisse erzielt wurden.

In Abschnitt 6.1 wird erklärt, wie die einzelnen Paragraphen mittels eines Entscheidungsbaums klassifiziert werden. Mit dem Entscheidungsbaum lässt sich also ermitteln, welcher Klasse ein Paragraph zugeordnet werden muss. Um nun zu erfahren wie bestimmte Ergebnisse zustande gekommen sind, kann es sinnvoll sein den Entscheidungsbaum zu analysieren. Der Benutzer öffnet hierfür über die Werkzeugliste (Abbildung 5.1, Button 5) den in Abbildung 5.6 gezeigten Dialog.

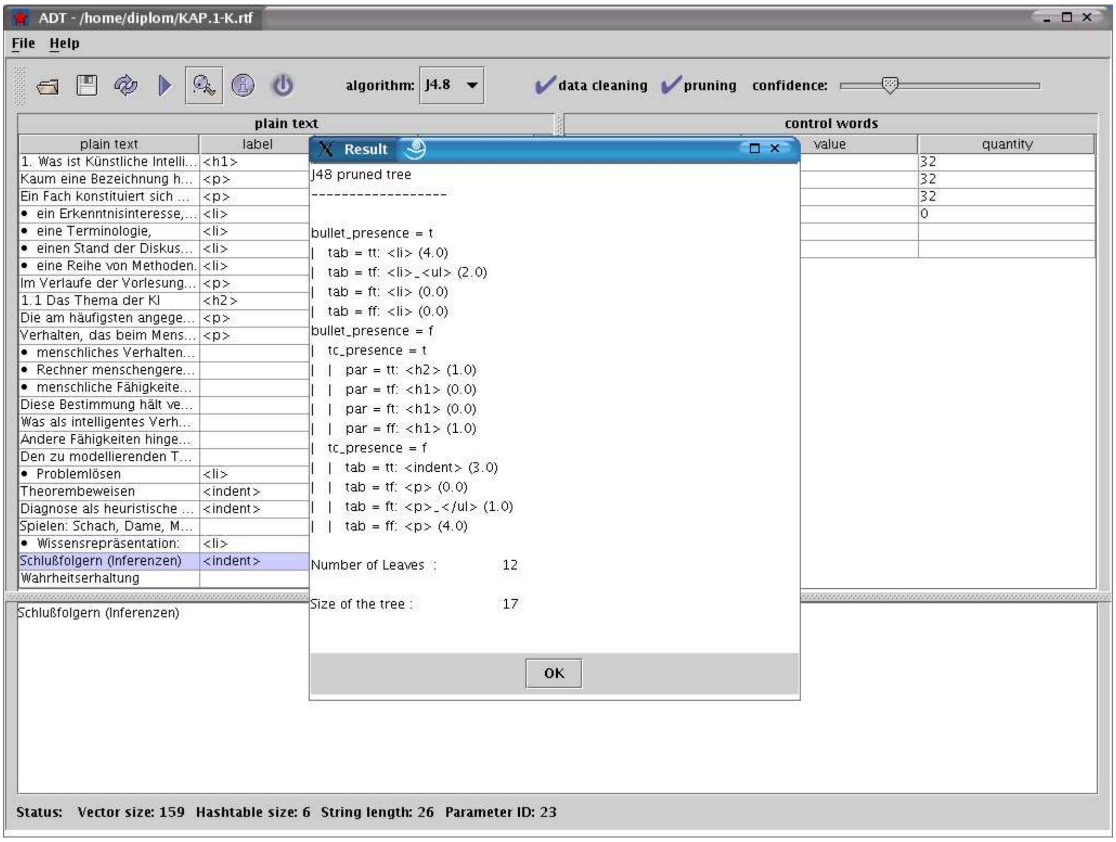

Abbildung 5.6.: Der mit ADT, unter Verwendung des J4.8 Algorithmuses, erstellte Entscheidungsbaum

6. Lernalgorithmen und Versuche

Dieses Kapitel besteht aus zwei Teilen. Der erste Teil liefert eine kurze Einführung in das Gebiet der Klassifikationsalgorithmen. Betrachtet werden hierbei die weitverbreiteten Algorithmen ID3 und C4.5. Der zweite Teil dieses Kapitels beschreibt verschiedene Versuche, die gemacht wurden, um ADT im Zusammenhang mit *geschachtelten* - und *ähnlichen Strukturen* zu testen.

6.1. Lernalgorithmen

Im Bereich des Maschinellen Lernens gibt es verschiedene Techniken um Daten zu klassifizieren. Zu den bekanntesten Klassifikationsverfahren zählen die von Ross Quinlan entwickelten Algorithmen ID3 und C4.5. Um die Arbeitsweise dieser Algorithmen zu verstehen, wird im Folgenden die Funktionsweise von ID3 beschrieben. Anschließend wird der Unterschied von ID3 und C4.5 angesprochen.

6.1.1. ID3

Der ID3 Algorithmus [Quinlan 1983] ist ein klassisches induktives Lernverfahren. Es konstruiert nach dem *top-down-Prinzip* einen *Entscheidungsbaum*, indem aus Beispielen mit Hilfe von statistischer Merkmalsselektion gelernt wird.

Ein Entscheidungsbaum hat Kanten, an denen Attributwerte stehen, Konten sind Verzweigungspunkte und Blätter stellen Klassifikationen dar. Um zu entscheiden, ob ein neues Beispiel zu einer Klasse gehört, wird den Kanten gefolgt, deren Beschriftung einem Attributwert des Beispiels entspricht, bis ein Blatt erreicht ist [Morik 1995]. Eine der wesentlichen Eigenschaften der Klassifikation ist dabei die Diskretisierung. D.h. die einzelnen Klassen sind scharf von einander getrennt, so dass ein Fall, bestehend aus einzelnen Werten, entweder zu einer Klasse gehört oder nicht.

Um einen Entscheidungsbaum aufzubauen, muss der Algorithmus für jeden Konten berechnen, welches Attribut ihm zugewiesen wird. Dies ist das zentrale Auswahlproblem von ID3. Es muss entschieden werden, welches Attribut den größten Informationsgewinn bringt, um die Trainingsdaten bestmöglich zu klassifizieren. Ist das Problem gelöst, wird das Attribut ausgewählt und für jeden möglichen Wert des Attributs wird eine Kante mit Kindknoten erzeugt. Das ausgewählte Attribut kann nun aus der Liste aller Attribute entfernt werden. Die Kindknoten werden weiter nach dem gleichen Schema rekursiv unterteilt, bis in den Knoten nur noch Objekte einer Klasse stehen. Diese Knoten bilden die Blätter des Baumes, welche die Klassenkennzeichnung des entsprechenden Objektes enthalten.

Der Informationsgehalt eines Attributs stützt sich auf die, aus der Informationstheorie stammenden Größe, *Entropie*. Sie berechnet sich nach [MORIK 1995] mit der Formel:

$$-\sum_{m=1}^{n} p_m \log_2 p_m$$

wobei n für die Anzahl der Attributwerte steht und p_m die Wahrscheinlichkeit, des Vorkommens des Zielattributs, mit dem Index m, in der Trainingsdatenmenge, angibt.

Der Informationsgewinn eines Attributs ergibt sich durch den Vergleich[1] der Entropie des Attributs mit der Entropie der Trainingsmenge. So ist es möglich das Attribut mit dem größten Informationsgehalt zu bestimmen und auszuwählen.

Der Aufbau eines optimalen Entscheidungsbaumes ist nach [HYAFIL und RIVEST 1976] NP-vollständig, d.h. das Problem kann nicht mit Hilfe eines deterministischen Algorithmuses in polynomineller Laufzeit berechnet werden. Der ID3 Algorithmus benutzt einen *nonbacktracking greedy Algorithmus*, was bedeutet, dass das zu wählende Attribut immer auf der Basis des momentanen Wissenstands bestimmt wird. Auch wenn sich später vielleicht herausstellen würde, dass die Wahl eines anderen Attributes günstiger gewesen wäre.

Zur Illustration des Algorithmuses wird als nächstes, an einem kleinen Beispiel, ein Entscheidungsbaum hergeleitet [MITCHELL 1997]. Die Tabelle 6.1 beschreibt, ob es an bestimmten Samstagmorgenden möglich war, Tennis zu spielen oder nicht.

Um die Entropie der Beispielmenge zu berechnen, betrachtet man die Spalte »Tennis spielen?« (Zielattribut).

$\frac{9}{14} \log_2 \frac{9}{14} = -0,4098$ für positive Beispiele
$\frac{5}{14} \log_2 \frac{5}{14} = -0,5305$ für negative Beispiele
Entropie der Beispielmenge $= -(-0,4098 - 0,5305) = \underline{0,9403}$

Berechnung der Entropie für Aussicht:

sonnig:
$\frac{2}{5} \log_2 \frac{2}{5} = -0,5288$ für positive Beispiele
$\frac{3}{5} \log_2 \frac{3}{5} = -0,4422$ für negative Beispiele
Summe mit umgekehrtem Vorzeichen: $0,971$

[1] d.h. man bildet die Differenz zwischen beiden Entropien. Daher bedeutet ein niedriger Entropiewert einen hohen Informationsgehalt.

Nummer	Aussicht	Temperatur	Luftfeuchtigkeit	Wind	Tennis spielen?
1	sonnig	heiß	hoch	nein	negativ
2	sonnig	heiß	hoch	ja	negativ
3	bewölkt	heiß	hoch	nein	positiv
4	Regen	mild	hoch	nein	positiv
5	Regen	kalt	normal	nein	positiv
6	Regen	kalt	normal	ja	negativ
7	bewölkt	kalt	normal	ja	positiv
8	sonnig	mild	hoch	nein	negativ
9	sonnig	kalt	normal	nein	positiv
10	Regen	mild	normal	nein	positiv
11	sonnig	mild	normal	ja	positiv
12	bewölkt	mild	hoch	ja	positiv
13	bewölkt	heiß	normal	nein	positiv
14	Regen	mild	hoch	ja	negativ

Tabelle 6.1.: War Tennisspielen samstagmorgens möglich?

bewölkt:

$\frac{4}{4}\log_2\frac{4}{4} = 0$ für die vier positiven Beispiele

Regen:

$\frac{3}{5}\log_2\frac{3}{5} = -0,5288$ für positive Beispiele

$\frac{2}{5}\log_2\frac{2}{5} = -0,4422$ für negative Beispiele

Summe mit umgekehrtem Vorzeichen: $0,971$

Entropie für Aussicht: $\frac{9}{14}\cdot 0,971 + \frac{9}{14}\cdot 0 + \frac{9}{14}\cdot 0,971 = \underline{0,6935}$

Der Informationsgewinn durch dieses Attribut ergibt sich aus dem Vergleich mit der Entropie der Beispielmenge:

$0,9403 - 0,6935 = \underline{\underline{0,247}}$

Die Informationsgewinne der anderen drei Attribute berechnen sich analog:

$$\begin{array}{lcl} \text{Informationsgewinn (Temperatur)} & = & 0{,}029 \\ \text{Informationsgewinn (Luftfeuchtigkeit)} & = & 0{,}151 \\ \text{Informationsgewinn (Wind)} & = & 0{,}048 \end{array}$$

Die Berechnungen des Informationsgewinns der einzelnen Attribute zeigen, dass das Attribut *Aussicht* am Besten geeignet ist um die Daten zu ordnen. Wenn *Aussicht* ausgewählt wurde, werden drei Kanten angelegt und mit *sonnig*, *bewölkt* und *Regen* beschriftet. Der unter *bewölkt* gebildete Knoten enthält nur positiv klassifizierte Beispiele und wird damit zum Blatt (siehe Abbildung 6.1).

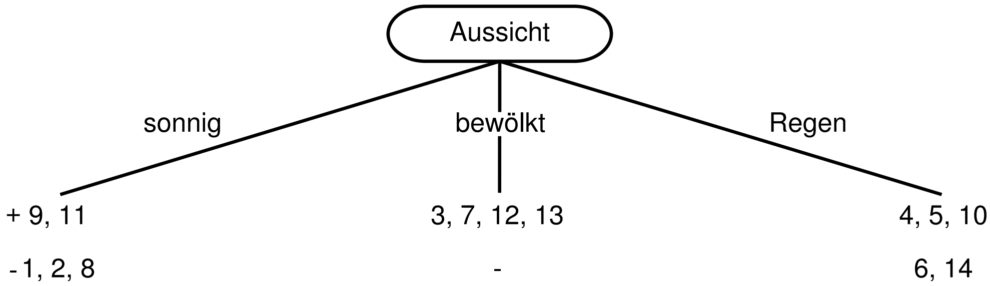

Abbildung 6.1.: halbfertiger Entscheidungsbaum

Der unter *sonnig* und *Regen* gebildete Knoten muss genau wie der oberste behandelt werden. Am Ende ergibt sich der in Abbildung 6.2 abgebildete Entscheidungsbaum.

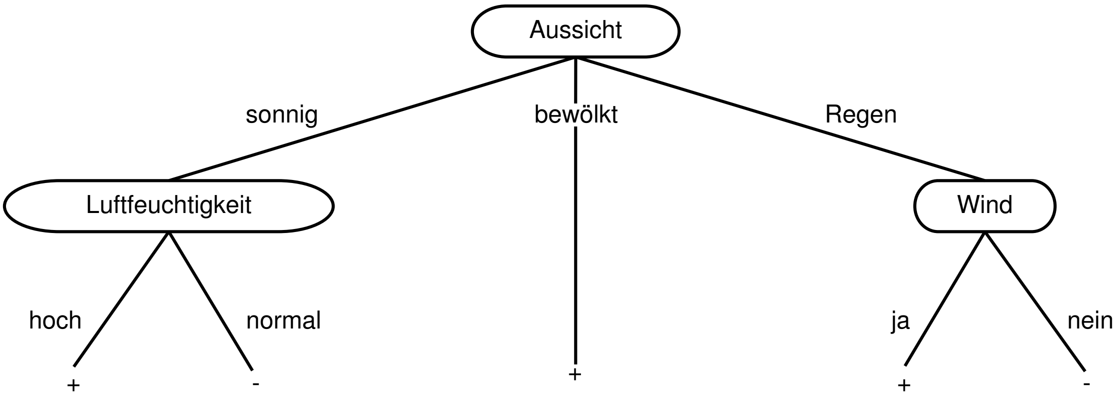

Abbildung 6.2.: fertiger Entscheidungsbaum

6.1.2. C4.5 und J4.8

Ross Quinlan forschte über viele Jahre an der Universität von Sydney um Ansätze zur Entscheidungsbauminduktion zu finden. Diverse Verbesserungen am, in Abschnitt 6.1.1 beschriebenen, ID3 Algorithmus führten zu einem einflussreichen und weit verbreiteten System für die Entscheidungsbauminduktion, C4.5. Bei den Verbesserungen geht es unter anderem um Methoden für die Verarbeitung numerischer Attribute, fehlender Werte, verrauschter Daten sowie das Erzeugen von Regeln aus Bäumen. Da genauere Details hier nicht weiter von Wichtigkeit sind, verweise ich, für interessierte Leser, auf eine gute Einführung von Ross Quinlan in [Quinlan 1993].

ADT verwendet den im WEKA-Paket enthaltenden J4.8-Algorithmus. J4.8 ist eine jüngere und leicht verbesserte Version von C4.5 Revision 8, der letzten veröffentlichten Version dieser Algorithmenfamilie, ehe C5.0, eine kommerzielle Implementierung, freigegeben wurde [Witten und Frank 2000]. J4.8 ist in JAVA implementiert, was auch das »J« in der Bezeichnung erklärt. Einen Vorteil, den J4.8 bietet, ist die Plattformunabhängigkeit. Es treten jedoch Performanzeinbußen gegenüber der originalen C Version auf.

6.2. Versuche

Dieser Abschnitt beschreibt Versuche, die mit A_D^T durchgeführt wurden. Es handelt sich hierbei um Experimente zu besonderen Schwierigkeiten der Klassifikation von Strukturen. Zum einen werden verschachtelten Strukturen, wie sie oft in Aufzählungslisten vorkommen, betrachtet. Weiterhin werden ähnliche Strukturen, die bezüglich ihrer Formatierung keine Unterschiede aufweisen, untersucht. Die Evaluierung des Systems bezüglich der Anzahl der zur Klassifikation benötigten Beispiele, sowie die Bewertung der Laufzeit, wird im folgenden Kapitel 7 behandelt.

6.2.1. Verschachtelte Strukturen

In Dokumenten kommen des Öfteren Strukturen vor, die ineinander geschachtelt sind. Meist handelt es sich dabei um Listen in denen verschiedene Punkte aufgeführt sind.

- **Lehrveranstaltung im 1. Semester**
  - Chemie
    - Vorlesung: organische Chemie
      - Klausur
    - Vorlesung: unorganische Chemie
      - Klausur
    - Praktikum
  - Physik
    - Vorlesung: Physik für Mediziner und Pharmazeuten
    - Praktikum
- **Lehrveranstaltung im 2. Semester**
  - Psychologie
    - Vorlesung: Medizinische Psychologie
      - Klausur
  - Anatomie
    - Präpkurs
    - Seminar

Abbildung 6.3.: Verschachtelte Strukturen kommen oft in Listen vor. Die aufgezählten Punkte werden hier meist durch Unterpunkte weiter differenziert.

Werden diese weiter unterteilt, geschieht dies im Allgemeinen durch eine zusätzliche

Struktur, die gegenüber der vorherigen Struktur eingerückt ist. Ein Beispiel für solch eine Liste zeigt Abbildung 6.3. Um gewisse Punkte zu differenzieren, werden neben Auflistungen oft auch Dokumentabschnitte geschachtelt.

Es ergibt sich also die Notwendigkeit, dass verschachtelte Strukturen von ADT korrekt behandelt werden.

Um zu untersuchen, welche verschachtelten Strukturen sich leicht erkennen lassen und bei welchen es zu Probleme kommt, wurde eine Reihe von Versuchen durchgeführt:

Trainingsmenge 1: Die erste getestete Struktur besteht aus einer einfachen, einstufigen Aufzählung. Abbildung 6.4 zeigt ihren Aufbau und die zum Lernen der Strukturen verwendeten Auszeichnungen.

Text Text Text Text	<p>
• Text Text	_
• Text Text	
• Text Text	
• Text Text	
• Text Text	_
Text Text Text Text	<p>

Abbildung 6.4.: Trainingsmenge 1

Für welche Strukturen diese einfache Trainingsmenge ausreichend ist und wo es zu Problemen kommt, zeigen die Testmengen in Abbildung 6.5 und 6.6.

Da die Testmenge 1 genauso aufgebaut ist wie die Trainingsmenge, kommt es hier zu keinen Problemen. Alle Paragraphen werden korrekt annotiert. Die schwierigere Testmenge 2 hingegen wird an vier Stellen falsch ausgezeichnet. Das Problem besteht hier darin, dass das Dokument mehrere Ebenen aufweist.

Text Text Text Text	<p>	✔
• Text Text	_	✔
• Text Text		✔
• Text Text		✔
• Text Text		✔
• Text Text	_	✔
Text Text Text Text	<p>	✔

Abbildung 6.5.: Testmenge 1

Der Beginn und das Ende der Aufzählung konnte korrekt erkannt werden. Die inneren Unterteilungen hingegen nicht. Dies liegt daran, dass es in der Trainingsmenge nur Beispiele für Übergänge vom normalen Text zur Aufzählung und von der Aufzählung in normalen Text gibt. Für Übergänge in weitere Ebenen, innerhalb einer Aufzählung, gibt es keine Beispiele.

Abbildung 6.7 zeigt den zur Trainingsmenge 1 gehörenden Entscheidungsbaum. Die Übergänge werden dadurch erkannt, dass das Kontrollwort `\hich`[2] zum betrachteten Paragraphen hinzu kommt, bzw. entfällt. Die Übergänge innerhalb der Aufzählung der zweiten Trainingsmenge weisen diese Veränderung nicht auf und können folglich mit diesem Entscheidungsbaum nicht korrekt klassifiziert werden.

Text Text Text Text	<p>	✔
• Text Text	_	✔
• Text Text		✔
• Text Text		✔
• Text Text		✘
• Text Text		✔
• Text Text		✘
• Text Text		✔
• Text Text		✔
• Text Text		✘
• Text Text		✔
• Text Text		✘
• Text Text		✔
• Text Text		✔
• Text Text	_	✔
Text Text Text Text	<p>	✔

Abbildung 6.6.: Testmenge 2

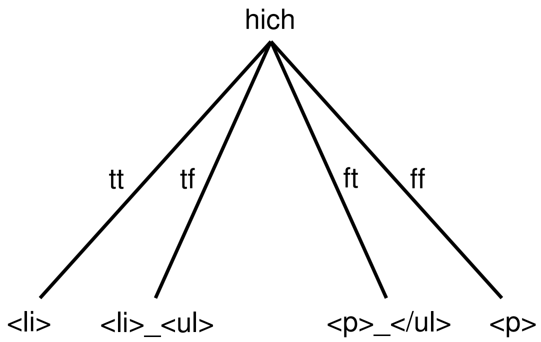

Abbildung 6.7.: Der mit der Trainingsmenge 1 gelernte Entscheidungsbaum

[2] Für Zeichen im high-ANSI (0x80 -0xFF) Bereich.

Trainingsmenge 2: Als zweite Trainingsmenge wurde die letzte Testmenge 2 verwendet. Abbildung 6.8 zeigt, dass diesmal alle Paragraphen, mit Ausnahme des Übergangs von der zweiten in die erste Aufzählungsebene, annotiert wurden.

Text Text Text Text	<p>
• Text Text	_
• Text Text	
• Text Text	
• Text Text	_
• Text Text	
• Text Text	
• Text Text	_
• Text Text	
• Text Text	_
• Text Text	
• Text Text	
• Text Text	
• Text Text	
• Text Text	_
Text Text Text Text	<p>

Abbildung 6.8.: Trainingsmenge 2

Versuche mit der Testmenge 3 aus Abbildung 6.9 zeigten, dass der Übergang von der ersten Aufzählungsebene in die zweite korrekt erkannt wurde.

Text Text Text Text	<p>	✔
• Text Text	_	✔
• Text Text		✔
• Text Text		✔
• Text Text	_	✔
• Text Text		✔
• Text Text		✔
• Text Text		✘
• Text Text		✔
• Text Text		✔
• Text Text	_	✔
Text Text Text Text	<p>	✔

Abbildung 6.9.: Testmenge 3

Der Übergang von der zweiten zurück in die erste Ebene wurde hingegen nicht erkannt.

Der Versuch mit einer weiteren Testmenge 4 ergab, dass auch hier der Abstieg von einer hohen in eine niedrigere Ebene nicht richtig annotiert wird.

Text Text Text Text	<p>	✔
• Text Text	_	✔
• Text Text		✔
• Text Text		✔
• Text Text	_	✔
• Text Text		✔
• Text Text	_	✔
• Text Text		✘
• Text Text		✔
• Text Text		✔
• Text Text	_	✔
Text Text Text Text	<p>	✔

Abbildung 6.10.: Testmenge 4

Trainingsmenge 3: Um überprüfen zu können, ob die Fehler aus den zwei vorherigen Testmengen nicht mehr auftreten, wenn ein innerer Übergang von einer höheren Ebene auf eine niedrige Ebene ausgezeichnet wird, wurde ein weiterer Versuch durchgeführt. Hierbei wurde mit Hilfe der Trainingsmenge 3, aus Abbildung 6.11, ein Entscheidungsbaum gelernt, mit dem die Testmengen 3 und 4 noch einmal klassifiziert wurden.

Text Text Text Text	<p>
• Text Text	_
• Text Text	
• Text Text	
• Text Text	_
• Text Text	
• Text Text	
• Text Text	_
• Text Text	
• Text Text	_
• Text Text	
• Text Text	_
• Text Text	
• Text Text	
• Text Text	_
Text Text Text Text	<p>

Abbildung 6.11.: Trainingsmenge 3

Wie in Abbildung 6.12 zu sehen, konnte die Testmenge 3 bzw. 5 nun ohne Fehler verarbeitet werden.

Text Text Text Text	<p>	✔
• Text Text	_	✔
• Text Text		✔
• Text Text		✔
• Text Text	_	✔
• Text Text		✔
• Text Text		✔
• Text Text	_	✔
• Text Text		✔
• Text Text		✔
• Text Text	_	✔
Text Text Text Text	<p>	✔

Abbildung 6.12.: Testmenge 5

Die Testmenge 6 (Abbildung 6.13) hingegen kann allerdings noch immer nicht korrekt klassifiziert werden. ADT erkennt zwar, dass die dritte Ebene geschlossen wird, das Ende der zweiten Ebene wird jedoch nicht ausgezeichnet. Das Problem besteht darin, dass innerhalb von ADT nur auswertet wird, ob die Einrücktiefe der Paragraphen kleiner oder größer geworden ist oder ob sie sich nicht geändert hat. Werden nach einem Paragraphen mehrere Ebenen geschlossen, verringert sich die Einrücktiefe der Absätze. Um welchen Wert sich die Tiefe verändert, bleibt unbeachtet, so dass für ADT das Schließen von einer oder mehreren Ebenen identisch ist.

Um diesen Fehler zu beheben wurde ADT erweitert. Es wird nun für jeden Paragraphen errechnet, in welcher Ebene er sich befindet. So kann für alle Absätze die Veränderung, bezüglich der Einrücktiefe ihrer Vorgänger, bestimmt werden. Genaue Details über die Erweiterungen finden sich im Anhang (siehe Abschnitt 9.3).

Das beschriebene Problem tritt eigentlich nur auf, wenn mehrere Ebenen geschlossen werden, da ein Paragraph nie mehr als eine Ebene öffnet[3].

Text Text Text Text	<p>	✔
• Text Text	_	✔
• Text Text		✔
• Text Text		✔
• Text Text	_	✔
• Text Text		✔
• Text Text	_	✔
• Text Text	_	✘
• Text Text		✔
• Text Text		✔
• Text Text	_	✔
Text Text Text Text	<p>	✔

Abbildung 6.13.: Testmenge 6

Trainingsmenge 4: Um noch einmal genauer feststellen zu können, welche Eigenschaften der Paragraphen vom Klassifikationsalgorithmus genutzt werden, um die Absätze zu klassifizieren, wurde die in Abbildung 6.14 dargestellte Trainingsmenge bearbeitet und der zugehörige Entscheidungsbaum aufgezeichnet.

Wie in Abbildung 6.15 zu sehen, ist von der Größenänderung des Kontrollwortes `\lin` abhängig, ob eine Verschachtelungsebene beginnt, endet oder ob es sich um einen normalen Aufzählungspunkt handelt. Wurde der Parameterwert von `\lin` erhöht (durch d dargestellt), beginnt eine Ebene. Wird er verkleinert (durch i dargestellt) endet eine Ebene. Bleibt der Wert gleich (uc) handelt es sich um einen normalen Aufzählungspunkt. Enthält ein Paragraph, sowie sein Vorgänger, das Schlüsselwort `\hich` nicht, so handelt es sich um einen normalen Absatz. Enthält jedoch der Vorgänger das Schlüsselwort,

[3]Das Öffnen mehrerer Ebenen an einer Stelle ist aus struktureller Sicht sinnlos und kommt somit in der Praxis eigentlich nicht vor. Daher wird dieser Fall hier auch nicht weiter betrachtet.

so handelt es sich um den ersten Paragraphen, der einer Aufzählungsumgebung folgt. Warum das Ende der Aufzählung erst hier erkannt wird und wie die Annotation richtig zum Vorgängerparagraphen verschoben wird, erläutert Kapitel 8.

Text Text Text Text	<p>
• Text Text	_
• Text Text	
• Text Text	
• Text Text	_
• Text Text	
• Text Text	
• Text Text	_
• Text Text	
• Text Text	
• Text Text	_
Text Text Text Text	<p>

Abbildung 6.14.: Trainingsmenge 4

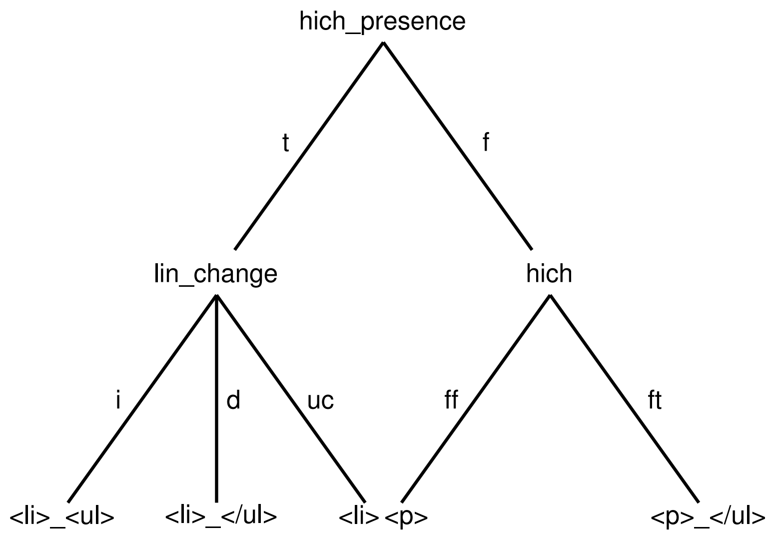

Abbildung 6.15.: Mit der Trainingsmenge 4 gelernter Entscheidungsbaum

Trainingsmenge 5: In den meisten Dokumenten kommen maximal drei bis vier ineinandergeschachtelte Strukturen vor. Eine Obergrenze der Verschachtelung wird durch das Programm gesetzt, mit dem ein Dokument erstellt wird. Da der größte Teil der Anwender, die ein Dokument erstellen, *Microsoft Word* verwenden und die maximal mögliche Tiefe der Schachtelung hier zehn Ebenen beträgt, sollten von ADT auch mindestens diese zehn Ebenen erkannt werden.

Um zu überprüfen, ob die Aufgabe - verschachtelte Strukturen korrekt erkennen und klassifizieren - von ADT zufriedenstellend bewältigt wird, wurde ein Versuchsdokument mit zehn Ebenen erstellt. Die Abbildung 6.16 zeigt eine Trainingsmenge mit drei Ebenen, die während des Versuches manuell ausgezeichnet wurde.

text text
- increase
  - top
- decrease

text text

Abbildung 6.16.: Trainingsmenge 5

text text
- increase
  - increase
    - increase
      - increase
        - increase
          - increase
            - increase
              - increase
                - top
              - decrease
            - decrease
          - decrease
        - decrease
      - decrease
    - decrease
  - decrease
- decrease

text text

Abbildung 6.17.: Testmenge 7

Die anschließende Klassifikation der Testmenge 7 (Abbildung 6.17) zeigte, dass alle zehn Ebenen des zweiten Abschnittes korrekt klassifiziert wurden.

Die maximal mit *Microsoft Word* erstellbaren verschachtelten Strukturen werden also erkannt. Im beschriebenen Versuch wurde zwar eine maximale Verschachtelungstiefe von zehn Ebenen getestet. Die Klassifikation weiterer Ebenen ist allerdings möglich, solange die Ebenen sich durch die Einrückungstiefe bezüglich des linken Seitenrandes unterscheiden. Das Kontrollwort `\linN` definiert hier die linke Einrückung, wobei `N` angibt, wie tief eingerückt wird.

In einigen Dokumenten kommen Aufzählungen vor, bei denen die einzelnen Aufzählungspunkte über eine Zeile hinausgehen. Abbildung 6.18 zeigt zwei verschiedene Arten, wie solche Aufzählungen erzeugt werden können. Nach dem ersten Aufzählungspunkt endet der Paragraph. Der nächste Absatz gehört zum vorherigen Aufzählungspunkt und wurde deshalb eingerückt. Beim zweiten Aufzählungspunkt wurde am Ende der Zeile der Paragraph nicht beendet, so dass die Aufzählung in der folgenden Zeile fortgesetzt wird. Eine Einrückung ist hier eigentlich nicht vorhanden, sie wird nur vom Textverarbeitungsprogramm dargestellt. Um die Strukturen in Abbildung 6.18 korrekt klassifizieren zu können, muss der erste Aufzählungpunkt so annotiert werden, dass die erste Zeile als Aufzählung (`<li>`) und die zweite Zeile als eingerückte Zeile (`<indent>`) gekennzeichnet wird. Da der komplette zweite Aufzählungspunkt ein Paragraph ist, muss er auch nur als Aufzählung ausgezeichnet werden.

Text Text Text Text		<p>
• Text Text	↵	
Text Text	↵	<indent>
• Text Text		
Text Text		
Text Text Text Text		<p>

Abbildung 6.18.: Trainingsmenge 6

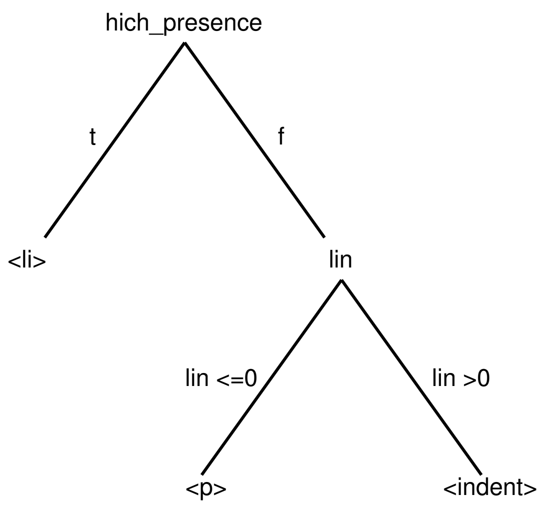

Abbildung 6.19.: Mit der Trainingsmenge 6 gelernter Entscheidungsbaum

Der Entscheidungsbaum in Abbildung 6.19 zeigt, dass die Paragraphen mit Aufzählungspunkten das Kontrollwort `hich` enthalten. Die Absätze sind eingerückt, wenn der Parameter des Kontrollwortes `lin` einen größer Wert als Null aufweist.

6.2.2. Anzahl der zu lernenden Strukturen variieren

Um fehlerhafte Annotationen zu vermeiden ist es wichtig, dass zu jeder Struktur, die im auszuzeichnenden Dokument vorhanden ist, eine Beispielklassifikation erstellt wird. Enthält ein Dokument beispielsweise neun unterschiedliche Strukturen und der Anwender zeichnet drei von ihnen manuell aus, so werden alle Strukturen des Dokumentes, die zu den sechs nicht ausgezeichneten Strukturtypen zählen, fehlerhaft annotiert. Die Ursache liegt darin begründet, dass ADT alle Paragraphen des Dokumentes klassifiziert. Absätze, für die dem Klassifizierer keine Klasse bekannt ist, können so nicht korrekt klassifizert werden.

6.2.3. Ähnliche Strukturen

In einigen Dokumenten kommen Textstrukturen vor, die sich semantisch unterscheiden. Von ihrer optischen Darstellung, also von ihrer visuellen Formatierung, sind sie jedoch identisch. Im Folgenden werden solche Texturen als *ähnliche Strukturen* bezeichnet. In Vorlesungsskripten kommt es beispielsweise des Öfteren vor, dass Definitionen und Beispiele, entgegen den im Kapitel 1 gezeigten Ausschnitten, gleich formatiert sind. Die Abbildung 6.20 zeigt solche ähnliche Strukturen.

...
Definition: Eine Assoziation ist eine Verbindung von einer Klasse bzw. einem Objekt zu einer anderen Klasse bzw. einem anderen Objekt. Insbesondere kann eine Assoziation von n Objekten einer Klasse auf m Objekte einer anderen Klasse gerichtet sein.

...

Beispiel: Jedes Objekt der Klasse Besitz „weiß", wem es gehört. Wenn nun ein Mensch alles, was er besitzt, versichern möchte, so bildet man die Behälterklasse Hausrat. Ein Objekt dieser Klasse besteht aus allen Objekten der Klasse Besitz, die diesem Menschen gehören. Wenn der Mensch etwas Neues bekommt, so wird diesem Objekt der Klasse Besitz mitgeteilt, dass es nun diesem Menschen gehört, und das Objekt der Klasse Hausrat wird aufgefordert, den neuen Besitz aufzunehmen.
...

Abbildung 6.20.: Die beiden Strukturen »Definition« und »Beispiel« sind bezüglich ihrer Formatierung identisch.

Eine Klassifikation der Paragraphen bezüglich der Formatierungsmerkmale ist hier nicht möglich.

Eine Gemeinsamkeit, die Paragraphen im beschriebenen Fall allerdings oft haben ist, dass sie sich bezüglich des ersten Wortes des Absatzes unterscheiden. Alle Definitionen werden durch »Definition:« eingeleitet. Die Bespiele beginnen entsprechend mit »Beispiel:«.

Um diese Eigenschaft auszunutzen, wird wie schon im Kapitel 5 beschrieben, das erste Wort der Absätze in die Klassifikation mit einbezogen.

Ein Versuch mit einem Dokument, das neben normalem Text ähnliche Strukturen enthielt, zeigte, dass pro Struktur nur ein Beispiel genügte, um die Paragraphen korrekt in ihre entsprechenden Gruppen einzuteilen.

Probleme treten auf, wenn beispielsweise alle Definitionen und Beispiele allein dadurch charakterisiert werden, dass sie mit dem Wort »Definition« bzw. »Beispiel« beginnen. Paragraphen die keine Definitionen oder Beispiele sind, aber auch mit dem Wort »Definition« oder »Beispiel« beginnen, werden ebenfalls als solche erkannt. Bei den durchgeführten Versuchen spielte die Formatierung der Paragraphen keine Rolle.

7. Evaluierung

Zu Beginn dieses Kapitels wird berechnet, wie viele Beispiele notwendig sind, um eine zufriedenstellende Klassifikation durchführen zu können. Anschließend wird mit einem Versuch getestet, wie viele Beispielklassifikationen das ADT-System unter Verwendung des J4.8 Algorithmuses benötigt, um Dokumentstrukturen zu klassifizieren. Zum Abschluss folgt eine Beschreibung eines Praxistests, bei dem die Laufzeit des Systems unter Verwendung von verschiedenen Algorithmen gemessen wird.

7.1. Wie viele Beispiele werden benötigt?

Um die Leistung des ADT-Systems beurteilen zu können, ist es zuerst einmal interessant zu bestimmen, wie viele Beispiele ein Lernverfahren benötigt, um zufriedenstellend zu klassifizieren.

Zuerst wird die Größe des Hypothesenraumes berechnet. Dazu bestimmt man die Anzahl der Attribute und die Anzahl der Attributwerte. Durchschnittlich gibt es 40 Attribute, die sich wie folgt zusammensetzen. Bei den Kontrollwörtern ohne Parameter gibt es 12 Attribute mit 2 Attributwerten und 12 mit 4 Attributwerten. Bei den Kontrollwörtern mit Parametern gibt es im Schnitt 8 Attribute mit 2 und 8 mit 4 Attributwerten.

Die Größe des Hypothesenraumes berechnet sich folgendermaßen:

Attribute:

$$\begin{aligned}
Attributwerte\, m_1 &= 2 \\
Attribute\, n_1 &= 12 \\
\\
Attributwerte\, m_2 &= 4 \\
Attribute\, n_2 &= 12 \\
\\
Attributwerte\, m_3 &= 2 \\
Attribute\, n_3 &= 8 \\
\\
Attributwerte\, m_4 &= 4 \\
Attribute\, n_4 &= 8
\end{aligned}$$

Hypothesenraum:

$$
\begin{aligned}
|H| &= m_1^{n_1} \cdot m_2^{n_2} \cdot m_3^{n_3} \cdot m_4^{n_4} \\
|H| &= 2^{8+12} \cdot 4^{8+12} \\
&= 2^{20} \cdot 2^{40} \\
&= 2^{60} \\
&= \underline{1,15292 \cdot 10^{18}}
\end{aligned}
$$

Die Anzahl m der mindestens benötigten Beispiele errechnet sich für PAC-Lerner nach [MORIK 2002] mit folgender Formel:

$$m \geq \frac{1}{\varepsilon}\left(ln(|H|) + ln\frac{1}{\delta}\right)$$

Setzt man den errechneten Hypothesenraum in die Formel ein,

$$
\begin{aligned}
& ln(|H|) + ln\frac{1}{0,2} \\
& ln(1,15292 \cdot 10^{18}) + ln\frac{1}{0,2} = \underline{\underline{43,19827}}
\end{aligned}
$$

ergibt sich, dass mindestens 43 Beispiele benötigt werden.

7.2. Evaluierung mit J4.8 Algorithmus

Nachdem der vorherige Abschnitt gezeigt hat, dass ein PAC-Lerner ca. 43 Beispiele benötigt, wird im Folgenden die Evaluierung des Systems unter Verwendung des J4.8 Algorithmuses beschrieben.

Als Versuchsdokument wurde das erste Kapitel des Skriptes [MORIK 1997] verwendet.

Im ersten Textdurchlauf wurde zu jeder Strukturart genau ein Beispiel gegeben. Die anschließend durchgeführte Klassifikation ergab, dass 41 Paragraphen korrekt und 118 inkorrekt klassifiziert wurden. Der sich durch dieses Ergebnis ergebende Wert für die *Precision* beträgt lediglich 25,79%.

In den anschließenden Versuchsdurchläufen wurde die Anzahl der Beispiele pro Strukturart um jeweils eins erhöht. Wie die Tabelle 7.1 zeigt, ergibt sich schon bei vier Beispielen pro Strukturart, dass 99,27% der Paragraphen vom System korrekt klassifiziert werden.

Die Abbildung 7.1 zeigt, wie schon bei zwei Beispielen pro Strukturart die Genauigkeit der Klassifikation sehr stark ansteigt.

Versuch	Label		Beispiele	Struktur-anzahl	Evaluierung	
1	korrekt	41	1	9	Precision:	**25,79%**
	inkorrekt	118			Recall:	**100,00%**
	nicht ausgegeben	0			F-Measure:	**41,00%**
2	korrekt	141	2	9	Precision:	**88,68%**
	inkorrekt	18			Recall:	**100,00%**
	nicht ausgegeben	0			F-Measure:	**94,00%**
3	korrekt	155	3	9	Precision:	**97,48%**
	inkorrekt	4			Recall:	**100,00%**
	nicht ausgegeben	0			F-Measure:	**98,73%**
4	korrekt	158	4	9	Precision:	**99,37%**
	inkorrekt	1			Recall:	**100,00%**
	nicht ausgegeben	0			F-Measure:	**99,68%**

Tabelle 7.1.: Precision-, Recall- und F-Measure-Werte die sich bei neun Klassen mit ein bis vier Beispielen pro Paragraphstruktur unter Verwendung des J.48 Algorithmus ergeben

7.3. Laufzeit im Praxistest

Um dem Leser eine Vorstellung zu geben, mit welcher Laufzeit er bei der Anwendung von ADT in der Praxis rechnen muss, wurde die reelle Laufzeit des Systems gemessen.

Die Testläufe für die Messungen wurden auf einem Targa Visionary 1300WS[1] Notebook durchgeführt.

Bei der Messung der Laufzeit erfolgte die Klassifikation der Paragraphen mit Hilfe der Algorithmen J4.8, KStar und IBk. Als Testdokument wurde zuerst das erste Kapitel des Skripts [Morik 1997] verwendet, welches 14 Seiten umfasst. Der anschließende Test umfasste die kompletten 172 Seiten des Skripts [Morik 1997].

Bei eingeschalteter Datenbereinigung konnten folgende Laufzeiten für das Einlesen und Parsen der Dokumente gemessen[2] werden:

erstes Kapitel: 215 ms - 450 ms
gesamtes Skript: 6840 ms - 7000 ms

[1]Kenngrößen: mobile AMD Athlon(tm) 4 Processor, CPU MHz: 1200, cache size: 256 kB, MemTotal: 482164 kB, Linux Version 2.4.19-4GB, KDE 3.0.4, Java(TM) 2 Runtime Environment - Standard Edition (build 1.4.1_01-b01)

[2]Die unterschiedlichen Laufzeiten haben verschiedene Ursachen. Zum einen kompiliert der JIT-Compiler bei der ersten Ausführung den JAVA-Byte-Code in Maschinencode. Beim erneuten Ausführen der Klassen fällt dieser Schritt zwar weg, verschiedene Hintergrundprozesse führen aber weiterhin zu unterschiedlichen Laufzeiten.

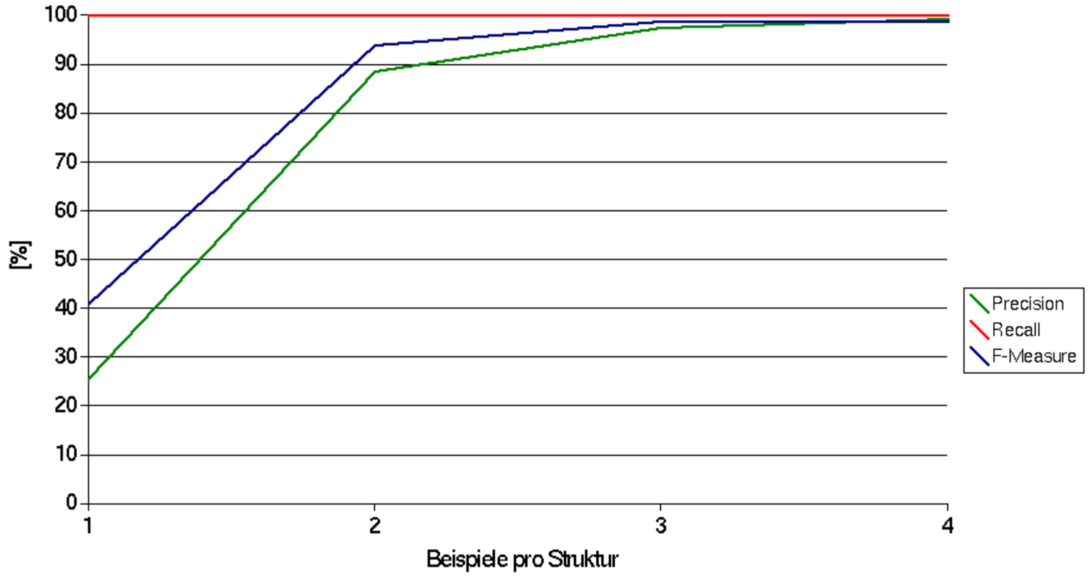

Abbildung 7.1.: Lernergebnisse nach vier Klassifikationen mit ein bis vier Beispielen pro Paragraphstruktur

Für die Klassifikation wurden sieben verschiedene Klassen mit insgesamt zwanzig Beispielen annotiert. Die Ergebnisse sind in Tabelle 7.2 sowie grafisch in Abbildung 7.2 dargestellt.

	erstes Kapitel	**gesamtes Dokument**
J4.8	25 ms - 30 ms	3500 ms - 4100 ms
KStar	480 ms	40200 ms
IBk	65 ms - 75 ms	7170 ms

Tabelle 7.2.: Laufzeitvergleich der Algorithmen J4.8, KStar und IBk für Attributgenerierung, Lernlauf und Klassifikation

Es kann nun festgehalten werden, dass J4.8 sowohl im kurzen als auch im langen Dokument ca. doppelt so schnell arbeitet wie IBk. Der Klassifizierungsalgorithmus KStar schneidet am schlechtesten ab und benötigt 10 bis 15 mal mehr Zeit für die Klassifikation als J4.8.

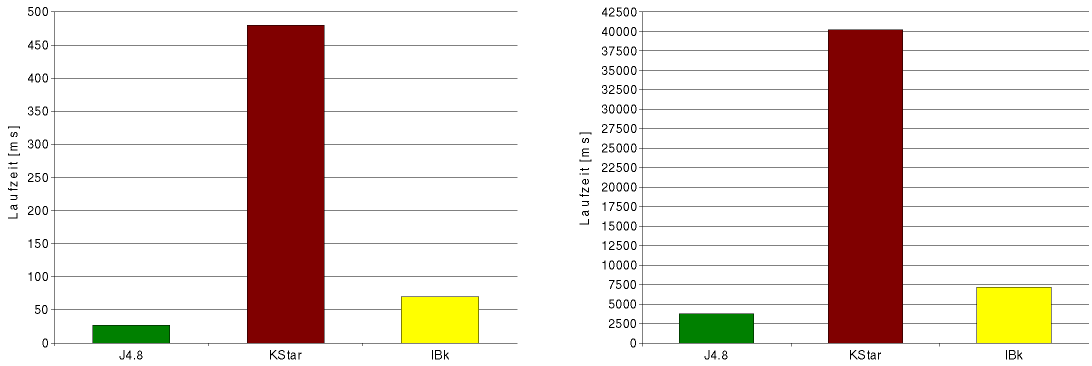

Abbildung 7.2.: Laufzeitvergleich der Algorithmen J4.8, KStar und IBk für Attributgenerierung, Lernlauf und Klassifikation. Links: Laufzeittest mit dem ersten Kapitel des Skriptes [Morik 1997]. Rechts: Laufzeit bezüglich des kompletten Skriptes [Morik 1997].

8. Implementierung

Dieses Kapitel beschreibt, die einzelnen Komponenten des entwickelten Systems. Dabei wird erklärt, wie bestimmte Systemmodule implementiert sind und warum sie so realisiert wurden.

8.1. Parser und Tokenizer

In Abschnitt 4.1.1 wurde bereits eine kurze Einführung in das Thema Parser gegeben. In diesem Abschnitt soll nun gezeigt werden, wie der für diese Diplomarbeit geschriebene Parser arbeitet.

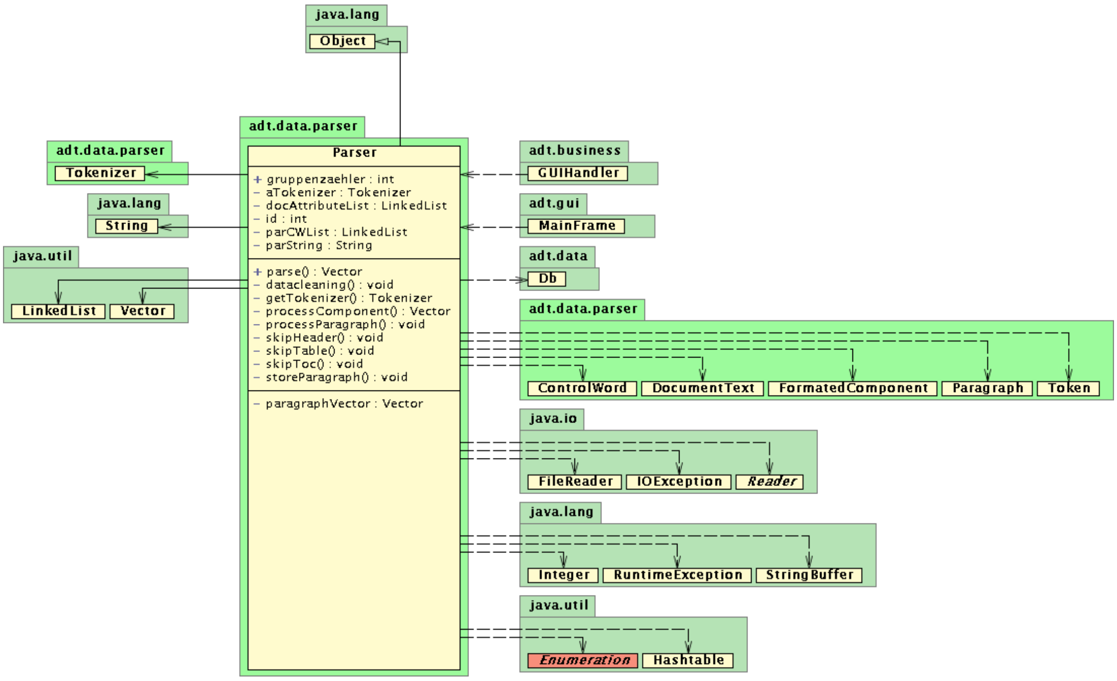

Abbildung 8.1.: UML Diagramme der Klasse `Parser.java`

8.1.1. Einlesen einer Datei

Um ein Dokument zu bearbeiten muss es natürlich zuerst einmal eingelesen werden. Wie in Kapitel 5 beschieben wählt der Anwender den entsprechenden Text über die Grafischeoberfläche aus. Die Klasse `FileHandler.java` öffnet daraufhin einen Datenstrom und liest mit Hilfe eines *gepufferten Readers* den Inhalt der Quelldatei zeilenweise aus. Die Hauptaufgabe des *parsings*, die Analyse der gelesenen Zeichenketten, erfolgt in der Klasse `Parser.java`.

Nachdem ein Eingabestrom zu einer Datei geöffnet wurde, kann die Auswertung der Daten beginnen. Hauptaufgaben in dieser Phase übernehmen die Klassen `Parser.java` und `Tokenizer.java`.

8.1.2. Tokenizer

Die Klasse `Tokenizer.java` übernimmt die Aufgabe eines Scanners. Die einzelnen Zeichen des Eingabedokuments werden von ihm gelesen. Abhängig von der Art des Zeichens werden daraufhin Teilzeichenketten, so genannte Tokens, zusammengesetzt.

Der Tokenizer unterscheidet die folgenden fünf Klassen von Tokens:

- **ControlWord**
  Ergeben die gelesen Zeichen ein Kontrollwort der RTF-Syntax, bilden sie ein Token das durch die Klasse `ConrolWord.java` repräsentiert wird.
- **DocumentText**
  Der eigentliche Text des Dokumentes wird durch Tokens der Klasse `DocumentText.java` dargestellt.
- **GroupStart**
  Findet der Tokenizer bei der Analyse des Textes eine öffnende Geschweifteklammer, so handelt es sich um den Beginn einer Gruppe. Der Gruppenstart wird durch ein Token der Klasse `GroupStart.java` repräsentiert.
- **GroupEnd**
  Findet der Tokenizer bei der Analyse des Textes eine schließende Geschweifteklammer, so handelt es sich um das Ende einer Gruppe. Das Gruppenende wird durch ein Token der Klasse `GroupEnd.java` repräsentiert.
- **DocumentEnd**
  Liefert die Klasse `FileHandler.java` keine Zeichen mehr, sondern gibt das Ende der Datei bekannt, wird das Token `DocumentEnd` erzeugt.

Die zuvor genannten Klassen sind in Abbildung 8.2 dargestellt. Da die Klassen einige Eigenschaften gemeinsam aufweisen, erben sie diese von der Klasse `Token.java`. Spezialisierungen sind in den einzelnen Unterklassen implementiert.

Damit die genannten Tokens aber erst einmal erkannt werden, liest der Tokenizer jedes einzelne Zeichen. Für diesen Vorgang wurden zwei wichtige Methoden implementiert:

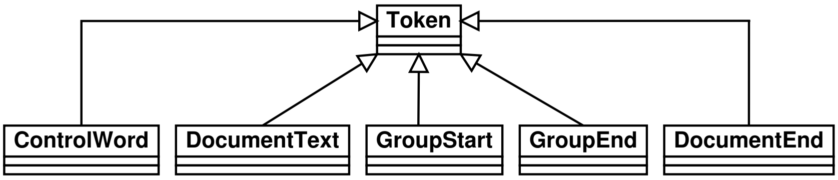

Abbildung 8.2.: UML Diagramm der Klassen die die einzelnen Tokens realisieren. Die Methoden und Attribute sind hier aus Gründen der Übersicht nicht dargestellt.

- `readChar():int`
- `peekChar(int i):int`

Die Methode `readChar():int` liest das nächste Zeichen ein. Mit der Methode `peekChar(int i)` ist es wiederum möglich nachzuschauen, welches Zeichen als nächstes gelesen wird, ohne es zu lesen[1]. Der Parameter `i` gibt dabei an, wie viele Zeichen »vorhergesagt« werden sollen. Der Tokenizer wurde mit einem Lookahead von zwei Zeichen implementiert, so dass i die Werte 0 und 1 annehmen kann. Zu wissen, welche Zeichen als nächstes gelesen werden, ist beispielsweise bei der Behandlung von Sonderzeichen oder Umlauten in Dokumenten wichtig:

Beispiel 8:

> Wie in Abschnitt 2.2.7 beschrieben, wird der Beginn eines Steuerwortes und eines Steuersymbols, in der RTF-Syntax, mit einem Backslash (\) eingeleitet.
>
> Zur Erinnerung noch einmal kurz die Hauptmerkmale dieser beiden Elemente:
>
> - Steuerwörter beginnen mit einem Backslash und werden durch ein Leerzeichen, eine Zahl oder ein Zeichen, das kein Buchstabe ist, begrenzt (dem so genannten Delimiter).
> - Steuersymbole bestehen nur aus einem Backslash gefolgt von einem nicht-alphabetischem Zeichen.
>
> Der Tokenizer soll nun den folgenden Text einlesen:
>
> *An dieser Stelle geht es darum, Beschreibungen im wissenschaftlichen Sinne etwas genauer zu fassen. \par*
>
> Alle Zeichen dieses Textes können zu einer Zeichenkette konkateniert werden und in einer Instanz der Klasse `DocumentText` gespeichert werden, solange bis das Zeichen »\« gelesen wird. Da der Backslash ein Steuerwort einleitet, wird nun ein neues Token der Klasse `ControlWord` erzeugt.
>
> Im folgenden Satz leitet der Backslash hingegen ein Steuersymbol ein:
>
> *Was ist K\'fcnstliche Intelligenz?*

[1] Das ist keine »Zauberei«, es sind lediglich zwei Zeichen gepuffert, so dass beim Nachsehen des nächsten Zeichens kein weiteres vom Datenstrom gelesen werden muss.

> Mit einem Tokenizer mit einem Lookahead von zwei ist es hier möglich, das nicht-alphabetische Zeichen »'« zuerkennen, bevor durch das Lesen des Backslashes ein neues Token erzeugt wird. Erkennt der Tokenizer also frühzeitig das Zeichen »'« nach einem Backslash, können die folgenden zwei Zeichen in ein Umlaut umgewandelt werden. Hier ergibt beispielsweise »\'fc« den Buchstaben »ü«.

Die privaten Methoden `readChar()` und `peekChar(int i)` werden also für die Erzeugung der verschiedenen Tokens und zur Behandlung von Sonderzeichen genutzt. Der Parser selbst kann nun die erstellten Tokens verarbeiten und braucht sich nicht mehr um einzelne Zeichen zu kümmern.

Analog zu den oben genannten Methoden für einzelne Zeichen stehen hierfür folgende Methoden zur Verfügung:

- `peekType(int i):int`
  liefert den Typ des nächsten bzw. des übernächsten Tokens, ohne das weitere Zeichen aus dem Datenstrom gelesen werden müssen.
- `peekToken(int i):Token`
  liefert das nächste bzw. das übernächste Token, ohne das weitere Zeichen aus dem Datenstrom gelesen werden müssen.
- `readToken():Token`
  liefert das nächste Token und veranlasst die Erzeugung eines neuen Tokens wofür weiter Zeichen aus dem Datenstrom gelesen werden.

Die Hauptaufgabe des Tokenizers, aus dem einzulesenden Text Teilstrings zu bilden, kann mit den vorgestellten Methoden erledigt werden.

8.1.3. Header-Gruppe

Ein RTF-Dokument besteht zuerst einmal aus den beiden »Hauptgruppen« *Document* und *Header*. Die Untergruppen der Gruppe Header definieren verschiedene Bereiche des Layouts innerhalb eines Dokumentes. Beispielsweise werden hier Schriften, eingebundene Dateien, verwendete Farben oder Stilvorlagen (Stylesheets) festgelegt. Da allerdings die im Header gemachten Festlegungen nicht zur Lösung des Problems der Diplomarbeit verwendet werden können, überspringt der Parser die Header-Gruppe.

Es erscheint vielleicht etwas merkwürdig, wenn hier behauptet wird, dass die Header-Informationen nicht beachtet werden müssen. Deshalb soll das folgende Beispiel die Richtigkeit der Aussage veranschaulichen:

Beispiel 9:

Die Gruppe *Stylesheets* enthält Definitionen für verschiedene Formate die im Dokument verwendet werden können. Die Formate legen dann beispielsweise die Schriftart, -größe oder die zu verwendende Farbe fest.

```
{\stylesheet{\s0\f3\fs20\qj Normal;}}
```

Hier wird ein Format definiert und mit dem eindeutigen Schlüssel \s0 bezeichnet. Anschließend wird für dieses Format die Schrift 3 (\f3), eine Schriftgröße von 10 Punkten (\fs20) und Blocksatz (\qj) festgelegt. »Normal« ist die Bezeichnung des Formates, wobei der Name nicht eindeutig sein muss. Im eigentlichen Text des Dokumentes dienen die in der Stylesheet-Gruppe definierten Formate zur Formatierung des Textes und werden über den eindeutigen Schlüssel referenziert. Also muss in allen Textstellen, die mit einem bestimmten Stylsheet formatiert sind, auch der Schlüssel enthalten sein, der genau zu der dort verwendeten Stilvorlage gehört. Der Inhalt der Stilvorlage ist somit irrelevant. Wichtig ist nur, dass alle Textstellen, die mit einer bestimmten Stilvorlage formatiert wurden, ein gemeinsames Attribut aufweisen, den Schlüssel, welcher in keinem anders formatierten Text vorkommt.

Die selbe Ausgabe kann über die anderen Festlegungen der Stylsheet-Gruppe gemacht werden. Gleichgültig ob Farben oder Schriften definiert werden. Sie werden stets über einen eindeutigen Schlüssel referenziert.

Aufgrund der im Beispiel 9 genannten Fakten können der Header-Gruppe keine verwendbaren Informationen entnommen werden, das Überspringen der Header-Gruppe ist also korrekt.

8.1.4. Datenbereinigung

Die Datenbereinigung (data cleaning) stellt einen Teil des Preprocessingschrittes innerhalb von ADT da. Allgemein geht es bei der Datenbereinigung um das Entfernen fehlerhafter oder irrelevanter Daten aus einer Datenmenge.

Wie bereits im Kapitel 4 erklärt, sind die Daten, die aus einem Dokument gewonnen werden, in der Regel nicht fehlerhaft, es finden sich allerdings eine Reihe von Daten die keine Informationen zur Lösung des Problems beitragen.

Zu Beginn des ersten Vorverarbeitungsschritts wird eine Liste der irrelevanten Kontrollwörter aufgestellt. Die Methode `datacleaning()`, der Klasse `Parser.java`, erstellt, nach der in Abschnitt 4.1.2 genau beschrieben Verfahren, eine Liste aller Steuerwörter. Anschließend können dann alle Kontrollwörter entfernt werden, die nicht in jedem Paragraphen vorkommen. Am Ende dieses Vorganges enthält die Liste alle für die Klassifikation irrelevanten Steuerwörter.

8.1.5. Verarbeitung der Paragraphen

Eine zentrale Aufgabe, das Dokument in Paragraphen zu zerlegen, übernimmt die Klasse `Parser.java`. Der Parser fordert vom Tokenizer einzelne Tokens an und speichert sie in einem Objekt der Klasse `Paragraph.java`. Findet der Parser das Token `\par` bedeutet dies, dass nun ein neuer Paragraph beginnt und ein neues Paragraph-Objekt angelegt werden muss. Die einzelnen Paragraphen-Objekte werden in einem Vektor gespeichert. Dieser wird dann bei der Datenbereinigung[2] und bei der im Abschnitt 8.2 erläuterten Klassifikation genutzt.

Abbildung 8.3 zeigt ein UML-Diagramm der Klasse `Paragraph.java`

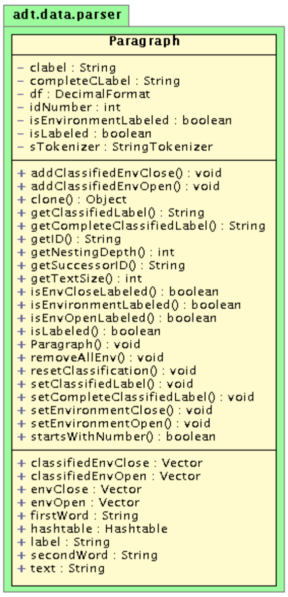

Abbildung 8.3.: UML Diagramme der Klasse `Paragraph.java`

Wie speichert der Parser die vom Tokenizer erzeugten Tokens in einem Paragraph-Objekt? Die RTF-Syntax bietet die Möglichkeit Gruppen zu bilden. Definitionen die innerhalb einer Gruppe gemacht wurden, haben auch nur innerhalb dieser und ihren Untergruppen Gültigkeit. Es ist also wichtig, dass der Parser nicht einfach zwischen Kontrollwörtern und Textelementen unterscheidet, sondern es muss gleichzeitig die Klammerstruktur beachtet werden. Nur dann kann auch festgestellt werden, welches Kontrollwort wie viele Zeichen innerhalb eines Paragraphen beeinflusst.

Bei nicht eingeschalteter Datenbereinigung werden alle Kontrollwörter des Paragraphen in einer Hashtabelle gespeichert. Dabei bildet ein Kontrollwort jeweils einen Schlüssel und die Anzahl der Zeichen, die vom Kontrollwort beeinflusst werden, wird als zugehöriger Wert gespeichert. Ist die Datenbereinigung aktiviert, wird für jedes Kontrollwort überprüft, ob es sich in der Menge der irrelevanter Kontrollwörter (siehe 8.1.4) befindet.

[2]der Vektor entspricht der im Abschnitt 4.1.2 beschiebenen Menge B.

Ist das nicht der Fall, so wird das Kontrollwort in der Hashtabelle gespeichert, andernfalls wird es verworfen.

Die komplette Hashtabelle und ein String, der dem Text des Paragraphen entspricht, wird letztlich im Paragraph-Objekt gespeichert.

Der Vollständigkeit halber sollte erwähnt werden, dass Gruppen, die Destinations enthalten, durch den Parser übersprungen werden. Fußnotentext oder Einträge im Inhaltsverzeichnis eines Dokumentes werden demzufolge nicht betrachtet.

8.2. Klassifikation

Dieser Abschnitt erläutert die Klassifikation der Paragraphen eines Dokuments. Zum einen wird die manuelle Auszeichnung der Paragraphen durch den Benutzer aus implementationstechnischer Sicht beschrieben. Zum anderen werden die Schritte aufgezeigt, die notwendig sind, um Lernalgorithmen aus dem WEKA-Paket zu verwenden. Der abschließende Abschnitt beschäftigt sich mit der Überprüfung der Klassifikation.

8.2.1. manuelle Auszeichnung

Wie in Kapitel 5 ausführlich beschrieben, ist es notwendig, dass der Anwender eine Beispielklassifikation erstellt. Die Auszeichnung der Paragraphen erfolgt, wie erwähnt, über grafische Auswahlmenüs.

Wählt ein Benutzer für einen Paragraphen einen bestimmten Typen aus, wird dieser dem Paragraphen zugeordnet. Die in Abbildung 8.3 dargestellte Klasse `Paragraph.java` besitzt die Methode:

- `setClassifiedLabel(String label):void`

Die Methode speichert den vom Benutzer klassifizierten Typen als String im entsprechenden Paragraphen-Objekt. Einem Paragraphen kann nur ein einzelner Typ zugeordnet sein. Ordnet der Anwender einem Paragraphen beispielsweise zuerst den Typ *Beispiel* zu und anschließend den Typ *eingerückt*, um ein eingerücktes Beispiel zu klassifizieren, bedeutet dies nicht, dass der Paragraph nun der Klasse *eingerückt* und der Klasse *Beispiel* zugeordnet wird. Es wird grundsätzlich die zuletzt durchgeführte Klassifikation verwendet. Soll nun ein Paragraph mehr als einer Klasse angehören, muss hierfür eine neue Klasse verwendet werden. Im genannten Beispiel wäre dementsprechend eine Klasse *eingerücktes_Beispiel* denkbar.

Neben der Festlegung des Typs eines Paragraphen, kann zusätzlich auch die Umgebung, in der sich ein Paragraph befindet, festgelegt werden. Hier ist es allerdings im Gegensatz zum Typ möglich, mehrere Labels auszuwählen. Dadurch wird die Möglichkeit geschaffen, anzugeben, dass sich ein Paragraphen in einer oder in mehreren ineinander geschachtelten

Ebenen befindet. In Dokumenten kommt es beispielsweise des Öfteren vor, dass sich innerhalb einer Aufzählung eine zweite Aufzählung befindet.

Beim ersten Aufruf der Methoden:

- addClassifiedEnvOpen(String env):void
- addClassifiedEnvClose(String env):void

wird ein Vektor angelegt und das Label für die Umgebung abgespeichert. Bei weiteren Aufrufen der Methoden wird der Vektor um das entsprechende Label `env` erweitert.

8.2.2. Generierung der Instanzen

Um die Algorithmen, die das WEKA-Paket zur Verfügung stellt, nutzen zu können, müssen die zu bearbeiteten Daten entweder in Form einer ARFF-Datei[3] vorliegen oder es wird ein Objekt der Klasse Instances[4] verwendet. Ein Instances-Objekt stellt dabei eine Menge von Instanzen dar.

Für die Erstellung der einzelnen Instanzen und des Instances-Objekts innerhalb von ADT sind die, in Abbildung 8.4, dargestellten Klassen `ParLeaner.java` und `ParClassifier.java` zuständig.

Beim Instanziieren eines Instance-Objektes muss bekannt sein, welche Attribute im gesamten Dokument vorkommen. Die Methode

- `collectAttributes(Vector parVector):void`

der Klasse `WEKAHandler.java` stellt deshalb verschiedene Listen zusammen. Zum einen eine Liste aller Kontrollwörter mit Parameter, zum anderen eine Liste aller Kontrollwörter ohne Parameter und weiterhin eine Liste aller verwendeten Annotationen.

Für die einzelnen, durch einen Anwender, annotierten Paragraphen, kann anschließend die Klasse `ParLeaner.java` überprüfen, welche Attribute, aus der Menge aller Attribute eines Dokuments, bei einem bestimmten Paragraphen vorhanden sind und welche nicht.

Um zusätzliche Attribute zu erzeugen, wodurch die Paragraphen besser klassifiziert werden können, werden wie schon erläutert, zwei Paragraphen verglichen und die Änderungen durch zusätzliche Attribute kenntlich gemacht. Die Analyse der Unterschiede übernehmen die Methoden der Klasse `ParAnalyser.java`:

- `getAddedCW(Paragraph par1, Paragraph par2):LinkedList`
  Erstellt eine Liste aller Kontrollwörter, die im zweiten, jedoch nicht im ersten der beiden verglichenen Paragraphen vorhanden sind.

[3]Das ARFF-Format (**A**ttribute-**R**elation **F**ile **F**ormat) ist ein ASCII-Format das eine Menge von Instanzen beschreibt und innerhalb von WEKA eingesetzt wird.

[4]weka.core.Instances

- `getLeftOutCW(Paragraph par1, Paragraph par2):LinkedList`
  Erstellt eine Liste aller Kontrollwörter, die im zweiten Paragraphen, verglichen mit dem ersten der beiden untersuchten Paragraphen, nicht mehr vorhanden sind.

- `valueAnalysis(Paragraph par1, Paragraph par2):Hashtable`
  Analysiert bei Kontrollwörtern mit numerischen Parametern, ob die verglichen Parameter, zweier Paragraphen, den selben, einen kleineren oder einen größeren Wert besitzen.

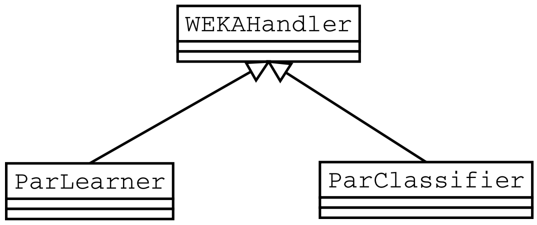

Abbildung 8.4.: `ParLearner.java` und `ParClassifier.java` sind Spezialisierungen vom `WEKAHandler.java`

Enthält ein Dokument aufeinanderfolgende Paragraphen, die eine zusammenhängende Struktur ergeben, können diese von ADT erkannt werden. Im Abschnitt 4.1.4 wurde gezeigt, dass der nachfolgende Paragraph einer solchen Struktur, gegenüber den Paragraphen der Struktur, bestimmte Attribute aufweist bzw. nicht mehr aufweist. Das Ende einer zusammenhängenden Struktur, wie z. B. Aufzählungen, sind also erst nach dem eigentlichen Ende der Struktur erkennbar. Ein Anwender, der Auszeichnungen mit Hilfe der Benutzeroberfläche erstellt hat, kennzeichnet korrekterweise das Ende am letzten Paragraphen der Struktur, da es hier allerdings vom Klassifikationalgorithmus nicht erkannt wird, verschiebt das System die Annotation, wie in Tabelle 8.1 gezeigt, zum Nachfolgerparagraphen.

Attribute										Annotation
tab	tab_C	bullet	bullet_C	li	li_C	lin	lin_C	par	par_C	
f	ff	f	tf	t	ff	t	tf	t	tf	P
t	tf	t	tf	t	tt	t	tt	t	tt	li_ul
t	tt	t	tt	t	tt	t	tt	t	tt	li
t	tt	t	tt	t	tt	t	tt	t	tt	li_/ul
f	ft	f	ft	t	tt	t	tt	t	tt	P

Tabelle 8.1.: Die Annotation, die das Ende der Aufzählung markiert, wird zum nachfolgenden Paragraphen verschoben.

8.2.3. Automatische Klassifikation anhand des erzeugten Entscheidungsbaums

Wurden mit Hilfe aller annotierten Paragraphen Instanzen erzeugt und diese verwendet um ein Instances-Objekt zu instanziieren, kann ein Klassifizierer gebildet werden. Je nachdem welcher Algorithmus innerhalb der Graphischenoberfläche gewählt wurde, erzeugt die Methode

- `learn(String algo):void`

einen entsprechenden Klassifizierer.

Die in ADT verwendeten Klassifizierer stammen alle aus dem WEKA-Paket der Universität Waikato und erben von der Klasse `Classifier`. Hier erzeugt der Aufruf der Methode `buildClassifier(Instances)` den entsprechenden Klassifizierer. Detaillierte Informationen zum erzeugen eines Klassifizierer finden sich in [Witten und Frank 2000] und in der API des WEKA-Paketes[5].

Entsprechend der Vorgehensweise bei der Erstellung der Instanzen, der annotierten Paragraphen, werden in der Klasse `ParClassifier.java` Instance-Objekte für alle nicht ausgezeichnete Paragraphen erzeugt. Die Methode

- `classify(Instance instance, String algo):void`

ermittelt dann für die noch nicht ausgezeichneten Absätze ihre Klassenzugehörigkeit.

Auszeichnungen, die das Ende einer zusammengehörigen Gruppe von Paragraphen kennzeichnet und wie erwähnt verschoben wurden, werden nach der Klassifikation zum vorausgehenden Paragraphen verschoben. Somit ist gewährleistet, dass das Gruppenende auch am eigentlichen Ende der Gruppe erkannt wird.

8.2.4. Verifikation

Damit es nicht zu Unstimmigkeiten zwischen den Annotationen des Anwenders und den durch den Klassifizierungsalgorithmus vorgenommenen Auszeichnungen kommt, wird überprüft ob diese Auszeichnungen übereinstimmen. Die Methode

- `verify():void`

vergleicht die Strings der manuellen und der maschinellen Annotation. Bei Nichtübereinstimmung wird die maschinell erstellte Auszeichnung rot in der Benutzerschnittstelle angezeigt, so dass der Benutzer den Fehler schnell erkennt und weiter Beispiele für die falsch klassifizierte Klasse annotiert, so dass der Fehler bei einer erneuten Klassifikation behoben werden kann.

[5] http://www.cs.waikato.ac.nz/~ml/weka/doc_book/packages.html

8.3. Ausgabe des annotierten Dokumentes

Für die Darstellung der Ergebnisse des Klassifikationsvorgangs in der Benutzeroberfläche ist die Klasse `GUIHandler.java` verantwortlich. Die Ausgabe des annotierten Dokumentes als XML-Dokument geschieht allerdings durch die Klasse `XMLCreator.java`. Die Methode

- `createXMLFile():void`

erstellt eine XML-Datei. Die einzelnen Paragraphen werden dabei, wie bei XML üblich, von einem öffnenden und einem schließenden Tag umschlossen. Der Tag entspricht dabei der Annotation des jeweiligen Paragraphen.

9. Ausblick und Zusammenfassung

In diesem letzten Kapitel sollen noch einmal die in der Einleitung aufgestellten Anforderungen zusammengefasst werden. Anschließend wird erörtert inwiefern die gesetzten Ziele erreicht werden konnten.

Der Abschnitt 9.2 diskutiert Funktionen und Lösungen, die nicht in ADT implementiert wurden (meist aus Zeitgründen).

9.1. Zusammenfassung

Ziel dieser Arbeit war es, ein System zu schaffen, mit dem es möglich ist, bei Dokumenten, die lediglich mit physikalischen Textattributen wie Schriftgröße oder Schriftschnitt formatiert sind, eine semantische Annotation vorzunehmen. Also beispielsweise eine Überschrift für das System als solche erkennbar zu markieren.

In der Einleitung wurden mehrere Anforderungen an das zu entwickelte System gestellt. Im Folgenden wird nun untersucht, inwieweit die einzelnen Anforderungen erfüllt wurden.

Anforderung 1: Das System soll absatzbezogene und geschachtelte Auszeichnungen durchführen.

In Kapitel 6 wurde in verschiedenen Versuchen gezeigt, wie ADT absatzbezogene Auszeichnungen vornimmt und dass wenig Beispielklassifikationen notwendig sind, um gute Ergebnisse zu erlangen.

Da ADT Attribute-Werte-Lernverfahren verwendet sind keine grammatikalischen Konzepte und Relationen vorhanden. Um geschachtelten Strukturen korrekt annotieren zu können erweist es sich als sinnvoll, alle zweistelligen Relationen durch vier Attribute zu approximieren.

Anforderung 2: Die Klassifizierung von Struktureinheiten der Dokumente soll anhand von Textattributen erfolgen.

Das eine Klassifikation von Paragraphen anhand ihrer Formatierungsmerkmale erfolgen kann, konnte gezeigt werden. Dokumente die im verwendeten RTF-Format gespeichert sind weisen jedoch eine Fülle an Kontrollwörtern auf, die für die Klassifikation unwichtig sind, da sie keine Informationen bieten, die zur Klassifikation verwendet werden könnten. In Kapitel 4 wurde gezeigt, wie diese unwichtigen Kontrollwörter gelöscht werden können.

Um die Paragraphen besser klassifizieren zu können, müssen neben dem Löschen der irrelevanten Kontrollwörter allerdings auch neue Attribute eingeführt werden. Das Wegfallen und Hinzukommen von Kontrollwörter bezüglich zweier aufeinanderfolgenden Paragraphen wird durch vier zusätzliche Attribute beschrieben. Weitere vier Attribute werden bei allen Kontrollwörtern mit Parameterwerten hinzugefügt. Diese beschreiben ob sich der Parameterwert verkleinert oder vergrößert hat, ob er gleich geblieben oder ganz weggefallen ist.

Bereitet man die, durch die Formatierung der Paragraphen gegebenen Attribute, wie beschrieben auf, können die Absätze eines Dokumentes anhand von Formatierungsattributen klassifiziert werden.

Anforderung 3: Dem Benutzer soll die Möglichkeit gegeben werden, interaktiv das Verfahren zu beeinflussen.

Durch interaktives Eingreifen des Anwenders kann die Anzahl der Beispiele, die für eine zufriedenstellende Annotation eines Dokumentes notwendig sind, entscheidend verringert werden. ADT bietet die Möglichkeit, das Ergebnis eines Klassifikationvorganges direkt am Bildschirm zu überprüfen. Einige falsch ausgezeichnete Paragraphen können anschließend manuell korrigiert und der Klassifikationsvorgang erneut gestartet werden. Dieses Vorgehen ermöglicht es, dass wenig Beispiele benötigt werden um ein Dokument korrekt zu annotieren, da genau zu den Klassen Beispiele gegeben werden, bei denen es während der vorherigen Klassifikation noch Probleme gab.

Anforderung 4: Mit dem System soll aktiv gearbeitet werden, die Klassifikation muss also sehr schnell ablaufen.

Der im vorherigen Abschnitt beschriebene Arbeitsablauf erfordert geringe Laufzeiten der Klassifikationvorgänge. Die in Kapitel 7 beschriebenen Laufzeittests ermittelten Laufzeiten von ca. 30 ms für ein Dokument mit 14 Seiten und ca. 3500 ms für ein Dokument mit 172 Seiten unter Verwendung des J4.8 Algorithmuses.

Anforderung 5: Das System soll ein von möglichst vielen Textverarbeitungsprogrammen unterstütztes Datenformat einlesen können.

Das am meisten verbreitetste Textverarbeitungsprogramm ist wahrscheinlich *Microsoft Word*. Da das von *Word* verwendete DOC-Format allerdings ein Binäresformat ist und keine Spezifikation dieses Formates freizugänglich ist, wurde dies Format im Rahmen dieser Diplomarbeit nicht verwendet.

Das ebenfalls von *Microsoft* stammende Datenformat RTF erweist sich durch die Hohe Unterstützung in vielen Textverarbeitungsprogrammen und der Tatsache, dass es im ASCII-Format gespeichert wird, als geeignet. Durch die Verwendung des RTF-Formates kann ADT von vielen Anwendern ohne großen Aufwand genutzt werden. Der Benutzer speichert seine Dokumente lediglich statt im Standardformat seiner Textverarbeitungs-anwendung im RTF-Format ab.

Anforderung 6: Die bearbeiteten Texte sollen so abgespeichert werden können, dass sie neben absatzbezogenen und geschachtelten Annotationen keine visuellen Auszeichnungen mehr enthalten.

Nach dem die Klassifikation zufriedenstellend durchgeführt wurde, kann das annotierte Dokument als XML- oder HTML-Datei gespeichert werden. Ein Beispiel einer mit ADT erstellten Datei zeigt Abbildung 5.5.

Anforderung 7: Der Benutzer soll die Möglichkeit erhalten, die Metainformationen, welche die Dokumente annotieren, frei zu wählen.

Neben einigen standard HTML-Tags kann ein Anwender auch beliebige Tags definieren und so ein individuelles Dokument erzeugen.

9.2. Ausblick

Diese Arbeit konnte zeigen, dass es möglich ist Dokument-Strukturauszeichnungen aus Formatierungsmerkmalen maschinell zu lernen. Im Folgenden sollen noch einmal verschiedene Lösungsvarianten und ihre Komplexität dargestellt werden:

Variante 1: Wird die Grammatik eines Dokumentes maschinell erlernt, erweist sich dies als eine sehr komplexe und zeitaufwändige Aufgabe. Die Erstellung der weiterhin notwendigen Regeln, von Hand, stellt sich dann als nicht mehr aufwändig dar. Im Ganzen betrachtet ist diese Variante jedoch, durch den hohen Aufwand beim Maschinellen Lernen, sehr aufwändig.

Variante 2: Eine Lösung, bei welcher der Gesamtaufwand geringer ausfällt, ist die in ADT umgesetzte Variante. Eine einfache Grammatik ist fest im System vorgegeben, wurde also von Hand erstellt. Der Aufwand Regeln zu erlernen ist hier nicht hoch. Im Gegensatz zur Variante 1 ist die Komplexität dieser Lösung hier entscheident geringer.

Variante 3: Erstellt man eine sehr komplizierte Grammatik von Hand ist der Aufwand hierfür natürlich auch höher als bei einer einfachen. Das Maschinelle Lernen der Regeln wird hier ebenfalls aufwändiger als bei der Variante 2.

Abbildung 9.1 stellt die Komplexität der unterschiedlichen Varianten grafisch dar. Gut zu erkennen ist der deutliche Aufwandsunterschied zwischen der ersten und zweiten, sowie der zweiten und dritten Variante. Dies untermauert noch einmal die Entscheidung, bei der Realisierung des ADT-Systems, die Variante zwei zu verwenden.

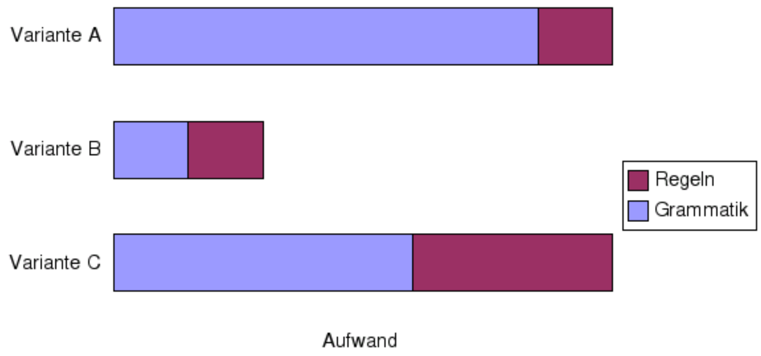

Abbildung 9.1.: Vergleich der Aufwände der verschiedenen Lösungsvarianten

Verlagerung des Arbeitsaufwands: Diese Diplomarbeit hat gezeigt, dass es Vorteilhaft ist, die Lernaufgabe möglichst leicht zu gestalten, indem grammatische Formen im Preprocessing behandelt werden. Je einfacher die Lernaufgabe also gestaltet wird, desto umfangreicher werden die Preprocessing-Schritte.

Erweiterungen: Diese Arbeit hat aufgezeigt, dass es möglich ist Dokument-Strukturauszeichnungen aus Formatierungsmerkmalen maschinell zu lernen. Die Betrachtung der Merkmale erfolgt bei ADT bezüglich eines Absatzes. Interessant wäre, auch die Formatierung einzelner Wörter mit einzubeziehen. So könnten auch auf Wortebene Annotationen durchgeführt werden. Dies bedeutet allerdings einen starken Anstieg der Komplexität, so dass diese Erweiterung nicht leicht durchzuführen ist.

Eine weitere Verbesserung des Ansatzes, Strukturauszeichnungen aus Formatierungsmerkmalen zu lernen, brächte die Unterstützung weiterer verbreiteter Formate wie *Postscript (PS)* oder *Portable Document Format (PDF)*. Zur Zeit gibt es zwar einen »Workaround«, so dass diese Formate in RTF konvertiert werden können. Ab der Version 5 des *Adobe Acrobat* ist es möglich, PDF sowie PS-Dokumente im RTF-Format zu speichern. Da die Konvertierung allerdings nicht immer korrekt funktioniert, ist dieser Umweg keine zufriedenstellende Lösung.

Eine zusätzliche Erweiterung, welche die Bedienbarkeit von ADT verbessern würde, wäre die Möglichkeit, einmal gelernte Entscheidungsbäume auf mehrere Dokumente anwenden zu können. In der aktuellen Version muss jedes neu geöffnete Dokument auch erneut manuell annotiert werden. Bei Dokumenten die sich bezüglich der Formatierung ihrer Paragraphen nicht unterscheiden, könnte so die zweite manuelle Auszeichnung eingespart werden.

Anhang

Der Anhang beschreibt kleine Erweiterungen innerhalb von ADT. Weiterhin wird erläutert, wie ADT auf verschiedenen Systemen installiert und gestartet wird.

9.3. Erweiterungen

Wie in den Versuchen aus Kapitel 6 bereits beschrieben wurde, kann es bei der Klassifikation von verschachtelten Strukturen zu Problemen kommen, wenn ein Paragraph mehrere Ebenen gleichzeitig schließt. D.h., dass bei einem Rücksprung eine oder mehrere Ebenen übersprungen werden. Das System erkennt an dieser Stelle zwar das eine Ebene geschlossen wird, wie viele Ebenen geschlossen werden bleibt allerdings unberücksichtigt. Um diesen Missstand zu beheben, wurde ADT so erweitert, dass nun die Anzahl der öffnenden bzw. schließenden Ebenen berücksichtigt wird.

Infolgedessen wurde der Parser und der zweite Preprocessing-Schritt, die Instanzenerzeugung, erweitert:

Während des Parsens wird festgehalten, wie viele verschiedene Parameterwerte des Kontrollwortes `lin` im zu bearbeitenden Dokument vorkommen. Diese Werte werden sortiert in einem Vektor gespeichert. Anschließend erhält jeder Paragraph ein zusätzliches Attribut, in dem gespeichert wird, in welcher Ebene sich der jeweilige Absatz befindet. Dabei entspricht die Ebene der Position des Parameterwertes des Kontrollwortes `lin`, des jeweiligen Absatzes, im Vektor. Im zweiten Preprocessing-Schritt wird beim Vergleich der benachbarten Paragraphen auch das Attribut, welches die Ebene angibt, verglichen. Dabei wird der Attributwert des n-ten Paragraphen von dem des $n+1$-sten Paragraphen abgezogen. Das Ergebnis stellt ein weiteres Attribut für den Klassifikationsvorgang dar. Ein negativer Wert gibt an, dass entsprechend viele Ebene geschlossen werden. Ein positiver Wert zeigt geöffnete Ebenen an. Bleibt der Attributwert des Ebenenvergleichs Null, gibt es keine Veränderung der Ebene von Paragraph n zu Paragraph $n+1$.

9.4. ADT installieren und starten

ADT ist plattformunabhängig und läuft auf jedem System, auf dem die im Folgenden aufgeführten JAVA-Versionen verfügbar sind.

ADT installieren:

Voraussetzung für den Start von ADT ist das *Java 2 Runtime Environment (JRE)* Version 1.4.0 oder Version 1.4.1. Anwender die die Quelldateien der Systems verändern und neu kompilieren wollen, benötigen das *Java 2 Software Development Kit (SDK)* Version 1.4.0 oder Version 1.4.1.

Ist die benötigte JAVA-Version installiert, kann das Paket `adt.jar` in ein beliebiges Verzeichnis kopiert werden.

ADT starten:

Um ADT auszuführen, muss das Paket `adt.jar` folgendermaßen aufgerufen werden.

Windows-Systeme:

1. MS-DOS-Eingabeaufforderung öffnen
2. Starten durch Eingabe von `java -jar adt.jar`

UNIX-Systeme:

1. Terminal-Fenster öffnen
2. Starten durch Eingabe von `java -jar adt.jar`. Siehe Abbildung 9.2

Macintosh-Systeme:

1. Terminal-Fenster öffnen
2. Starten durch Eingabe von `java -jar adt.jar`

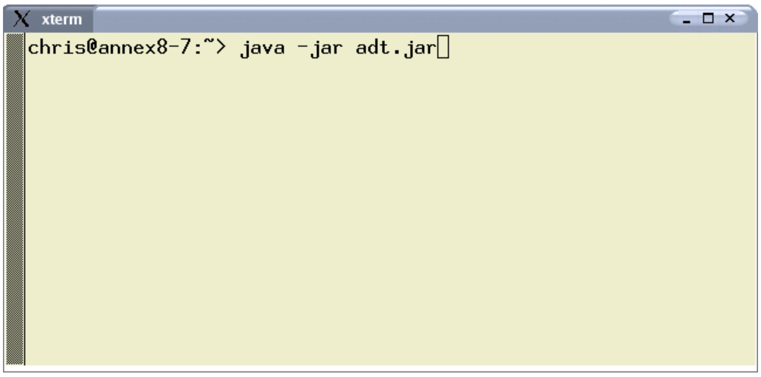

Abbildung 9.2.: ADT im xterm starten

Literaturverzeichnis

[ric 2001] (2001). *Rich Text Format (RTF) Specification*. Microsoft. Version 1.7.

[Aho et al. 1989] Aho, Alfred V., R. Sethi und J. D. Ullman (1989). *Compiler-Bau*. Teil 1. Addison-Wesley, Universität Bonn.

[Altamura et al. 2000] Altamura, Oronzo, F. Esposito und D. Malerba (2000). *Transforming Paper Documents into XML Format with WISDOM++*. Technischer Bericht, Dipartimento di Informatica, Università degli Studi di Bari - Italy.

[Altamura et al. 2001] Altamura, Oronzo, F. Esposito und D. Malerba (2001). *Transforming paper documents into XML format with WISDOM++*. International Journal on Document Analysis and Recognition.

[Behme und Mintert 2000] Behme, Henning und S. Mintert (2000). *XML in der Praxis*. Addison-Wesley.

[Born 1995] Born, Günter (1995). *Noch mehr Dateiformate: Neue Dateiformate für Grafik, Text, Tabellenkalkulation und Sound*. Addison-Wesley, 1. Aufl.

[Born 2001] Born, Günter (2001). *Dateiformate - Die Referenz*. Galileo Press GmbH, Bonn, 1 Aufl.

[Bryan 1988] Bryan, Martin (1988). *SGML - an author's guide to the Standard Generalized Markup Language*. Addison-Wesley.

[Dahn 2001a] Dahn, Ingo (2001a). *Slicing Book Technology - Providing Online Support for Textbooks*. In: *ICDE 2001*.

[Dahn 2001b] Dahn, Ingo (2001b). *Using Networks for Advanced Personalization of Documents*. In: *SSGRR 2001 International Conference on Advances in Infrastructure for Electronic Buisness, Science and Education on the Internet*, L'Aqu. Scuola Superiore Guglielmo Reiss Romoli.

[Dahn et al.] Dahn, Ingo, M. Armbruster, U. Furbach und G. Schwabe. *Slicing Books - The Authors' Perspective*. `http://www.citeseer.nj.nec.com/487947.html`.

[Esposito et al. 2000] Esposito, Floriana, D. Malerba und F. A. Lisi (2000). *Machine Learning for Intelligent Processing of Printed Documents*. Journal of Intelligent Information Systems, 14(2/3):175–198.

[Esposito et al. 1995] Esposito, Floriana, D. Malerba und G. Semeraro (1995). *A Knowledge-Based Approach to the Layout Analysis*. Proc. of the 3rd Int. Conf. on the Practical Application of Prolog, S. 429–443.

[Goldfarb 1996] Goldfarb, Charles F. (1996). *The Roots of SGML – A Personal Recollection*.

[Goossens et al. 2000] Goossens, Michel, F. Mittelbach und A. Samarin (2000). *Der LaTeX Begleiter*. Addision-Wesley.

[Hyafil und Rivest 1976] Hyafil, Laurent und R. L. Rivest (1976). *Constructing optimal binary decision trees is NP-complete*. In: Glushkov, V.M., D. Gries, D. Knuth, M. Paul, W. Turski und W. Van der Poel, Hrsg.: *Information Processing Letters*, Bd. 5, S. 15–17. North-Holland Publishing Company.

[Knappen et al. 1995] Knappen, Jörg, H. Partl, E. Schlegl und I. Hyna (1995). *LaTeX2ε-Kurzbeschreibung*. Zentrum für Datenverarbeitung, Johannes Gutenberg-Universität Mainz.

[Kobert 1999] Kobert, Thomas (1999). *XML*. bhv Verlags GmbH.

[Lobin 2001] Lobin, Henning (2001). *Informationsmodellierung in XML und SGML*. Springer-Verlag.

[Mintert 1999] Mintert, Stefan (1999). *Automatisierte WWW-Veröffentlichung auf der Basis formaler Auszeichnungssprachen*. Diplomarbeit, Universität Dortmund, Lehrstuhl für Künstliche Intelligenz.

[Mintert 2002] Mintert, Stefan (2002). *XML & Co*. Addison-Wesley.

[Mitchell 1997] Mitchell, Tom M. (1997). *Machine Learning*. The McGraw-Hill Companies, Inc.

[Morik 1995] Morik, Katharina (1995). *Einführung in die Künstliche Intelligenz*, Kap. 3, S. 243–297. Addison-Wesley, 2 Aufl.

[Morik 1997] Morik, Katharina (1997). *Einführung in die Künstliche Intelligenz*. Vorlesungsskript. `http://www-ai.cs.uni-dortmund.de/LEHRE/VORLESUNGEN/KI/SKRIPT/skript97_0.pdf`.

[Morik 1998] Morik, Katharina (1998). *Programmierung I: JAVA Skript zur Vorlesung WS 98/99*. Vorlesungsskript. `http://www-ai.cs.uni-dortmund.de/DOKUMENTE/morik_2000b.ps.gz`.

[Morik 2002] Morik, Katharina (2002). *Maschinelles Lernen und Data Mining*. Begleitfolien zur Vorlesung. `http://www-ai.cs.uni-dortmund.de/LEHRE/VORLESUNGEN/MLRN/FOLIEN/FolienSuche.pdf`.

[Quinlan 1983] Quinlan, John Ross (1983). *Learning Efficient Classification Procedures And Their Application To Chess End Games*. In: Michalski, Ryszard S., J. G. Carbonell und T. M. Mitchell, Hrsg.: *Machine Learning - An Artifical Intelligence Approach*, S. 463–482. Tioga Publishing Company.

[QUINLAN 1993] QUINLAN, JOHN ROSS (1993). *C4.5 programs for machine learning*. Morgan Kaufmann Publishers, Inc.

[VALERIUS et al. 2001] VALERIUS, MARIANNE, I. DAHN und G. SCHWABE (2001). *Adaptive Bücher für das kooperative Lernen: Anwendungen - Konzepte - Erfahrungen*. In: M. ENGELIEN, J. HOMANN, Hrsg.: *Virtuelle Organisation und Neue Medien 2001*, S. 391–413. Josef Eul Verlag.

[WANG et al. 1999] WANG, Y., T. PHILLIPS und R. HARALICK (1999). *From Image to SGML/XML Representation: One Method*. Proc. of the Int. Workshop on Document Layout Interpretation and its Applications.

[WITTEN und FRANK 2000] WITTEN, IAN H. und E. FRANK (2000). *Data Mining - Practical Machine Learning Tools and Techniques with Java Implementations*. Morgan Kauffmann Publishers, San Francisco.

[WONG et al. 1982] WONG, KWAN Y., R. G. CASEY und F. M. WAHL (1982). *Document analysis system*. IBM Journal of Research and Development, 26(6):647–656.