Studienarbeit

Interaktives Erkennen von Textstrukturen durch Maschinelles Lernen

von

Sebastian Gerke

eingereicht am 19. August 2005 beim
Institut für Angewandte Informatik
und Formale Beschreibungsverfahren
der Universität Karlsruhe

Referent: Prof. Dr. Rudi Studer

Betreuer: Dipl.-Inform. Max Völkel

Heimatanschrift:
Birkenweg 11
55413 Weiler

Studienanschrift:
Hohenzollernstr. 7
76135 Karlsruhe

Interaktives Erkennen von Textstrukturen durch Maschinelles Lernen

Halbautomatische Strukturextraktion

aus strukturierten und semi-strukturierten Texten

Studienarbeit

Institut für
Angewandte Informatik und Formale Beschreibungsverfahren
Lehrstuhl Studer
Fakultät für Wirtschaftswissenschaften
Universität Karlsruhe (TH)

cand. inform. Sebastian Gerke

Betreuer:
Dipl.-Inform. Max Völkel

verantwortlicher Betreuer:
Prof. Dr. Rudi Studer

Tag der Anmeldung: 21. Januar 2005
Tag der Abgabe: 19. August 2005

Inhaltsverzeichnis

1 Motivation

Angesichts der heutigen Flut an textuellen Informationen (z. B. E-Mails) wird es immer schwieriger, diese Informationen gewinnbringend einzusetzen. Das größte Problem ist dabei jedoch nicht die reine Menge der Informationen, sondern die große Vielfalt an Datenformaten, in denen diese vorliegen. Selbst Textdaten, die zur maschinellen Verarbeitung bestimmt sind, liegen häufig in proprietären Formaten und Formatierungen vor. Dadurch wird eine Verwendung dieser Daten in anderen als den ursprünglich angedachten Systemen erschwert, obwohl die Struktur dieser Daten für Menschen relativ leicht zu erkennen ist.

Daher müssen für verschiedene Datenformate entweder Programme entwickelt werden, die eine Konvertierung eines gegebenen Datenformats in ein für die jeweilige Anwendung benötigtes Datenformat durchführen oder einen Parser für das vorliegende Dateiformat in das Programm integrieren, so dass es direkt auf diese zugreifen kann. Beide Optionen laufen letztendlich auf die Entwicklung eines Parsers hinaus, da ein Konvertierungsprogramm zunächst die Eingabedaten einlesen muss, um sie anschließend in einem anderen Format abspeichern zu können. Anwender, die keine Erfahrung in der Software-Entwicklung haben, müssen dann entweder die Entwicklung einer Konvertierungsroutine veranlassen oder die Konvertierung (insbesondere bei einmaligen Datenübernahmen) manuell durchführen. Beide Optionen hindern den Anwender daran, die gewünschten Daten unmittelbar in der gewünschten Anwendung zu nutzen. Für erfahrene Benutzer und Entwickler ist das Erstellen eines Parsers meist eine lästige, aber zeitintensive Pflichtaufgabe. Auch für diese Benutzergruppe wären vereinfachende Möglichkeiten, einen Parser für solche Daten zu erstellen, hilfreich.

Die vorliegende Arbeit stellt ein Software-System zur einfachen Erstellung von Parsern vor, bei dem die Parser anhand von konkreten Beispielen generiert werden. Zur Nutzung des Systems ist keine Kenntnis bzgl. der Erstellung von Parsern nötig.

1.1 Beispiele

Im Folgenden werden einige Beispiele gegeben, die die Problematik der wachsenden Formatvielfalt verdeutlichen sollen.

Wikis Ein Beispiel für die wachsende Anzahl verschiedener Datenformate sind die zunehmend beliebter werdenden Wikis. Wikis sind Internet-Seiten, die nicht

nur von ihren Besuchern gelesen werden können, sondern auch von Ihnen verändert werden können. Der Name stammt von *wikiwiki*, dem hawaiianischen Wort für „schnell“.

Die Eingabe des Inhalts einer Wiki-Seite erfolgt jedoch nicht in HTML, sondern in einem Format mit relativ intuitiven Auszeichnungen für Textformatierungen, wie sie z. B. häufig in E-Mails anzutreffen sind. Dadurch wird die Datenerfassung deutlich vereinfacht, da nicht umständliche HTML-Tags getippt werden müssen, sondern kürzere Markierungen verwendet werden, die eine schnellere Dateneingabe möglich machen.

So werden etwa Überschriften in einem Wiki-Text durch `== Überschrift ==` gekennzeichnet, während eine Überschrift in einem HTML-Dokument durch `<h1>Überschrift</h1>` ausgezeichnet wird. Noch deutlicher wird die sehr einfache Syntax von Wikis bei Listen, wie in Abbildung 1.1 zu sehen ist.

Zu beachten ist jedoch, dass die oben stehenden Beispiele nicht für alle Wiki-Systeme gelten. Die Syntax von Wikis unterscheidet sich von System zu System. Diese Tatsache stellt das größte Problem für Benutzer mehrerer Wikis dar. Sie müssen teilweise für jede Wiki-Seite, die sie editieren, eine andere Syntax erlernen. Gerade dadurch, dass sich die Wiki-Dialekte zum Teil nur in wenigen Details unterscheiden, ist die Verwechslungsgefahr besonders groß. So werden z. B. Abschnitte mit (Top-Level-)Überschriften in verschiedenen Wiki-Systemen wie in Abbildung 1.3 gekennzeichnet.

Anhand der Beispiele in Abbildung 1.3 wird deutlich, wie sich die einzelnen Wiki-Systeme unterscheiden. Teilweise sind sich die einzelnen Dialekte sehr ähnlich, wie im o.a. Beispiel MediaWiki und MoinMoin, während sich einige Dialekte

```
* Listitem 1
* Listitem 2
* Listitem 3
```

Abbildung 1.1: Eine Liste in Wiki-Syntax

```
<ul>
 <li>Listitem 1</li>
 <li>Listitem 2</li>
 <li>Listitem 3</li>
</ul>
```

Abbildung 1.2: Eine Liste in HTML-Syntax

MediaWiki	`== Header ==`
MoinMoin	`= Header =`
Twiki	`---+ Header`
SnipSnap	`1 Header`

Abbildung 1.3: Auszeichnung von Top-Level-Überschriften in verschiedenen Wikis

stärker von anderen unterscheiden, wie das im obigen Beispiel bei SnipSnap der Fall ist.

CSV-Datei Eine CSV-Datei („Character Separated Values“ oder „Comma Separated Values“) ist eine Textdatei, die tabellarisch strukturierte Daten enthält. Dabei stellt eine Zeile in der CSV-Datei einen Datensatz dar (Allerdings können einzelne Einträge auch Zeilenumbrüche enthalten. In diesem Fall muss dieser Eintrag von doppelten Anführungszeichen (") umschlossen sein). Die einzelnen Felder eines Datensatzes werden durch ein Trennzeichen voneinander abgegrenzt. Dieses Beispiel ist hier stellvertretend für viele ähnliche Datenformate, deren Datensätze durch Trennzeichen getrennt werden, aufgeführt. Ein weiteres Beispiel, welches den CSV-Dateien ähnelt, ist z. B. eine Winamp-Playlist-Datei.

Das CSV-Format hat sich als Quasi-Standard durchgesetzt. Es wird häufig benutzt, um Daten zwischen Computerprogrammen auszutauschen. In letzter Zeit setzt sich allerdings immer mehr XML als Datenaustauschformat durch. In Abbildung 1.4 ist ein Beispiel für eine CSV-Datei angegeben, eine mögliche, äquivalente XML-Ausgabe ist in Abbildung 1.5 zu sehen.

1.2 Aufgabenstellung

Für den Benutzer wäre es in der Regel wesentlich angenehmer, wenn er den Parser/Konverter nicht von Hand schreiben müsste, sondern einfach Beispiele geben könnte, wie seine Daten aussehen. So könnte er zum Beispiel in einem Wiki-Dokument eine Überschrift markieren und die Software versucht anschließend, anhand des Beispiels die Textformatierungsvorschrift für Überschriften zu erkennen. Anschließend soll das Dokument in eine XML-Datei überführt werden, um eine einfache Weiterverarbeitung möglich zu machen. Diese XML-Datei repräsentiert dann die Struktur des ursprünglichen Dokuments. Zusätzlich werden alle Textbereiche, die nicht durch den Benutzer markiert wurden, in das XML-Dokument aufgenommen, um nicht versehentlich Informationen zu verlieren und das ursprüngliche Dokument wieder rekonstruieren zu können. Sollten nicht alle Informationen benötigt werden, wie z. B. die Semikola aus der CSV-Datei, können ungewünschte Teile des XML-Dokuments bei Bedarf durch XSL-Transformationen aus diesem herausgefiltert werden.

In Kapitel 2.2 werden Arbeiten aus dem Umfeld des automatischen Erstellens von Parsern und der Informationsextraktion vorgestellt. Insbesondere wird in diesem Kapitel auf den Texteditor LAPIS eingegangen, da dieser einige Funktionen aufweist, die einem Teil der Anforderungen an die vorliegende Arbeit ähneln.

```
Cafe;3,3;0,2;100
Espresso;2;0,05;200
Cappuchino;3;0,2;185
Tee;2,3;0,33;213
Wasser;1,8;0,5;312
```

Abbildung 1.4: CSV-Datei

```
<products>
  <product>
    <name>Cafe</name>
    <price>3,3</price>
    <amount>0,2</amount>
    <sold>100</sold>
  </product>
  <product>
    <name>Espresso</name>
    <price>2</price>
    <amount>0,05</amount>
    <sold>200</sold>
  </product>
    <product>
    <name>Cappuchino</name>
    <price>3</price>
    <amount>0,2</amount>
    <sold>185</sold>
  </product>
  <product>
    <name>Tee</name>
    <price>2,3</price>
    <amount>0,33</amount>
    <sold>213</sold>
  </product>
  <product>
    <name>Wasser</name>
    <price>1,8</price>
    <amount>0,5</amount>
    <sold>312</sold>
  </product>
</products>
```

Abbildung 1.5: gewünschte Ausgabe

2 Verwandte Arbeiten

In diesem Kapitel wird zunächst der Texteditor und Webbrowser *LAPIS* beschrieben, da dieser einige Funktionen enthält, die in der vorliegenden Arbeit zumindest konzeptionell wieder verwendet werden, bzw. als Basis für eigene Verbesserungen dienen. Da die Zielsetzung von *LAPIS* sich von der der vorliegenden Arbeit unterscheidet, werden zum Teil auch Vereinfachungen getroffen.

2.1 LAPIS

LAPIS [Mil02] ist ein Texteditor und Webbrowser, der einige erweiterte Funktionen besitzt, die über den Funktionsumfang üblicher Texteditoren hinausgehen. So können Textmuster durch Angabe von Beispielen induziert werden. Diese können dann verwendet werden, um eine Selektion eines Textbereichs auf mehrere Bereiche gleichen Typs auszuweiten. Anhand dieser Selektion können dann Änderungen an mehreren Stellen im Text gleichzeitig vorgenommen werden. So können z. B. alle URLs in einem HTML-Dokument durch Einbetten in ein `<b>`-Tag hervorgehoben werden.

Die Pattern-Beschreibungssprache, die von LAPIS genutzt wird, nennt sich *Text Constraints* und basiert auf der sog. *Region Algebra*. Eine Region ist dabei zusammenhängender Textbereich. Viele Textverarbeitungssysteme operieren auf Regionen. So werden Textformatierungen wie Schrift, Schriftfarbe, kursive Schrift usw. mit Textregionen verknüpft. Das „Lightweight structure model“ basiert auf Mengen von Regionen (Region Sets), da Strukturabstraktionen wie z. B. Kapitel Mengen von Regionen beschreiben. Im Folgenden wird eine Algebra beschrieben, mit der Region Sets kombiniert werden können, Diese Algebra bildet die Basis für die Implementierung der *Text Constraints*.

Folgende Anwendungsfälle definieren einige Anforderungen an die *Region Algebra*:

- Regionen können in anderen Regionen enthalten sein. So sind z. B. in Texten Wörter komplett in einem Satz enthalten, Sätze in einem Abschnitt und Abschnitte in Kapiteln.
- Regionen können sich überlappen. So können sich z. B. in einem Word-Dokument fett gedruckte Regionen mit kursiven Regionen überlappen.
- Regionen können auch die Länge 0 haben. So wird z. B. in einem Texteditor jede Selektion des Nutzers durch eine Region repräsentiert, also auch der

Cursor *zwischen* zwei Zeichen.

Da es also viele Möglichkeiten gibt, Text zu strukturieren, sei es inhaltlich oder optisch, sollte die *Region Algebra* beliebige *Region Sets* zulassen. Jedoch ist die Anzahl aller möglichen *Region Sets* quadratisch in Bezug auf die Länge der Zeichenkette. Dies führt zu Problemen bei der effizienten Implementierung dieser Algebra. Andere Systeme haben dieses Problem umgangen, indem sie *Region Set*s auf festgelegte Typen beschränkt haben. Dadurch wurde dann aber die Ausdrucksmächtigkeit geopfert, um die Verarbeitung in linearer Zeit durchführen zu können.

Die verwendete *Region Algebra* kann beliebige *Region Set*s kombinieren. Dies hat mehrere Vorteile: Es wird nur eine handvoll einfacher Operatoren benötigt, die Standard-Mengen-Operatoren und ein `iteration`-Operator. Andere Systeme benötigen weitaus mehr Operatoren. Des weiteren müssen keine Restriktionen der beiden zu verknüpfenden *Region Sets* beachtet werden. Diese Eigenschaft ist zum einen nützlich für den Benutzer, da es einfacher ist, Pattern zu schreiben, ohne auf spezielle Bedingungen (außer der syntaktischen Korrektheit) zu achten. Außerdem vereinfacht es die Generierung von Features und Hypothesen durch Machine-Learning-Algorithmen.

2.1.1 Regionen

Regionen innerhalb einer Zeichenkette der Länge n werden durch ein Paar $[s, e]$ beschrieben, wobei $0 \leq s \leq e \leq n$ gilt. Der Start-Offset s gibt dabei die Anzahl der Zeichen an, die vor dem Startpunkt der Region liegen. Der End-Offset e gibt analog dazu die Anzahl der Zeichen an, die vor dem Endpunkt der Region liegen.

2.1.2 Relationen zwischen Regionen

Die Algebra basiert auf drei grundlegenden binären Relationen zwischen Regionen:

$$[s,e]\ before\ [s',e'] \quad gdw.\ e \leq s'\ und\ s < s' \tag{2.1}$$

$$[s,e]\ overlaps-start\ [s',e'] \quad gdw.\ s \leq s'\ und\ e \leq e' \tag{2.2}$$

$$[s,e]\ contains\ [s',e'] \quad gdw.\ s \leq s'\ und\ e' \leq e \tag{2.3}$$

Die Relationen $overlaps - start$ und $contains$ sind reflexiv, $before$ jedoch nicht. $before$ und $contains$ sind transitiv, $overlaps - start$ jedoch nicht. Keine der Relationen ist symmetrisch. Daher werden deren Inversen folgendermaßen genannt:

- $[s,e]\ after\ [s',e']$ gdw. $[s',e']\ before\ [s,e]$
- $[s,e]\ overlaps-end\ [s',e']$ gdw.$[s',e']\ overlaps-start\ [s,e]$

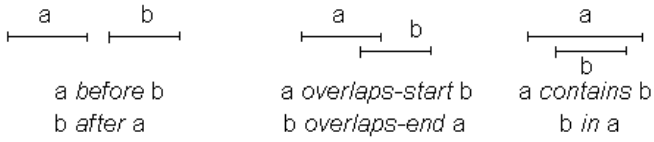


Abbildung 2.1: Grundlegende Relationen zwischen Regionen

- $[s, e]$ *in* $[s', e']$ gdw. $[s', e']$ *contains* $[s, e]$

In Abbildung 2.1 sind alle sechs Relationen zur Veranschaulichung graphisch dargestellt.

Satz Für zwei beliebige Regionen $[s, e]$ und $[s', e']$ existiert mindestens eine Relation *op*, so dass $[s, e]$ *op* $[s', e']$ gilt.

Auf den Beweis dieses Satzes wird an dieser Stelle verzichtet, er ist jedoch unter [Mil02] nachzulesen. Diese Relationen können nun genutzt werden, um einige wichtige Klassen von *Region Set*s zu definieren. Sei A ein *Region Set*, dann gilt:

- A ist **flach** gdw. für alle $a, b \in A$ gilt: mind. eine der folgenden Relationen gilt: $a\ before\ b$; $a\ after\ b$ oder $a = b$.
- A ist **verschachtelt** gdw. für alle $a, b \in A$ gilt: mind. eine der folgenden Relationen gilt: $a\ before\ b$; $a\ after\ b$; $a\ in\ b$; $a\ contains\ b$ oder $a = b$.
- A ist **überlappend** gdw. für alle $a, b \in A$ gilt: mind. eine der folgenden Relationen gilt: $a\ before\ b$; $a\ after\ b$; $a\ overlaps-start\ b$; $a\ overlaps-end\ b$ oder $a = b$.

Flache, verschachtelte und überlappende *Region Set*s sind so wichtig, weil die maximal mögliche Anzahl der darin enthaltenen Regionen linear abhängig von der Länge der Zeichenkette ist:

- A ist überlappend $\rightarrow |A| \leq 2n + 1$
- A ist flach $\rightarrow |A| \leq 2n + 1$
- A ist verschachtelt $\rightarrow |A| \leq 3n$

2.1.3 Region Algebra

Dieser Abschnitt gibt einen kurzen Überblick über die Region Algebra. Die Algebra besteht aus einer Menge von Operatoren, denen null oder mehr *Region Set*s als Argument übergeben werden und liefern ein *Region Set* als Resultat. Die Algebra benutzt zum einen die bekannten Menge-Operatoren für den Schnitt, die

Vereinigung und die Differenz:

$$A \cap B \equiv \{r | r \in A \wedge r \in B\} \tag{2.4}$$
$$A \cup B \equiv \{r | r \in A \vee r \in B\} \tag{2.5}$$
$$A - B \equiv \{r | r \in A \wedge r \notin B\} \tag{2.6}$$

Basierend auf diesen grundlegenden Operatoren werden nun einige höherstufige Pattern Matching Operatoren definiert, die im Folgenden aufgeführt werden. Auf eine nähere Erläuterung dieser Operatoren wird an dieser Stelle verzichtet.

2.1.4 Text Constraints

Die Textmuster-Beschreibungssprache, die in LAPIS benutzt wird, nennt sich *Text Constraints* (TC). Sie setzt auf der vorgestellten *Region Algebra* auf. Im Folgenden wird lediglich anhand einiger Beispiele die Syntax der *Text Constraints* erläutert. Eine vollständige Spezifikation der *Text Constraints* findet man in [Mil02].

```
from "["
  to "]]"
  starting "[["
```

Beispiel für ein Text Constraint: Dieses Textmuster beschreibt Textabschnitte, die mit `"[["` beginnen und mit `"]]"` enden (Wiki-Link)

```
from start of Paragraph
  to CapitalizedWord
  in CapitalizedWord
```

Diese Textmuster beschreibt das erste Wort in einem Abschnitt.

```
from end of 2nd ";" in Token
  to start of 3rd ";" in Token
  in Token
```

Dieses Textmuster beschreibt die dritte Spalte einer CSV-Datei.

2.1.5 Induktion von Text Constraints

Dieser Abschnitt beschreibt, wie in LAPIS Selektionen „erraten“ werden. D.h., wie durch positive und negative Beispiele die dazu passenden *Text Constraints* induziert werden.

Benutzeroberfläche

Für das sog. „Selection Guessing“ muss der Benutzer zunächst in den „Guessing Mode“ wechseln, um dem System mitzuteilen, dass es versuchen soll, anhand einer Selektion weitere zu erraten. Wenn der Benutzer nun den „Guessing Mode“ aktiviert hat, kann er einen gewünschten Textbereich markieren. Nun generiert das System einige *Text Constraints*, die alle den vom Nutzer markierten Bereich abdecken.

Diese Hypothesen werden vom System anhand einer Heuristik nach der geschätzten Wahrscheinlichkeit ihres Auftretens sortiert. Die Textregionen der Hypothese, deren Auftretenswahrscheinlichkeit am höchsten ist, werden im Text farblich hervorgehoben. Der Benutzer kann nun weitere Positiv- und Negativbeispiele, die vom System erraten wurden, durch Anklicken entfernen, so dass die Hypothesen diese Region nicht mehr matchen. Wurden die gewünschten Regionen selektiert, kann der Benutzer diese Selektionen gleichzeitig editieren. So können z. B. in Java-Quellcode die Sichtbarkeit aller Variablen einer Klasse von `public` auf `private` geändert werden. In diesem Zusammenhang ist noch zu erwähnen, dass die *Text Constraints* zusätzlich zu den reinen zeichenbasierten Mustern noch eine Bibliothek an vorgefertigten Mustern haben, wie z. B. `Java.Class`, `Java.Method`, `URL`, `Email-Adress` usw.

Der Algorithmus

Die Eingabedaten, die dem Algorithmus zur Verfügung stehen, sind das zugrunde liegende Dokument, eine Menge von Positivbeispielen in Form eines *Region Set*s und einer Menge von Negativbeispielen, ebenfalls in Form eines *Region Set*s. Die Ausgabe ist eine geordnete Liste von Hypothesen, die jeweils aus einem TC-Pattern bestehen, welches die Positivbeispiele beschreibt und die Negativbeispiele nicht enthält. Ein Feature besteht in dem Algorithmus aus einem TC-Pattern der Form `op F`, wobei `op` einer der folgenden TC-Operatoren ist: `equals`, `just_before`, `just_after`, `starting`, `ending`, `in` oder `contains`. `F` ist entweder ein Literal oder ein bestehendes Pattern aus der Bibliothek vorgefertigter Pattern. Eine Hypothese besteht nun aus einer boolschen Verknüpfung von Features, wie z. B. `just_after "--" AND just_before "--"`.
Der Algorithmus besteht aus folgenden drei Teilen:

1. Generierung von Features

2. Generierung von Hypothesen

3. Ordnen der Hypothesen

Generierung von Features Die Eingabe dieses Teils des Algorithmus besteht aus dem Dokument und den Positivbeispielen. Er generiert zwei Arten von Features: „library features“, die auf der Pattern-Bibliothek basieren, und „literal features“, die durch Untersuchung des Textes gewonnen werden. Die „library features“ werden dadurch erzeugt, indem jedes Pattern aus Bibliothek auf den Text angewandt und überprüft wird, ob alle Positivbeispiele in dem resultierenden *Region Set* enthalten sind. Die „library features“ werden durch Kombination der relationalen Operatoren mit Zeichenketten generiert. Diese Zeichenketten werden durch Untersuchen der Positivbeispiele gewonnen. In der folgenden Beschreibung besteht das i-te Positivbeispiel aus der Region $x_i[y_i]z_i$, wobei das gesamte Dokument die Zeichenkette $x_iy_iz_i$ ist, und y_i der Teil des Textes, der durch das i-te Positivbeispiel abgedeckt ist.x_i ist dementsprechend also der Teil des Textes vor dem i-ten Positivbeispiel, y_i der Teil des gesamten Textes nach dem i-ten Positivbeispiel ($i = 1..n$, n ist die Anzahl der vom Benutzer angegebenen Positivbeispiele).

- `equals`: Wenn alle y identisch sind, wird das Feature `equals y` generiert.
- `starting`: Durch Finden eines gemeinsamen Präfix von allen y, wobei bei zwei äquivalenten Präfixen nur der kürzere beibehalten wird. Wenn z. B. im Text jedes Vorkommen von „http“ von „://“ gefolgt wird, wird nur das Feature `starting "http"` beibehalten.
- `ending`: Generierung eines gemeinsamen Suffix von allen y, analog zu `starting`.
- `just_before`: Generierung eines gemeinsamen Präfix von allen z.
- `just_after`: Generierung eines gemeinsamen Suffix von allen x.
- `contains`: Generierung eines gemeinsamen Teilstrings mit Hilfe eines Suffix-Baumes.

Generierung von Hypothesen Nach der Generierung der Features werden von dem Algorithmus die Features konjunktiv verknüpft, um so Hypothesen zu erhalten, die konsistent zu allen Positivbeispielen sind. Da nicht alle Kombinationen von Features exakte Start- und Endpunkte definieren (so definiert z. B. `containing 'f'` alle Textregionen, die ein „f“ enthalten), werden nur solche Hypothesen (sog. *Kernhypothesen*) generiert, die exakte Start- und Endpunkte festlegen. Das kann zum Einen ein einzelnes Feature sein, das sowohl Start-, als auch den Endpunkt definiert, wie z. B. `equals F`. Zum anderen kann eine solche Hypothese aus einem Feature, welches den Startpunkt einer Region definiert (`starting` oder `just_after`), und aus einem Feature, das den Endpunkt bestimmt (`ending` oder `just_before`), bestehen.

Wenn Negativbeispiele vorhanden sind, werden diese Kernhypothesen durch zusätzliche Einschränkungen so spezialisiert, dass die Negativbeispiele nicht mehr durch diese Pattern abgedeckt werden. So kann z. B. aus einer Kernhypothese

```
Link
```

, welche alle Links in einem HTML-Dokument erkennt, eine endgültige Hypothese

```
Link contains "uka.de"
   just_before Linebreak
```

werden. Kernhypothesen, die nicht spezialisiert werden können, um alle Negativbeispiele auszuschließen, werden verworfen.

Ordnen der Hypothesen Nach dem Generieren der Hypothesen muss nun die beste ausgewählt werden. Lernalgorithmen nutzen meist *Occam's Razor*, welches die Hypothesen mit der kürzesten Beschreibung bevorzugt. Im vorliegenden Fall wäre dies also die Anzahl der benutzten Features. Hier wird *Occam's Razor* durch eine Heuristik erweitert, die solche Hypothesen bevorzugt, welche auf sog. regulären Features bestehen. Jedes Feature `contains F` ist regulär, wenn die Anzahl der Vorkommen von `F` in allen Positivbeispielen gleich ist. Analog dazu werden auch für die übrigen Features ähnliche Maße eingeführt, die bestimmte Hypothesen vorziehen.

Fazit LAPIS ist ein sehr interessantes Tool, insbesondere die Benutzerschnittstelle und der Lernalgorithmus waren für diese Arbeit.

2.2 Weitere Arbeiten

Es existieren folgende weitere Arbeiten aus dem Bereich der automatischen Parsergenerierung und Informationsextraktion, die jedoch ihren Fokus eher auf die Informationsextraktion aus Freitext legen. Darüber hinaus sind für die meisten Algorithmen größere Trainingsmengen nötig, die im vorliegenden Fall nicht vorhanden sind. Daher kommen die verwendeten Algorithmen für die vorliegende Arbeit nicht in Betracht. Diese sind:

- *LP^2, an Adaptive Algorithm for Information Extraction from Web-related Texts.* [Cir01]: In dieser Arbeit wird ein Verfahren zur Informationsextraktion aus Texten vorgestellt. Dazu wird in einem Lerndurchgang die gewünschte Information markiert. Anschließend werden Regeln zur Beschreibung der Markierungen vom Algorithmus gelernt. Die Regeln basieren dabei auf den Wörtern, die sich in der Nähe der gewünschten Information befinden.
- *Learning Information Extraction Rules for Semi-Structured and Free Text* [Sod99]: Hier wird ein Algorithmus zur Informationsextraktion aus semistrukturierten (z. B. Wohnungsanzeigen auf Internet-Seiten) und unstrukturierten Daten vorgestellt. Auch hier werden umfangreiche Trainingsdaten benötigt.

Außerdem wurden folgende Arbeiten untersucht, die jedoch auf der geometrischen Struktur des Dokuments basierten, was sich für die vorliegende Arbeit jedoch als nicht notwendig herausstellte:

- *Integrating Geometrical and Linguistic Analysis for E-Mail Signature Block Parsing* [CHS99]: Hier wird ein Verfahren vorgestellt, welches Adressdaten aus E-Mail-Signaturen extrahiert. Dieser Vorgang geschieht in zwei Schritten: Zunächst wird eine geometrische Analyse auf die E-Mail-Signatur angewandt, mit der z. B. mehrspaltige Layouts erkannt werden, anschließend folgt eine linguistische Analyse der Signatur.
- *Combining Visual layout and lexical cohesion features for text segmentation* [MY00]: In dieser Arbeit wird ein Verfahren zur automatischen Erstellung eines Dokumentstrukturbaums aus einem Textdokument vorgestellt. Dabei kann es sich sowohl um stark formatierte Dokumente wie z. B. PDF- oder Word-Dokumente, als auch um einfache Textdokumente handeln. Ein Dokumentstrukturbaum ist eine Datenstruktur zur Speicherung von Textdokumenten, welche z. B. in Kapitel, Absätze und Abschnitte unterteilt sind. Diese Struktur wird mit dem vorgestellten Verfahren automatisch erkannt. Es basiert hauptsächlich auf Layoutmerkmalen wie Schriftgröße, Zeilenabstände, Rändern, usw.

Des weiteren wurden folgende Arbeiten studiert, die Parser für natürliche Sprache mit Methoden des Maschinellen Lernens erstellen. Auch hier werden umfangreiche Trainingsdaten benötigt, daher konnten diese Ansätze nicht weiter verfolgt werden und sind hier nur kurz erwähnt.

- *Automatic Grammar Induction and Parsing Free Text: A Transformation-Based Approach* [Bri93]
- *Rapid Parser Development: A Machine Learning Approach for Korean* [Her00]
- *A Corpus-Based Learning Technique for Building A Self-Extensible Parser* [LS94]

3 Entwurf

Im Folgenden wird der Entwurf des Systems zur einfachen Erstellung von Parsern vorgestellt. Zunächst werden kurz die Anforderungen an das zu erstellende Software-System beschrieben. Anschließend werden die verwendeten Datenstrukturen vorgestellt und erläutert. Dabei werden die Begriffe „Textregion“ und „Textkonzept“ folgendermaßen verwendet:

Eine **Textregion** beschreibt einen zusammenhängenden Teilbereich eines Textes. Textregionen können entweder durch ihren Start-Index im Text und ihrer Länge, oder aber durch Start- und End-Index beschrieben werden.

Ein **Textkonzept** wird durch eine Menge von Textregionen repräsentiert. So gibt es z. B. im Wiki-Kontext die Textkonzepte `Header`, `Link` o.ä..

Dann werden die verwendeten Algorithmen zur Induktion von Parsern beschrieben. Schließlich wird die Benutzeroberfläche beschrieben und die genaue Benutzer-Interaktion dargelegt.

3.1 Idee

Es soll ein Software-System erstellt werden, dass es erlaubt, auf intuitive Weise einen Parser zu erstellen. Dazu soll der Benutzer Positiv- und Negativbeispiele für Textregionen eines zu erlernenden Textkonzepts dem System geben können. Das System soll dann anhand dieser Beispiele Hypothesen für Klassifikatoren erstellen, welche die Textregionen des gewünschten Textkonzepts klassifizieren. Darüber hinaus kann der Benutzer angeben, ob ein zu erlernendes Textkonzept in einem bereits bestehenden Textkonzept enthalten ist bzw. ein solches enthält. Diese Angabe ist jedoch optional, sie beschleunigt lediglich den Lernvorgang. Wenn das System alle gewünschten Textkonzepte gelernt hat, erstellt das System aus diesen Informationen automatisch ein XML-Dokument. Ein Beispiel für ein mögliches Eingabedokument sowie die gewünschte Ausgabe ist in den Abbildungen 1.4 und 1.5 zu finden.

3.2 Datenstrukturen und Dateiformate

3.2.1 Textregionen

Die Darstellung von Textkonzepten erfolgt durch Textregionen. Eine Textregion wird dabei durch einen Start-Index und einen End-Index beschrieben. Dabei gibt

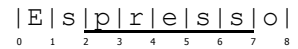


Abbildung 3.1: Darstellung einer Textregion

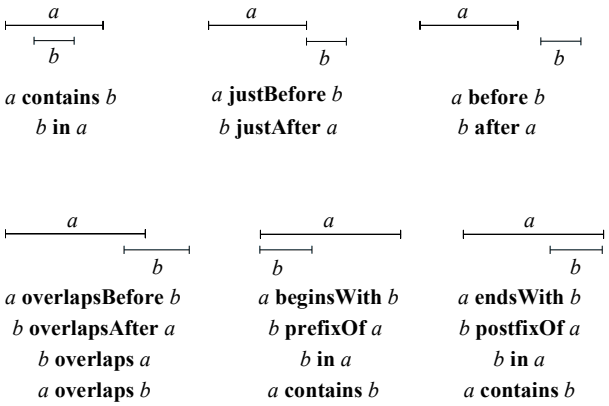


Abbildung 3.2: Relationen zwischen Textregionen

der Start-Index die Position des ersten Zeichens der Textregion an, der End-Index die Position des ersten Zeichens nach der Textregion, wobei das erste Zeichen im Text den Index 0 besitzt. Die Interpretation der beiden Indizes ist damit analog zu der Interpretation der `substring()`-Methode in Java und zu den meisten weiteren Implementierungen von Textregionen in Programmiersprachen, was die Implementierung und Benutzung von Textregionen vereinfacht. Textbereiche werden im Folgenden durch $[s, e]$ notiert, wobei s der Start-Index und e der End-Index ist. So wird z. B. in Abbildung 3.1 das Wort `Espresso` durch $[0, 8]$, die in der Abbildung unterstrichene Teilzeichenkette `press` durch $[2, 7]$ beschrieben. Die Stelle zwischen `s` und `p` in der Abbildung wird durch $[2, 2]$ beschrieben.

Relationen zwischen Textbereichen

Zwischen zwei Textbereichen gibt es einige Relationen, die ihre Lage im Text zueinander beschreiben. So kann sich ein Textbereich hinter einem weiteren Textbereich befinden, zwei Textbereiche können sich überlappen, ein Textbereich kann sich in einem anderen Textbereich befinden, usw.. Es existieren folgende Relatio-

nen zwischen Textbereichen:

$$
\begin{aligned}
[s,e]\,\textbf{equals}\,[s',e'] \quad & gdw. \quad s = s' \wedge e = e' && (3.1)\\
[s,e]\,\textbf{contains}\,[s',e'] \quad & gdw. \quad s \leq s' \wedge e \geq e' && (3.2)\\
[s,e]\,\textbf{in}\,[s',e'] \quad & gdw. \quad [s',e']\ \text{contains}\ [s,e] && (3.3)\\
[s,e]\,\textbf{beginsWith}\,[s',e'] \quad & gdw. \quad s = s' \wedge e \geq e' && (3.4)\\
[s,e]\,\textbf{prefixOf}\,[s',e'] \quad & gdw. \quad [s',e']\ \text{beginsWith}\ [s,e] && (3.5)\\
[s,e]\,\textbf{endsWith}\,[s',e'] \quad & gdw. \quad s \leq s' \wedge e = e' && (3.6)\\
[s,e]\,\textbf{postfixOf}\,[s',e'] \quad & gdw. \quad [s',e']\ \text{endsWith}\ [s,e] && (3.7)\\
[s,e]\,\textbf{justBefore}\,[s',e'] \quad & gdw. \quad e = s' && (3.8)\\
[s,e]\,\textbf{justAfter}\,[s',e'] \quad & gdw. \quad [s',e']\ \text{justBefore}\ [s,e] && (3.9)\\
[s,e]\,\textbf{before}\,[s',e'] \quad & gdw. \quad e \leq s' && (3.10)\\
[s,e]\,\textbf{after}\,[s',e'] \quad & gdw. \quad [s',e']\ \text{before}\ [s,e] && (3.11)\\
[s,e]\,\textbf{overlapsBefore}\,[s',e'] \quad & gdw. \quad s < s' \wedge e < e' \wedge e > s' && (3.12)\\
[s,e]\,\textbf{overlapsAfter}\,[s',e'] \quad & gdw. \quad [s',e']\ \text{overlapsBefore}\ [s,e] && (3.13)\\
[s,e]\,\textbf{overlaps}\,[s',e'] \quad & gdw. \quad [s',e']\ \text{overlapsBefore}\ [s,e] \vee && (3.14)\\
& \qquad\quad [s',e']\ \text{overlapsAfter}\ [s,e] &&\\
[s,e]\,\textbf{intersects}\,[s',e'] \quad & gdw. \quad [s,e]\ \text{overlaps}\ [s',e'] \vee && (3.15)\\
& \qquad\quad [s,e]\ \text{contains}\ [s',e'] \vee &&\\
& \qquad\quad [s',e']\ \text{contains}\ [s,e] &&
\end{aligned}
$$

3.2.2 Textklassifikatoren

Um bestimmte Strukturen im Text beschreiben zu können, wird ein Konzept benötigt, welches zu einer gegebenen Beschreibung die relevanten Textregionen markiert. Im vorliegenden Fall sind dies die Klassifikatoren. Ein Klassifikator erhält als Eingabe den gesamten zu klassifizierenden Text und gibt eine Menge von Textregionen zurück, die das von dem Klassifikator beschriebene Textkonzept im Text markieren. Es gibt zwei Arten von Klassifikatoren: Zum einen gibt es die Basisklassifikatoren, die nur auf Zeichenketten oder numerischen Parametern beruhen. Zum anderen gibt es die erweiterten Klassifikatoren, die auf bestehenden Klassifikatoren basieren. Beispiele für solche erweiterten Klassifikatoren sind in den Abbildung 3.3 und 3.4 zu finden. Es existieren folgende Klassifikatoren, die zur Beschreibung von Textkonzepten nötig sind:

Basisklassifikatoren

String-Klassifikator Klassifikator, der alle Vorkommen eines Strings erkennt.

RegExp-Klassifikator Klassifikator, der alle Textregionen erkennt, die auf einen beliebigen regulären Ausdruck passen.

BeginOfLine-Klassifikator Klassifikator, der alle Zeilenanfänge erkennt.

EndOfLine-Klassifikator Klassifikator, der alle Zeilenenden erkennt.

erweiterte Klassifikatoren Die ersten vier erweiterten Klassifikatoren sind Hilfsklassifikatoren, die die formale Definition weiterer Klassifikatoren vereinfachen. Für die Textregionen r, r_1, r_2, r_p gilt jeweils $r_i = [s_i, e_i], i \in \{1, 2, p\}$

Max-Klassifikator Erkennt alle maximalen Textregionen des übergebenen Klassifikators, d.h. alle Textregionen, die nicht in einer anderen Textregion enthalten sind.

$$C_{max}(C_p) = \{r | r \in C_p : \neg\exists r_2 \in C_p : (r_2 \textbf{ contains } r)\}$$

Min-Klassifikator Erkennt alle minimalen Textregionen des übergebenen Klassifikators, d.h. alle Textregionen, die keine andere Textregion enthalten.

$$C_{min}(C_p) = \{r | r \in C_p : \neg\exists r_2 \in C_p : (r_2 \textbf{ in } r)\}$$

Head-Klassifikator Erkennt die erste Textregion des übergebenen Klassifikators.

$$C_{head}(C_p) = \{r | r \in C_p : \forall r_2 \in C_p, r \neq r_2 : (s \leq s_2 \vee (s = s_2 \wedge e < e_2)\}$$

Tail-Klassifikator Erkennt die letzte Textregion des übergebenen Klassifikators.

$$C_{tail}(C_p) = \{r | r \in C_p : \forall r_2 \in C_p, r \neq r_2 : (e \geq e_2 \vee (e = e_2 \wedge s > s_2)\}$$

Inverter-Klassifikator Klassifikator C_{inv}, der alle maximalen Textregionen R erkennt, die nicht vom übergebenen Klassifikator C_p klassifiziert wurden. Es gilt also (wobei $\mathcal{P}$ die Menge aller möglichen Textregionen im Text ist):

$$C_{inv}(C_p) = C_{max}\left(\{r | r \in \mathcal{P} : \neg\exists r_p \in C_p : (r \textbf{ intersects } r_p)\}\right)$$

Alternating-Klassifikator Klassifikator C_{sec}, der nur jede zweite Textregion des übergebenen Klassifikator C_p erkennt. Dabei sind die Textregionen des übergebenen Klassifikators aufsteigend nach ihrem Start-Index sortiert.

$$C_{sec}(C_p) = \{r_i | r_i \in C_p, i \bmod 2 = 0\}$$

Set-Klassifikator Erkennt direkt aufeinander folgende Textregionen eines Klassifikators C_p als eine Textregion.

EndOf-Klassifikator Erkennt die Enden aller Textregionen des übergebenen Klassifikators. Die Textregionen dieses Klassifikators haben also alle die Länge 0.

$$C_{end}(C_p) = \{[e_p, e_p] \,|\, [s_p, e_p] \in C_p\}$$

StartOf-Klassifikator Erkennt die Anfänge aller Textregionen des übergebenen Klassifikators. Die Textregionen dieses Klassifikators haben die Länge 0.

$$C_{start}(C_p) = \{[s_p, s_p] \,|\, [s_p, e_p] \in C_p\}$$

And-Klassifikator Erkennt Textregionen, die von den beiden übergebenen Klassifikatoren abgedeckt werden.

$$C_{and}(C_p) = C_{max}(\{r | r \in \mathcal{P} : \exists r_1 \in C_1, r_2 \in C_2 : (r \textbf{ in } r_1) \wedge (r \textbf{ in } r_2)\})$$

Or-Klassifikator Erkennt Textregionen, die von einem der beiden übergebenen Klassifikatoren abgedeckt werden.

$$\begin{aligned} C_{or}(C_p) = C_{max}(\{r | r \in \mathcal{P} : \exists r_1 \in C_1, r_2 \in C_2 : (r \textbf{ in } r_1) \vee (r \textbf{ in } r_2) \vee \\ (s = \min(s_1, s_2) \wedge e = \max(e_1, e_2) \wedge (r_1 \textbf{ in } r_2))\}) \end{aligned}$$

FromTo-Klassifikator Erkennt Textregionen, die mit einer Textregion des ersten übergebenen Klassifikators beginnen und mit der nächsten im Text vorkommenden Textregion des zweiten übergebenen Klassifikators enden.

$$C_{fromTo}(C_1, C_2) = \{[s_1, e_2] \,| r_1 \in C_1, r_2 \in C_{head}(C_{after}(C_2, \{[s_1, e_1]\}))\}$$

Sequence-Klassifikator Erkennt alle Sequenzen von Textregionen der übergebenen Klassifikatoren. D.h. die Textregionen der Klassifikatoren müssen alle direkt hintereinander liegen, um von diesem Klassifikator erkannt werden.

$$C_{sequence}(C_1, C_2) = \{[s_1, e_2] \,| r_1 \in C_1, r_2 \in C_2 : (r_1 \textbf{ justBefore } r_2)\}$$

$$C_{sequence}(C_1, \ldots, C_n) = C_{sequence}(C_{sequence}(C_1, C_2), \ldots, C_n), n \geq 3$$

After-Klassifikatoren Erkennt alle Textregionen des ersten übergebenen Klassifikators, die hinter einer Textregion des zweiten Klassifikators stehen.

$$C_{after}(C_1, C_2) = \{r | r \in R_1 : \exists r_2 \in R_2 : (r \textbf{ after } r_2)\}$$

JustAfter-Klassifikator Erkennt alle Textregionen des ersten übergebenen Klassifikators, die direkt hinter einer Textregion des zweiten Klassifikators stehen.

$$C_{justAfter}(C_1, C_2) = \{r | r \in R_1 : \exists r_2 \in R_2 : (r \textbf{ justAfter } r_2)\}$$

Before-Klassifikatoren Erkennt alle Textregionen des ersten übergebenen Klassifikators, die vor einer Textregion des zweiten Klassifikators stehen.

$$C_{before}(C_1, C_2) = \{r | r \in R_1 : \exists r_2 \in R_2 : (r \ \ \textbf{before}\ r_2)\}$$

JustBefore-Klassifikator Erkennt alle Textregionen des ersten übergebenen Klassifikators, die direkt vor einer Textregion des zweiten Klassifikators stehen.

$$C_{justBefore}(C_1, C_2) = \{r | r \in R_1 : \exists r_2 \in R_2 : (r \ \ \textbf{justBefore}\ r_2)\}$$

BeginsWith-Klassifikator Erkennt alle Textregionen des ersten übergebenen Klassifikators, die mit einer Textregion des zweiten Klassifikators beginnen.

$$C_{beginsWith}(C_1, C_2) = \{r | r \in R_1 : \exists r_2 \in R_2 : (r \ \ \textbf{beginsWith}\ r_2)\}$$

EndsWith-Klassifikator Erkennt alle Textregionen des ersten übergebenen Klassifikators, die mit einer Textregion des zweiten Klassifikators enden.

$$C_{endsWith}(C_1, C_2) = \{r | r \in R_1 : \exists r_2 \in R_2 : (r \ \ \textbf{endsWith}\ r_2)\}$$

Contains-Klassifikator Erkennt alle Textregionen des ersten übergebenen Klassifikators, die eine Textregion des zweiten Klassifikators enthalten.

$$C_{contains}(C_1, C_2) = \{r | r \in R_1 : \exists r_2 \in R_2 : (r \ \ \textbf{contains}\ r_2)\}$$

ContainsNot-Klassifikator Erkennt alle Textregionen des ersten übergebenen Klassifikators, die keine Textregion des zweiten Klassifikators enthalten.

$$C_{containsNot}(C_1, C_2) = \{r | r \in R_1 : \neg\exists r_2 \in R_2 : (r \ \ \textbf{contains}\ r_2)\}$$

In-Klassifikator Erkennt alle Textregionen des ersten übergebenen Klassifikators, die in einer Textregion des zweiten Klassifikators enthalten sind.

$$C_{in}(C_1, C_2) = \{r | r \in R_1 : \exists r_2 \in R_2 : (r \ \ \textbf{in}\ r_2)\}$$

IsPostfixOf-Klassifikator Erkennt alle Textregionen des ersten übergebenen Klassifikators, die innerhalb einer Textregion des zweiten Klassifikators stehen und deren Start-Indizes gleich sind:

$$C_{postfix}(C_1, C_2) = \{r | r \in R_1 : \exists r_2 \in R_2 : (r \ \ \textbf{postfixOf}\ r_2)\}$$

IsPrefixOf-Klassifikator Erkennt alle Textregionen des ersten übergebenen Klassifikators, die direkt vor einer Textregion des zweiten Klassifikators stehen.

$$C_{prefix}(C_1, C_2) = \{r | r \in R_1 : \exists r_2 \in R_2 : (r \ \ \textbf{prefixOf}\ r_2)\}$$

Overlaps-Klassifikator Erkennt alle Textregionen des ersten übergebenen Klassifikators, die sich mit einer Textregion des zweiten Klassifikators überlappen.

$$C_{overlaps}(C_1, C_2) = \{r | r \in R_1 : \exists r_2 \in R_2 : (r \ \ \textbf{overlaps}\ r_2)\}$$

OverlapsBefore-Klassifikator Erkennt alle Textregionen des ersten übergebenen Klassifikators, die sich mit einer Textregion des zweiten Klassifikators am Anfang der zweiten Textregion überlappen.

$$C_{overlapsBefore}(C_1, C_2) = \{r | r \in R_1 : \exists r_2 \in R_2 : (r \ \ \textbf{overlapsBefore}\ r_2)\}$$

OverlapsAfter-Klassifikator Erkennt alle Textregionen des ersten übergebenen Klassifikators, die sich mit einer Textregion des zweiten Klassifikators am Ende der zweiten Textregion überlappen.

$$C_{overlapsAfter}(C_1, C_2) = \{r | r \in R_1 : \exists r_2 \in R_2 : (r \ \ \textbf{overlapsAfter}\ r_2)\}$$

3.2.3 Erstellung des Syntaxbaums

Aus den erlernten Textkonzepten wird automatisch ein Syntaxbaum erstellt. Aus diesem Syntaxbaum wird anschließend ein XML-Dokument erstellt werden. Der Syntaxbaum besteht dabei aus Knoten, die jeweils den Namen des beschriebenen Textkonzepts sowie den Textbereich im Quelldokument enthält. Die Kinder $c_i[s_i, e_i],\ i = 1..n$ eines Knotens $p[s, e]$ sind dabei nach ihrem Vorkommen im Text sortiert, wobei folgende Einschränkungen gelten:

$$s = s_1 \tag{3.16}$$
$$e = e_n \tag{3.17}$$
$$s_i = e_{i-1} \quad \forall i = 2..n \tag{3.18}$$

Das heißt also, dass sich die Kinder eines Knotens über den gesamten Bereich des aktuellen Knotens erstrecken , es existieren also keine „Zwischenräume“. Die Textregionen, die nicht von Textkonzepten abgedeckt werden, werden mit anonymen Knoten aufgefüllt.

Die automatische Generierung des Syntaxbaums funktioniert folgendermaßen: Zunächst wird ein Wurzelknoten erstellt, welcher das komplette Quelldokument enthält. Anschließend wird für jede Textregion aller Textkonzepte geprüft, an welcher Stelle im Syntaxbaum ein neuer Knoten eingefügt werden soll (siehe Abbildung 3.5 und Abbildung 3.6). Dazu wird im aktuellen Knoten – beginnend mit dem Wurzelknoten – geprüft, ob die einzufügende Textregion a die Textregion eines Kindknoten c_i enthält, d.h. ob a *contains* c_i gilt. Wenn dies der Fall ist und der einzufügende Knoten keinen anderen Kindknoten überdeckt, wird der

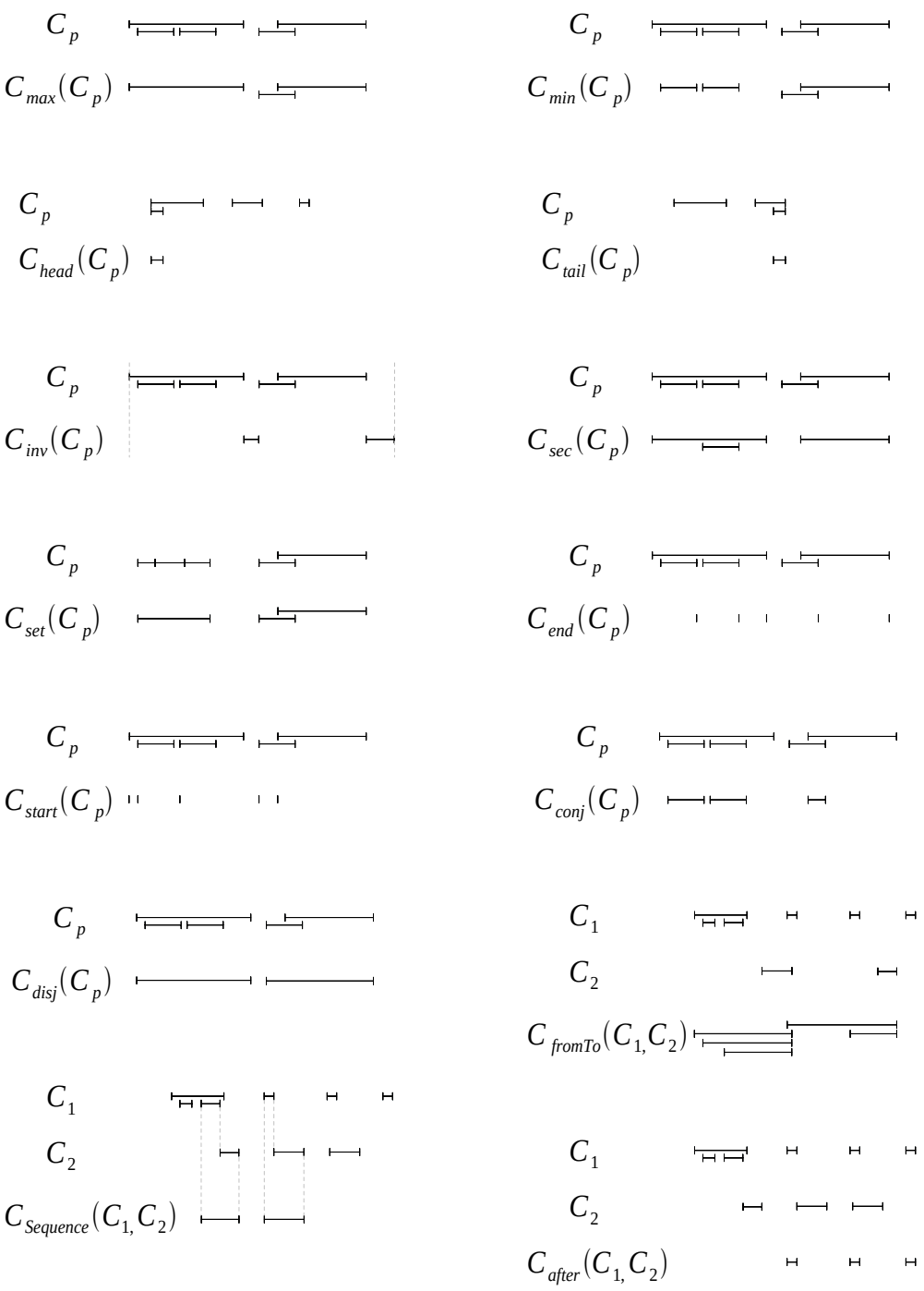
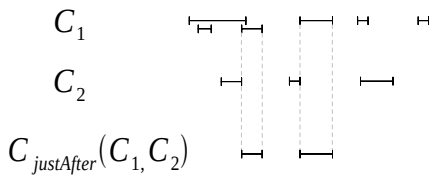
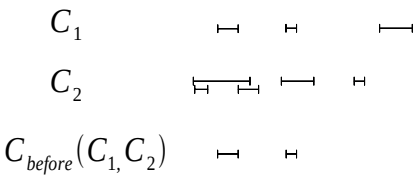
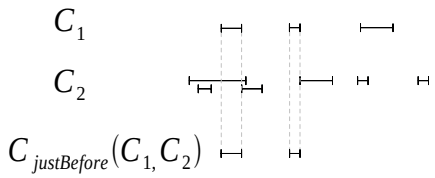
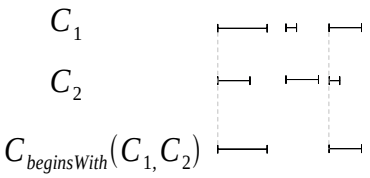


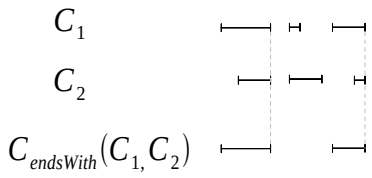
Abbildung 3.3: Beispiele für Textklassifikatoren.

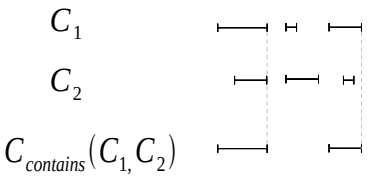


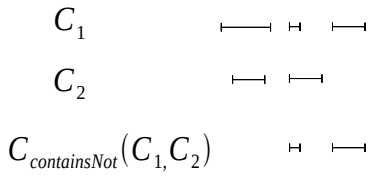


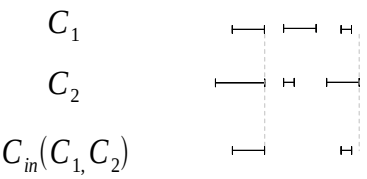


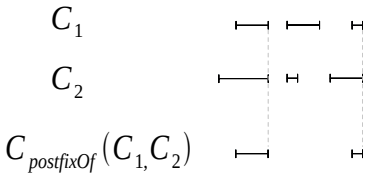


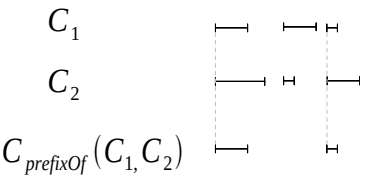


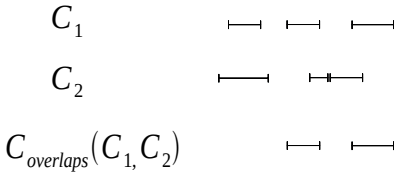


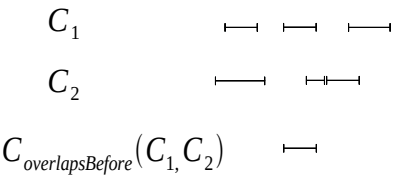












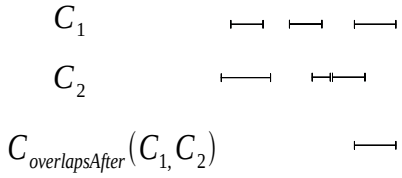


Abbildung 3.4: Beispiele für sämtliche Textklassifikatoren. (Fortsetzung)

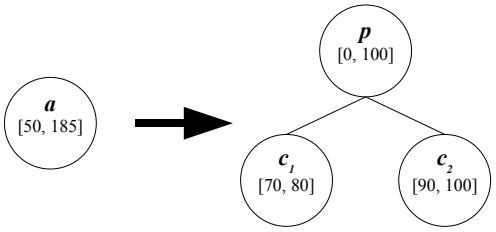


Abbildung 3.5: Einfügen eines Knotens in den Syntaxbaum

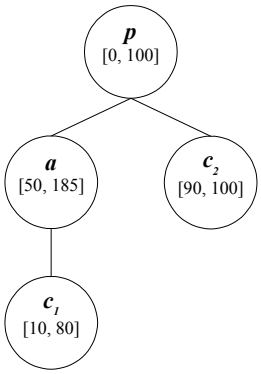


Abbildung 3.6: Syntaxbaum nach Einfügen von Knoten a

einzufügende Knoten als neuer Kindknoten des aktuellen Knotens eingefügt, der zu ersetzende Kindknoten wird als neuer Kindknoten des eingefügten Knotens eingefügt. Ist der einzufügende Knoten innerhalb eines Kindknotens c_k, so wird c_k der neue Vaterknoten p und der Algorithmus wird rekursiv mit dem neuen Vaterknoten aufgerufen. Enthält der aktuelle Knoten p keine Kindknoten, so wird der Knoten a dem Knoten p als Kindknoten hinzugefügt. In Abbildung 3.7 ist ein Beispiel für einen korrekten Syntaxbaum zu sehen. Wenn eine Überlappung zwischen dem einzufügenden Knoten a und einem beliebigem Knoten aus dem Syntaxbaum auftritt, wird dies als Fehler interpretiert und eine entsprechende Fehlermeldung wird ausgegeben. Eine solche Überlappung tritt insbesondere bei fehlerhaften Eingabedokumenten und Klassifikatoren auf.

Wenn der Syntaxbaum vollständig aufgebaut ist, werden leere Textregionen im Syntaxbaum mit speziellen Knoten aufgefüllt, um später auch nicht vom Benutzer klassifizierte Textregionen in der Ausgabedatei wieder zu finden. Dazu wird rekursiv für jeden Knoten $p[s, e]$ - beginnend mit dem Wurzelknoten - geprüft, ob die Kindknoten $c_i[s_i, e_i]$ den kompletten Textbereich von p abdecken, d.h. ob

$$\forall j \in [s, e] \exists i : s_i \leq j \leq e_i \tag{3.19}$$

gilt. Ist dies nicht der Fall, so werden anonyme Knoten eingefügt, um diese Bedingung herzustellen. Dazu werden alle Bereiche $b_k[s_k, e_k]$, die sich nicht mit Textbereichen aus dem Syntaxbaum überlappen, in den Syntaxbaum mit einem speziellen Namen (z. B. “not classified“) eingefügt. In Abbildung 3.8 ist der um anonyme Knoten erweiterte Syntaxbaum aus Abbildung 3.7 zu sehen.

3.2.4 Das XML-Ausgabeformat

Die Ausgabe des geparsten Dokuments erfolgt in einer XML-Datei. Dabei soll es möglich sein, jederzeit aus der generierten XML-Datei die ursprüngliche Datei re-

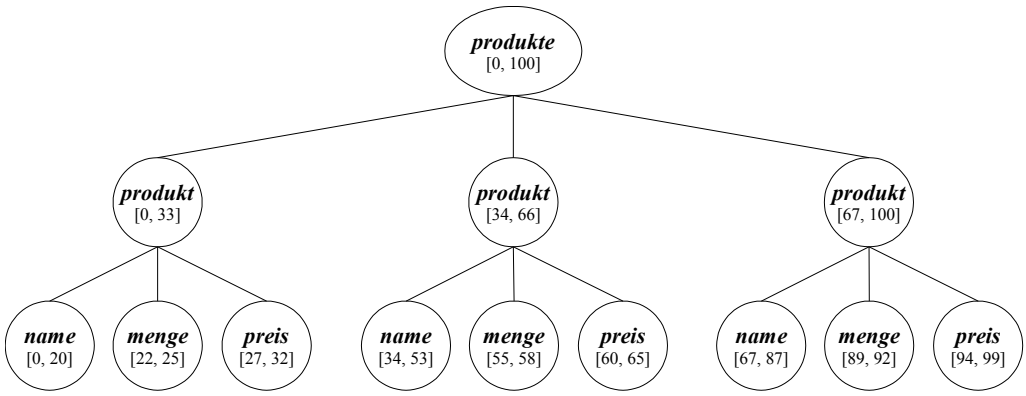


Abbildung 3.7: Ein korrekter Syntaxbaum

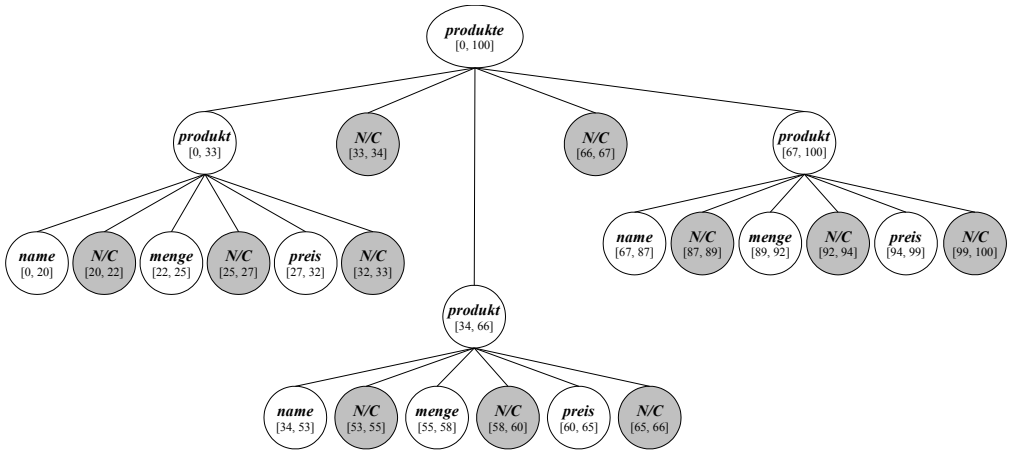


Abbildung 3.8: Ein korrekter Syntaxbaum mit anonymen Knoten

```
== Beispiele für das Semantic Web ==
Techniken des Semantic Web beginnen sich nur langsam und
teilweise durchzusetzen. Anwendungsbeispiele sind:
* Das [[Annotea]]-Projekt [http://www.w3.org/2001/Annotea/]
* Friend of a Friend [[FOAF]] [http://www.foaf-project.org/]
* W3Photo Project [http://w3photo.org]
* Reiseportal allIknow.net [http://alliknow.net]
```

Abbildung 3.9: Wiki-Quelltext

konstruieren zu können. Das führt natürlich dazu, dass häufig auch unerwünschte Teile des ursprünglichen Textes in der erstellten XML-Datei vorhanden sind, wie z. B. Trennzeichen oder Zeichen zur Textformatierung. Diese nicht benötigten Informationen können jedoch leicht per XSL-Transformation o.ä. herausgefiltert werden. Anhand des Beispiels in Abbildung 3.10 kann man nachvollziehen, wie die XML-Ausgabe zu dem Wiki-Text (aus [wik05]) in Abbildung 3.9 aussieht.

3.3 Die Lernalgorithmen

Zunächst einmal wird in diesem Abschnitt die Architektur des Systems zur Erlernung von Textklassifikatoren beschrieben. Anschließend werden die vom System verwendeten grundlegenden Algorithmen, z. B. zur Suche von gemeinsamen Substrings zweier Zeichenketten, dargelegt. Schließlich werden die Lernalgorithmen beschrieben und erläutert.

3.3.1 Architektur

Die Architektur des Lernsystems soll es ermöglichen, aus der Menge der Positivbeispiele, der Menge der Negativbeispiele, der Menge der bereits existierenden Textkonzepte sowie der Menge der Restriktionen eine Menge von Hypothesen, dargestellt durch Klassifikatoren, die das jeweilige Textkonzept klassifizieren, zu generieren. Außerdem soll die resultierende Menge der Hypothesen mit Hilfe von Heuristiken sortiert werden können, um möglichst schnell die passende Hypothese zu erhalten. Die Heuristiken zur Sortierung der Hypothesen werden in 3.3.3 vorgestellt. Einen Überblick über die Architektur ist in Abbildung 3.11 zu finden, während in Abbildung 3.12 eine detailliertere Übersicht über die Hypothesen-Generatoren zu finden ist.

Ein Positiv- oder Negativbeispiel wird dabei durch eine Textregion dargestellt, die in dem zu erlernenden Textkonzept zwingend enthalten sein muss bzw. nicht enthalten sein darf. Diese beiden Mengen sind disjunkt, es kann also keine Textregion sowohl in der Menge der Positivbeispiele, als auch in der Menge der Negativbeispiele enthalten sein. Diese Bedingung wird dadurch gewahrt, dass beim

```
<Document>
<Header>Beispiele für das Semantic Web</Header>
<not_Classified>Techniken des Semantic Web
beginnen sich nur langsam und teilweise durchzusetzen.
Anwendungsbeispiele sind:</not_Classified>
<List>
 <ListItem>
  <not_Classified>* </not_Classified>
  <Text>
   <not_Classified>Das </not_Classified>
   <InternalLink><Target>Annotea</Target></InternalLink>
   <not_Classified>-Projekt</not_Classified>
   <ExternalLink>
    <Target>http://www.w3.org/2001/Annotea/<Target>
   <ExternalLink>
  </Text>
 </ListItem>
 <ListItem>
  <not_Classified>* </not_Classified>
  <Text>
   <not_Classified>Friend of a Friend </not_Classified>
   <InternalLink><Target>FOAF</Target></InternalLink>
   <ExternalLink>
    <Target>http://www.foaf-project.org/<Target>
   <ExternalLink>
  </Text>
  </ListItem>
 <ListItem>
  <not_Classified>* </not_Classified>
  <Text>
   <not_Classified>W3Photo Project </not_Classified>
   <ExternalLink><Target>http://w3photo.org<Target><ExternalLink>
  </Text>
 </ListItem>
 <ListItem>
  <not_Classified>* </not_Classified>
  <Text>
   <not_Classified>Reiseportal allIknow.net </not_Classified>
   <ExternalLink><Target>http://alliknow.net<ExternalLink>
  </Text>
 </ListItem>
</List>
</Document>
```

Abbildung 3.10: Die gewünschte XML-Ausgabe

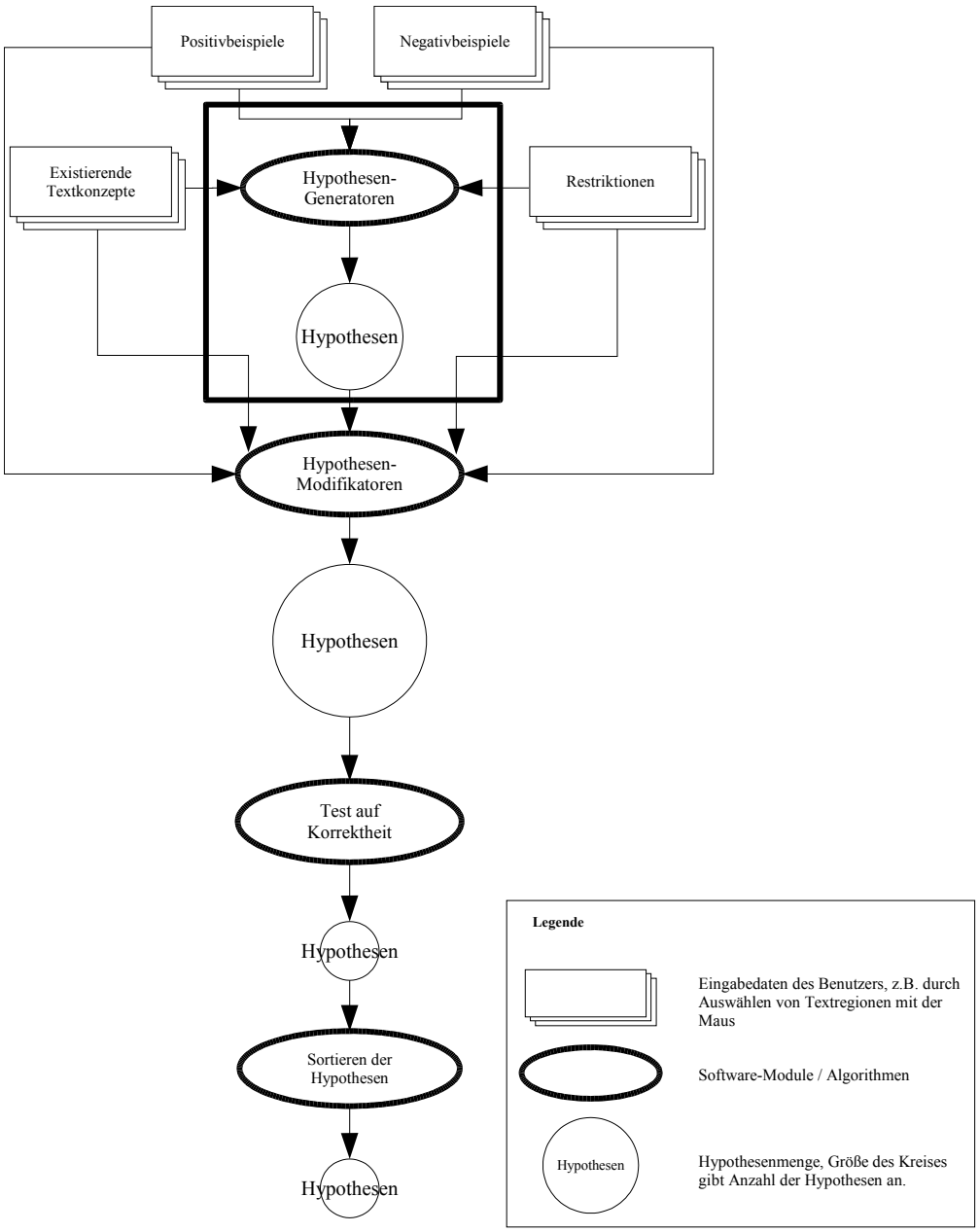


Abbildung 3.11: Architektur des Systems

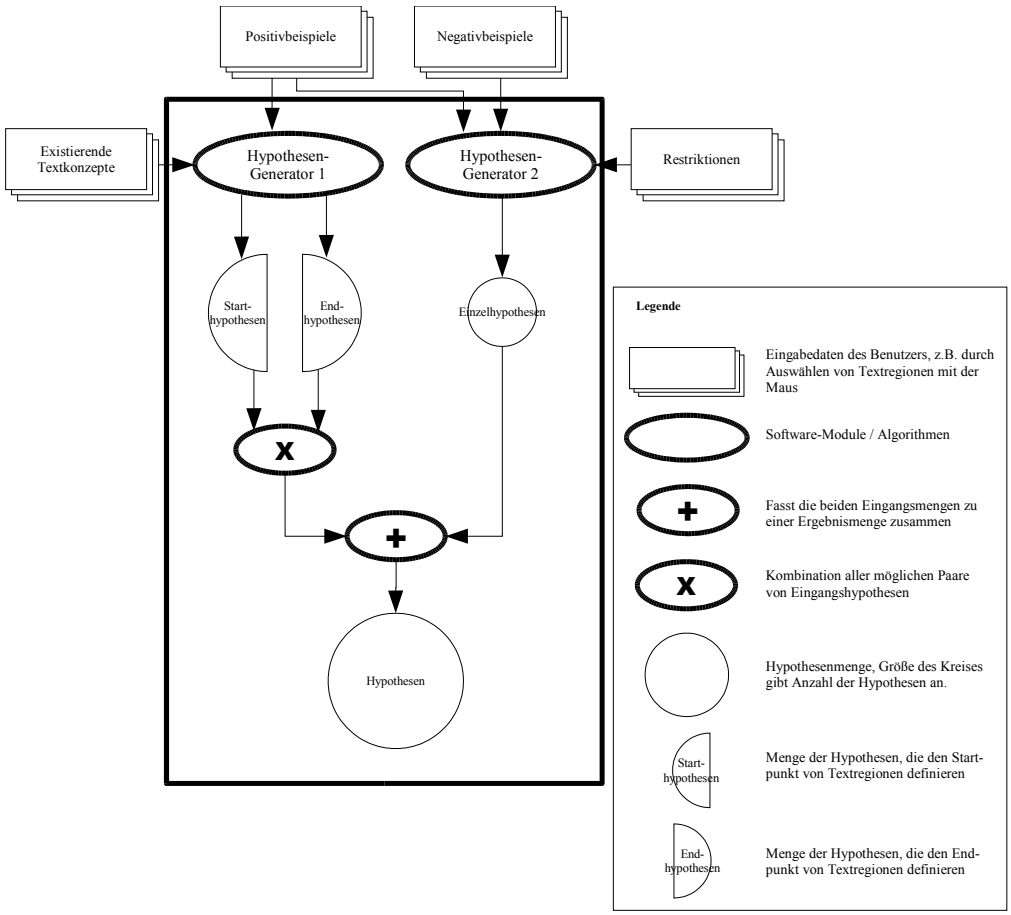


Abbildung 3.12: Architektur der Hypothesen-Generatoren mit zwei beispielhaften Hypothesen-Generatoren

Einfügen einer Textregion, die schon in der Menge der Negativbeispiele enthalten ist, in die Menge der Positivbeispiele, diese Region aus der Menge der Negativbeispiele entfernt wird. Analog dazu wird bei Textregionen verfahren, die in die Menge der Positivbeispiele eingefügt werden sollen und schon in der Menge der Negativbeispiele enthalten sind.

Ein fertiges Textkonzept setzt sich aus seinem Namen und dem verwendeten Klassifikator zusammen. Eine Restriktion besteht hier aus einer Einschränkung des Textkonzepts insofern, dass sie angibt, ob das zu erlernende Textkonzept in einem bestehenden Textkonzept enthalten ist oder ein solches enthält. D.h. bei einer einer Restriktion, dass Textkonzept A das Textkonzept B enthält, gilt:

$$\forall a_i \exists b_j : a_i \quad \textbf{is in} \quad b_j \tag{3.20}$$

Das System ist so aufgebaut, dass unabhängige Module jeweils Hypothesen generieren und zu der Hypothesenmenge hinzufügen können. Da jedoch häufig Textkonzepte durch charakteristische Textmerkmale am Anfang und am Ende einer Textregion beschrieben werden und diese Merkmale für Start und Ende der Textregion auch von verschiedenen Modulen erstellt werden können, ist es notwendig, die Hypothesen in Start- und Endhypothesen zu unterteilen (siehe Abbildung 3.12). Zusätzlich gibt es die Menge der Hypothesen, die nicht durch Start- und Endklassifikator beschrieben werden, sondern durch einen geschlossenen Klassifikator (z. B. Klassifikatoren, die exakt eine Zeichenkette klassifizieren. Hier sind keine separaten Start- und Endpunkte nötig. Beispiel: `'http://'`) definiert werden. Es existieren daher die folgenden drei Hypothesenmengen:

- Starthypothesen
- Endhypothesen
- Einzelhypothesen

Durch Kombination aller Starthypothesen mit allen Endhypothesen mittels eines `FromToClassifiers` erhält man nun eine Menge von Hypothesen. Basierend auf dieser Menge gibt es nun Module, die anhand der bestehenden Hypothesen (Klassifikatoren) modifizierte Hypothesen erstellen können. Üblicherweise werden so bestehende Klassifikatoren mit Einschränkungen versehen. Diese Module verändern nicht bestehende Hypothesen, sondern fügen lediglich anhand bestehender Hypothesen neue hinzu. Die Menge aller Hypothesen, inklusive der Einzelhypothesen, werden anschließend auf Korrektheit überprüft. Dazu wird jeder Klassifikator auf den zugrunde liegenden Text angewandt und jedes Positiv- und Negativbeispiel geprüft, ob es in der aus der Klassifikation entstandenen Menge von Textregionen enthalten ist, bzw. nicht enthalten ist. Wenn ein Positivbeispiel nicht vorhanden ist oder ein Negativbeispiel in der Menge der Textregionen enthalten ist, wird diese Hypothese verworfen. Die übrig bleibenden Hypothesen sind dann jeweils konsistent mit den Positiv- und Negativbeispielen.

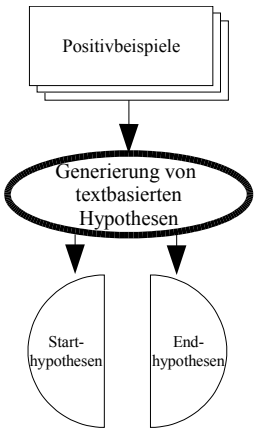
Algorithmus 1 Überprüfung der Korrektheit eines Klassifikators c

```
    function testCorrectness(c)
 2:     R = c.Classify(t)        ▷ Klassifiziere den Text t anhand des Klass. c.
        for all p ∈ P do                                  ▷ Für jedes Positivbeispiel
 4:         if p ∉ R then
                return false
 6:         end if
        end for
 8:     for all n ∈ N do                                  ▷ Für jedes Negativbeispiel
            if n ∈ R then
10:             return false
            end if
12:     end for
        return true
14: end function
```

3.3.2 Generierung von Hypothesen

Im folgenden Abschnitt werden die Algorithmen zur Generierung von Hypothesen vorgestellt und erläutert. Zunächst werden die Algorithmen zur Generierung von Start- und Endhypothesen beschrieben, anschließend die Algorithmen zur Generierung von Einzelhypothesen. Die Generierung von Start- und Endhypothesen wird zusammen behandelt, da die dazu verwendeten Algorithmen prinzipiell gleich sind und sich lediglich in Einzelheiten wie Laufrichtung oder Start- und Endwerten von Indexvariablen in Schleifen unterscheiden.

Textbasierte Hypothesen



Generiert: *String-Klassifikatoren*

Mit textbasierten Hypothesen sind solche Start- und Endhypothesen gemeint, die lediglich auf rein textuellen Eigenschaften der Positivbeispiele beruhen. Sie sind völlig unabhängig von bestehenden Textkonzepten. Bei der folgenden Beschreibung dieser Eigenschaften gilt für das i-te Positivbeispiel p_i aus der Menge der Positivbeispiele P mit einer passenden Zerlegung $p_i = b_i c_i d_i$, wobei b_i, c_i und d_i jeweils Zeichenfolgen sind, dass dieses sich direkt hinter einer Zeichenkette a_i und direkt vor einer Zeichenkette e_i befindet. Die Konkatenation der Zeichenfolgen b_i, c_i und d_i entspricht also dem i-ten Positivbeispiel. Es existieren folgende Eigenschaften zwischen mehreren Positivbeispielen:

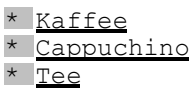


Abbildung 3.13: Beispiel für ein Präfix (grau hinterlegt) von Textregionen (unterstrichen)

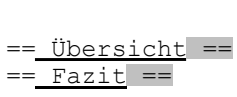


Abbildung 3.14: Beispiel für ein Postfix (grau hinterlegt) von Textregionen (unterstrichen)

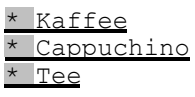


Abbildung 3.15: Beispiel für gemeinsame Textanfänge (grau hinterlegt) von Textregionen (unterstrichen)

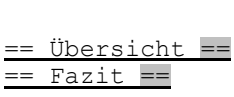


Abbildung 3.16: Beispiel für gemeinsame Textenden (grau hinterlegt) von Textregionen (unterstrichen)

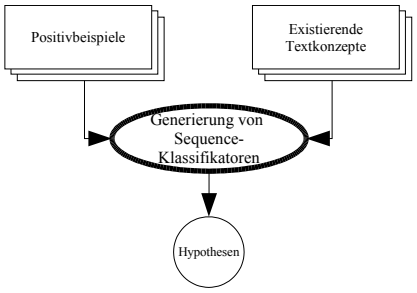
- gemeinsame Präfixe: $\forall p_i, p_j \in P : a_i = a_j$
- gemeinsame Postfixe: $\forall p_i, p_j \in P : e_i = e_j$
- gemeinsame Textanfänge: $\forall p_i, p_j \in P : b_i = b_j$
- gemeinsame Textenden: $\forall p_i, p_j \in P : d_i = d_j$

In den Abbildungen 3.13, 3.14, 3.15 und 3.16 sind Beispiele für die genannten Eigenschaften zu finden.

Sollte nun eins der der oben angegebenen Texteigenschaften gefunden werden, werden für diese Eigenschaft Start- bzw. Endhypothesen generiert. Dazu werden bei gemeinsamen Präfixen und Textenden $t[s, e]$ alle Teilzeichenfolgen t_i mit $t_i[s_i, e]$, $s \leq s_i < e$ gebildet. Analog dazu werden bei einem gemeinsamen Postfix oder Textanfang $t[s, e]$ alle Teilzeichenfolgen t_i mit $t_i[s, e_i]$, $s < e_i \leq e$ gebildet. Diese Teilzeichenfolgen enden bzw. beginnen also an der gleichen Stelle wie die zugrundeliegende Zeichenfolge.

Aus jeder so erhaltenen Teilzeichenfolge zu einem *Präfix* werden nun Starthypothesen erstellt. Dazu werden `StringClassifier` mit den jeweiligen Teilzeichenfolgen des Präfix erzeugt. Anschließend werden `EndOfClassifier` erzeugt, denen diese `StringClassifier` übergeben werden und die somit jeweils das Ende eines Präfixes markieren. Dieser Klassifikator wird dann zu den Starthypothesen hinzugefügt. Der `EndOfClassifier` ist notwendig, da ansonsten das Präfix bei dem letztendlich resultierenden Klassifikator innerhalb einer klassifizierten Textregion sein würde, was jedoch nicht erwünscht ist. Analog dazu wird zu jeder Teilzeichenfolge eines Postfix eine Endhypothese erstellt. Diese besteht aus einem `StringClassifier`, der die jeweilige Teilzeichenfolge klassifiziert und einem `StartOfClassifier`, der lediglich den Beginn jeder dieser Teilzeichenfolgen markiert. Dieser Klassifikator wird dann zu den Endhypothesen hinzugefügt. Für die Teilzeichenfolgen der Textanfänge und -enden werden jeweils entsprechende `StringClassifier` zu den Start- bzw. Endhypothesen hinzugefügt.

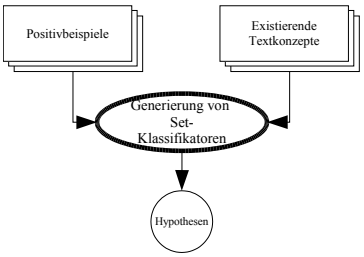
Auf bestehende Textkonzepten basierende Hypothesen



Generiert: *Sequence-Klassifikatoren*

Die rein textbasierten Hypothesen sind für einige Beispiele nicht mächtig genug, um alle gewünschten Textkonzepte zu klassifizieren. So ist es z. B. in einer CSV-Datei nicht möglich, alle Elemente einer Spalte zu klassifizieren, wenn die Datei mehr als drei Spalten enthält (Hat die CSV-Datei max. drei Spalten, so lässt sich die erste Spalte mit `BEGIN_OF_LINE until ';'` klassifizieren, die zweite Spalte mit `';' until ';' NOT_CONTAINS '\ n'` und die dritte Spalte mit `';' until EOL` klassifizieren). Um also mehr als drei Spalten korrekt klassifizieren zu können, werden mächtigere Klassifikatoren benötigt. Wurde etwa die erste Spalte bereits klassifiziert, könnte die zweite Spalte durch `EndOf <column1, ';'> until ';'` klassifiziert werden. Weitere Spalten können dann ebenso klassifiziert werden. Um solche Klassifikatoren zu generieren, wird zunächst geprüft, ob die vorhandenen Positivbeispiele direkt vor oder hinter bereits bestehenden Textkonzepten liegt. Wenn dies der Fall ist, kann direkt ein `EndOfClassifier` bzw. ein `StartOfClassifier` mit dem bestehenden Textkonzept zu den Start- bzw. Endhypothesen hinzugefügt werden. Häufig kommt es jedoch vor, dass ein zu lernendes Textkonzept nicht direkt hinter oder vor einem bestehenden liegt, sondern zwischen den beiden Textkonzepten noch ein festes Zeichen oder eine Zeichenfolge liegt. So sind z. B. bei CSV-Dateien zwei aufeinanderfolgende Spalten durch ein Semikolon getrennt. Um solche Textkonzepte zu klassifizieren, werden vor die bestehenden Starthypothesen und hinter die bestehenden Endhypothesen alle existierenden Textkonzepte mittels eines `SequenceClassifier`s gestellt. Bei einer Spalte einer CSV-Datei könnte z. B. `';'` als Starthypothese existieren. Dann wird vor diese Starthypothese ein bestehendes Textkonzept, z. B. `Spalte1`, gestellt. Die neu generierte Starthypothese wäre ann `<Spalte1, ';'`. Bei existierenden Start- bzw. Endhypothesen, die als äußeren Klassifikator einen `EndOfClassifier` bzw. `StartOfClassifier` enthalten, wird die Sequenz innerhalb dieser Klassifikatoren hinzugefügt. Die erweiterten Hypothesen werden dann zu den Start- bzw. Endhypothesen hinzugefügt.

Hypothesen, die zusammenhängende Textregionen existierender Textkonzepte klassifizieren



Generiert: *Set-Klassifikatoren*

Mit den bisher vorgestellten Modulen zur Erstellung von Hypothesen ist es z. B. möglich, alle Listeneinträge in einem Wiki-Dokument zu klassifizieren. Bisher ist es jedoch nicht möglich, eine ganze Liste zu klassifizieren. Es wird also ein Modul benötigt, welches in der Lage ist, zusammenhängende Textregionen eines Textkonzepts in einer Textregion zusammenzufassen. Um dies zu erreichen, wird wie folgt verfahren:

Für jede Textregion t_i des Textkonzepts, dessen zusammenhängende Textregionen gefunden werden sollen, wird geprüft, ob die darauf folgende Textregion t_{i+1} direkt hinter der aktuellen Textregion liegt. Ist dies der Fall, wird eine temporäre Textregion t_{temp} mit dem Start-Index von t_i und dem End-Index von t_{i+1} erzeugt. Ist bereits eine temporäre Textregion t_{temp} vorhanden, wird deren End-Index auf den End-Index von t_{i+1} gesetzt. Liegt t_{i+1} jedoch nicht direkt hinter t_i, wird t_{temp} dem aktuellen Textkonzept hinzugefügt und eine Kopie von t_{i+1} wird zur neuen temporären Textregion t_{temp}. Nachdem alle Textregionen durchlaufen sind, enthält das Textkonzept alle zusammenhängenden Textregionen des Basis-Textkonzepts.

3.3.3 Sortieren der Hypothesen

Nachdem alle Hypothesen generiert und auf Korrektheit überprüft wurden, werden in einem weiteren Schritt die Hypothesen mit Hilfe von Heursitiken sortiert, da in der Regel einige Hypothesen plasuibler für ein gesuchtes Textkonzept sind als andere. So sind z. B. bei folgenden zwei Beispielen für einen Wiki-Listeneintrag

```
*Wiki
*World Wide Web
```

folgende Hypothesen denkbar:

- `From '*W' UNTIL EndOfLine`
- `From '*' UNTIL EndOfLine`.

In diesem Fall ist die zweite Hypothese korrekt, da die erste Hypothese lediglich auf einem zufällig gleichen Anfangsbuchstaben der Listeneinträge beruht, der jedoch kein charakteristisches Merkmal des zu beschreibenden Textkonzepts ist. Da Textkonzepte häufig auf Sonderzeichen zur Markierung von Anfang und Ende beruhen, wurde das Verhältnis der Anzahl von Sonderzeichen zu der gesamten Zeichenzahl eines Klassifikators als Sortierkriterium verwendet. Klassifikatoren,

deren Sonderzeichen-Anteil gleich sind, werden nach der Anzahl der verwendeten Unterklassifikatoren sortiert. Dadurch wird vermieden, dass unnötig komplexe Klassifikatoren verwendet werden (*Occam's Razor*).

3.4 Beispiel

Im Folgenden wird die Erstellung eines Klassifikators anhand eines Beispiels nachvollzogen. Dazu wird versucht, aus dem folgenden Text aus einer Wiki-Seite alle Link-Ziele zu erkennen.

```
[[germany|Deutschlands]] Nachbaarstaaten:

* [[france|Frankreich]]
* Luxemburg
* Belgien
* [[netherland|Niederlande]]
* Dänemark
* Polen
* Tschechien
* [[austria|Österreich]]
* [[switzerland|Schweiz]]
```

Dazu werden zunächst dem System Positivbeispiele gegeben, zum Beispiel werden die Wörter `netherland` und `switzerland` markiert. Nun generiert das System Hypothesen. Zunächst wird der Text auf gemeinsame Präfixe, Postfixe, Textanfänge und Textenden untersucht. Als gemeinsames Präfix wird die Zeichenfolge `[[` erkannt, als gemeinsames Textende `erland` und als gemeinsames Postfix `|`. Daraus werden nun die beiden Starthypothesen `EndOf '['` und `EndOf '[['` sowie unter anderem die Endhypothesen `'land'`, `'erland'`, `'d'` und `BeginOf '|'` erzeugt. Die Start- und Endhypothesen werden nun kombiniert, d.h. es werden zum Beispiel folgende Hypothesen erstellt:

- `EndOf '[' until 'd'`
- `EndOf '[' until BeginOf '|'`
- `EndOf '[[' until BeginOf '|'`
- `EndOf '[' until 'land'`

Diese Hypothesen werden anschließend nach ihrem Anteil an Sonderzeichen absteigend sortiert. Man erhält nun also folgende Reihenfolge der Hypothesen:

1. `EndOf '[' until BeginOf '|'`
2. `EndOf '[[' until BeginOf '|'`

3. `EndOf '[' until 'd'`
4. `EndOf '[' until 'land'`

Die erste Hypothese dieser Reihenfolge wird dem Benutzer nun im Text visualisiert. Im vorliegenden Fall würden also folgende Textregionen von dem Klassifikator `EndOf '[' until BeginOf '|'` markiert werden:

- `[germany`
- `germany`
- `[france`
- `france`
- `[netherland`
- `netherland`
- `[austria`
- `austria`
- `[switzerland`
- `switzerland`

Wie man sieht, überlappen sich die Textregionen, da diese Hypothese den Beginn der gewünschten Textregion nach bereits einem `'['` erkennt, was jedoch nicht korrekt ist. Teilt man dem System nun mit, dass diese Textregionen, die eine eckige Klammer enthalten, nicht gewünscht sind, werden erneut mit Hilfe dieser zusätzlichen Information Hypothesen generiert. Diese könnten nun z. B. folgendermaßen aussehen:

- `EndOf '[[' until 'd'`
- `EndOf '[[' until BeginOf '|'`
- `EndOf '[[' until 'land'`

Diese Hypothesen werden wiederum nach ihrem Sonderzeichen-Anteil sortiert und die erste Hypothese (`EndOf '[[' until BeginOf '|'`) im Text hervorgehoben. Es würden im vorliegenden Beispiel dann folgende Textregionen markiert werden:

- `germany`
- `france`
- `netherland`
- `austria`
- `switzerland`

Es wurden also alle Link-Ziele in einem Wiki-Text mit Hilfe des Klassifikators `EndOf '[[' until BeginOf '|'` gefunden. Nun kann der Benutzer dem weitere Wiki-Elemente analog zu dem gezeigten Vorgehen erkennen lassen.

3.5 Benutzeroberfläche

Die Benutzeroberfläche ermöglicht eine möglichst einfache und intuitive Erstellung von Parsern. Folgende Funktionen stehen dabei zur Verfügung:

- Öffnen und Anzeigen von Textdateien
- Benennung von zusammengehörenden Textregionen
- Hinzufügen von Positivbeispielen
- Hinzufügen von Negativbeispielen
- Erfassung von Einschränkungen zu zusammengehörenden Textregionen
- Speichern der erlernten Parserdefinition
- Exportieren des aktuellen Dokuments in eine XML-Datei

Zunächst ermöglicht es die Benutzeroberfläche, beliebige Textdateien aus dem Dateisystem (sowohl lokal als auch aus dem Netzwerk) auszuwählen und zu öffnen. Diese Textdateien werden dann in einem Textfeld angezeigt. Zur Vereinfachung ist zunächst keine Möglichkeit vorgesehen, die aktuell geöffnete Datei zu editieren. Anschließend besteht die Möglichkeit, einen Namen für das anschließend zu erlernende Textkonzept einzugeben. So wäre z. B. im Wiki-Kontext „Überschrift“ ein mögliches Textkonzept. Man kann dann als Namen für dieses Konzept beispielsweise `Header` eingeben. Anschließend können dann im Text Beispiele für Überschriften markiert und zu den Positivbeispielen hinzugefügt werden. Das Programm generiert nun Hypothesen für weitere Bereiche im Text, die die gleichen Textkonzepte darstellen. Der Benutzer hat dann entweder die Möglichkeit, weitere Positivbeispiele anzugeben, oder aber Vorschläge des Systems zu verwerfen und somit dem Systsm Negativbeispiele zu geben. Dieser Vorgang kann nun beliebig oft wiederholt werden, bis das System das gewünschte Textkonzept erlernt hat.

Nun können weitere Textkonzepte erlernt werden, bei denen zusätzlich die Möglichkeit besteht, weitere Einschränkungen anzugeben. Diese Einschränkungen sind:

- Enthaltensein des aktuellen Textkonzepts in einem bestehenden Textkonzept
- Enthaltensein eines bestehenden Textkonzepts in dem aktuellen Textkonzept

Ein Beispiel zu diesen Einschränkungen aus dem Wiki-Umfeld wäre z. B. die Beschreibung eines Links, die sich immer in einem Link befindet, wie in Abbildung 3.5 (Beschreibung des Links ist unterstrichen) zu sehen. Diese zusätzlichen Informationen beschleunigen das Erlernen der Textkonzepte, sind jedoch nicht zwingend notwendig. Hat der Benutzer dem System alle gewünschten Textkonzepte

```
[Homepage von Sebastian Gerke|sgerke]
```

Abbildung 3.17: Ein Wiki-Link mit Label

beigebracht, kann diese Parserdefinition in eine Datei gespeichert werden, um später wiederverwendet zu werden. So kann diese Parserdefinition für ein Kommandozeilenprogramm verwendet werden, welches Text-Dateien anhand dieser Parserdefinition in XML umwandelt. Außerdem kann die aktuell geöffnete Datei anhand der erlernten Parserdefinition als XML-Datei exportiert werden.

4 Implementierung

Das in dieser Arbeit vorgestellte System wurde in Java implementiert. Java wurde wegen der Plattformunabhängigkeit und wegen der einfachen Erweiterbarkeit von Kontrollelementen der Benutzeroberfläche (Swing) gewählt. Da auch LAPIS in Java implementiert wurde, bestand auch die Möglichkeit, Datenstrukturen, Text Constraints oder die Lernalgorithmen zu übernehmen. Nach eingehender Untersuchung der Bibliotheken von LAPIS hat sich jedoch herausgestellt, dass die Möglichkeit, Text Constraints auf Objektebene zu erstellen, nicht ausgereift und für den vorliegenden Zweck unpassend ist. Es hätte lediglich die Möglichkeit bestanden, Text Constraints in ihrer String-Repräsentation dem Pattern-Matching-System der Text Constraints zu übergeben. Es ist zwar eine neue Version von LAPIS und damit auch der Text Constraints Planung, jedoch war nicht klar, wann diese veröffentlicht würde. Daher wurden diese Konzepte selbst implementiert. Darüber hinaus hätten bei der Verwendung der Text Constraints die Lernalgorithmen von LAPIS zumindest angepasst werden müssen, da im vorliegenden System zusätzliche Informationen für die Erstellung von Hypothesen vorliegen.

Textregionen wurden in dem beschriebenen System in der Klasse `Span` implementiert. Sie enthält die Felder `start` und `end`, die die Position der Textregion beschreiben. Die Methoden sind die Relationen zwischen zwei Spans, wie in den Formeln 3.1 bis 3.15 auf Seite 15 definiert.

4.1 Datenstrukturen

Klassifikatoren wurden in Unterklassen der abstrakten Klasse `Classifier` implementiert. Ein `Classifier` hat die Felder `String name` und eine geordnete Mengen von Abhängigkeiten `dependencies`. Unterklassen `Classifiers` müssen die Methoden `Layer classify(Text t)` implementieren. In dieser Methode werden aus dem übergebenen Text eine Menge von `Span`s in einem `Layer` zurückgegeben, die das gewünschte Textkonzept klassifizieren. Ein `Layer` ist ein Interface, deren implementierende Klassen hauptsächlich die Speicherung einer Menge von `Span`s übernehmen. Da verschiedene Möglichkeiten der effizienten Speicherung existieren und eine leichte Austauschbarkeit zwischen diesen Möglichkeiten gegeben sein soll, wurde hierzu ein Interface verwendet. In dem implementierten System wurde ein Layer mit Hilfe eines `TreeSet`, sortiert nach den Start-Indizes

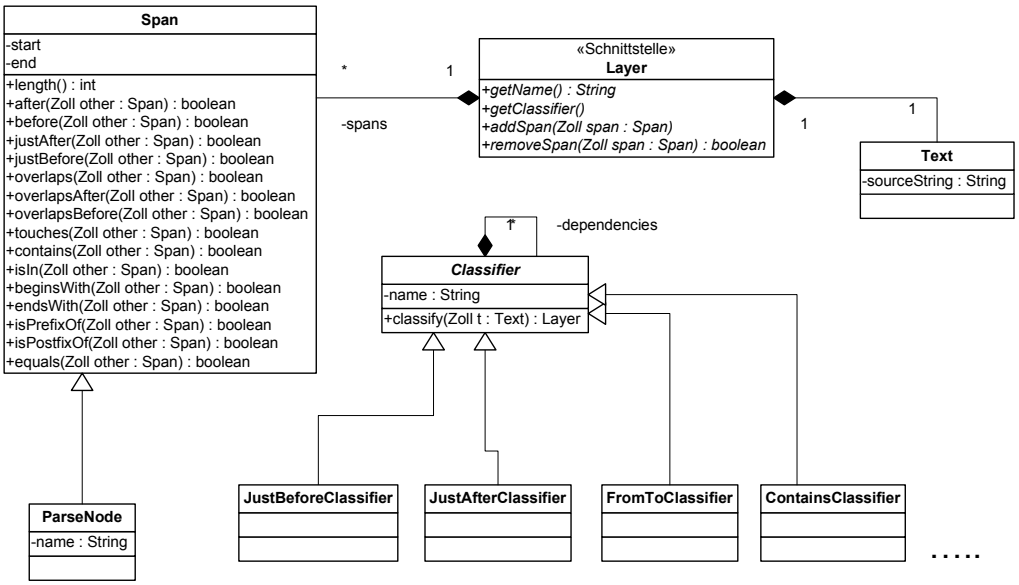


Abbildung 4.1: UML-Klassenmodell der Datenstrukturen

der `Span`s, realisiert. Eine Übersicht über die Datenstrukturen findet sich in den UML-Diagrammen in den Abbildungen 4.1 und 4.2.

4.2 Generierung der Hypothesen

Die Module zur Generierung von Hypothesen implementieren das Interface `ClassifierLearner`. Dazu müssen sie lediglich die Methode `learn()` implementieren, die als Parameter alle vorhandenen Informationen wie Positivbeispiele, Negativbeispiele, Einschränkungen und existierende Textkonzept erhält. Die Classifier-Learner sind dabei so implementiert, wie in 3.3.2 beschrieben. Dazu generieren alle vorhandenen Module Hypothesen und fügen sie der Menge der Hypothesen hinzu. Bei dem Test auf Korrektheit werden dann die unpassenden Hypothesen herausgefiltert.

Die Algorithmen zum Auffinden von gemeinsamen Präfixen und Teilzeichenketten sind in den Code-Auszügen 4.3 und 4.4 beschrieben. Die Algorithmen zum Auffinden von gemeinsamen Postfixen, Textanfängen und Textenden sind analog zu dem Algorithmus in Code-Auszug in 4.3, sie unterscheiden sich lediglich in der Anfangsposition und in einigen Vorzeichen.

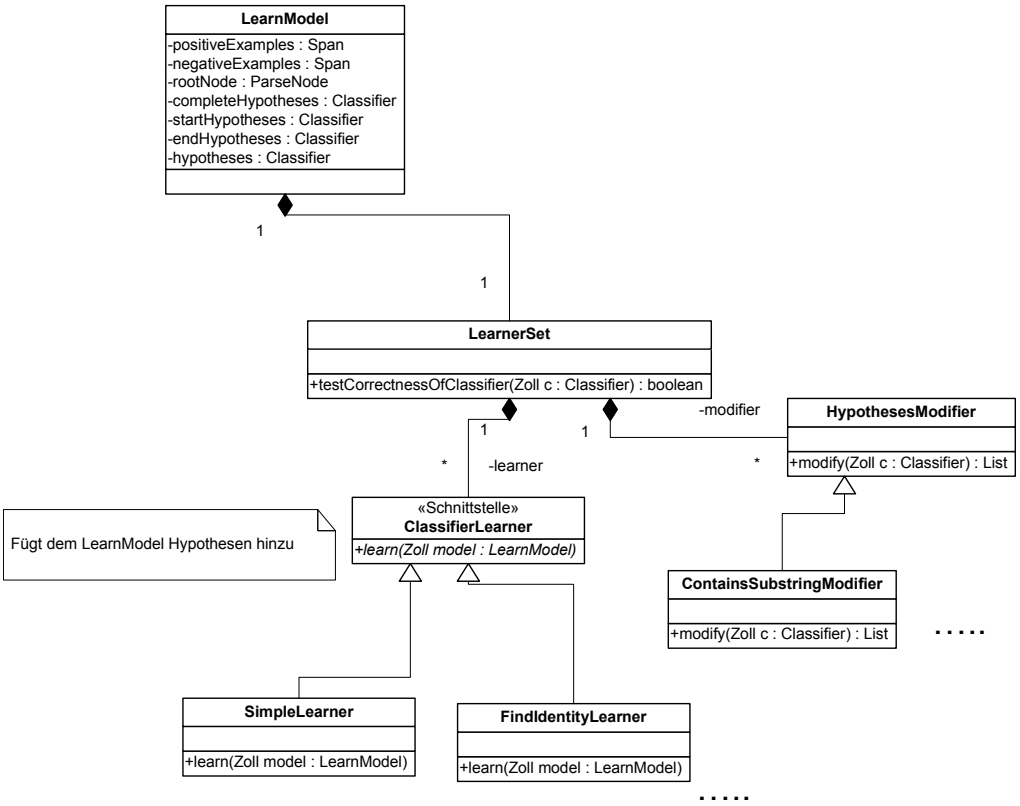


Abbildung 4.2: UML-Klassenmodell der Lern-Architektur

```
private String findPrefix(List<Span> spans)
{
  // Looking for prefixes of length j
  String retVal = null;

  int maxSize = spans.size() == 1 ? 5 : 50;
  for (int j = 1; j < maxSize; j++)
  {
    String currentSubstring = null;
    for (int i = 0; i < spans.size(); i++)
    {
      Span span = spans.get(i);
      int begin = span.getStart();
      if (begin - j >= 0
          && begin - j <= text.length())
      {
        String s = text.getSourceString().substring(
          Math.min(begin - j, begin),
          Math.max(begin - j, begin));
        if (currentSubstring == null)
          currentSubstring = s;
        else
        {
          if (!s.equals(currentSubstring))
            return retVal;
        }
      } else
        break;
    }
    if (currentSubstring != null)
      retVal = currentSubstring;
  }
  return retVal;
}
```

Abbildung 4.3: Algorithmus zum Auffinden von gemeinsamen Präfixen

```
public List<String> commonSubstrings(
  String s1, String s2)
{
  int m = s1.length();
  int n = s2.length();
  if( m > n )
  {   // Switch the order to make table smaller.
    return commonSubstrings( s2, s1 );
  }
  List<String> subStrings = new LinkedList<String>();
  int[] table = new int[ m + 1 ];
  for( int i = 0; i < n; i++ )
  {
    for( int j = m - 1; j >= 0; j-- )
    {
      if( s1.charAt( j ) == s2.charAt( i ) )
      {
        int len = table[ j ] + 1;
        table[ j + 1 ] = len;
        String s = s1.substring(j+1 - len, j+1);
        if (!subStrings.contains(s))
          subStrings.add(s);
      }
      else
        table[ j + 1 ] = 0;
    }
  }
  return subStrings;
}
```

Abbildung 4.4: Algorithmus zum Auffinden gemeinsamer Substrings

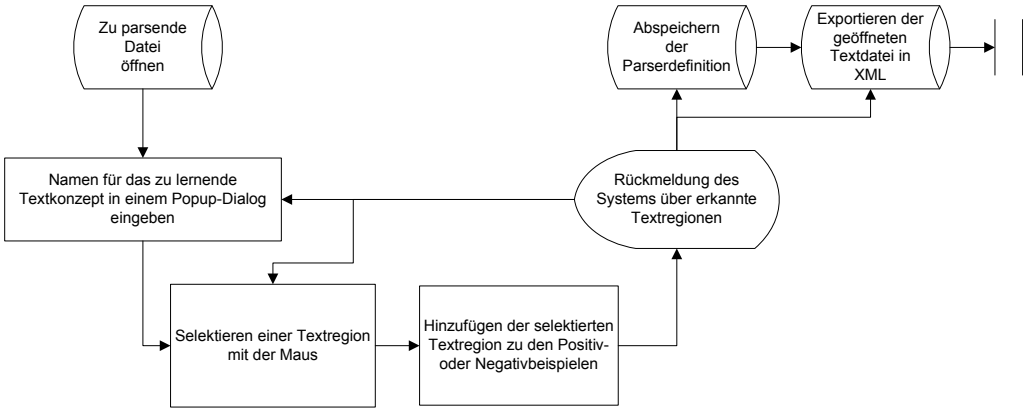


Abbildung 4.5: Ablauf der Benutzerinteraktion bei der Erstellung eines Parsers

4.3 Benutzeroberfläche

Die Benutzeroberfläche wurde mit Hilfe des Swing-Toolkits in Java implementiert. In Abbildung 4.6 ist ein Überblick über die Programmoberfläche zu sehen. Auf der linken Seite des Fensters befindet sich ein Textfenster, in dem die zu parsende Datei angezeigt wird. In der rechten Spalte (oben) befindet sich eine Liste, in der alle Hypothesen, die vom System vorgeschlagen wurden, aufgelistet sind. Die oberste Hypothese wird zusätzlich im Textfeld visualisiert. Dabei wird jede Textregion des Klassifikationsergebnisses des besten Klassifikators abwechselnd gelb oder orange hervorgehoben, um direkt aufeinander folgende Textregionen besser unterscheiden zu können. Unter der Liste der Hypothesen befindet sich eine Liste der bereits erlernten Textkonzepte. Schließlich befindet sich darunter ein Panel zur Eingabe von Einschränkungen, die die Schachtelung von Textkonzepten betreffen. So kann hier z. B. eingegeben werden (per Maus-Klick auf die entsprechende Check-Box), dass ein `LinkLabel` immer in einem `Link` enthalten ist. Unter dem Textfeld befindet sich noch eine Statusleiste, die den Namen des aktuell zu erlernenden Textkonzepts anzeigt. Die Erstellung eines Parsers läuft nun wie in dem Ablaufdiagramm in Abbildung 4.5 ab.

Zur Visualisierung von Textregionen in dem Textfeld wurde das in Swing existierende Textfeld so erweitert, dass man lediglich mit der Methode `addClassifier(Classifier c, Color col)` des `ClassifierHighlightingTextPane` den gewünschten `Classifier` und die gewünschte Farbe, in der der Klassifikator hervorgehoben werden soll, bekannt machen muss.

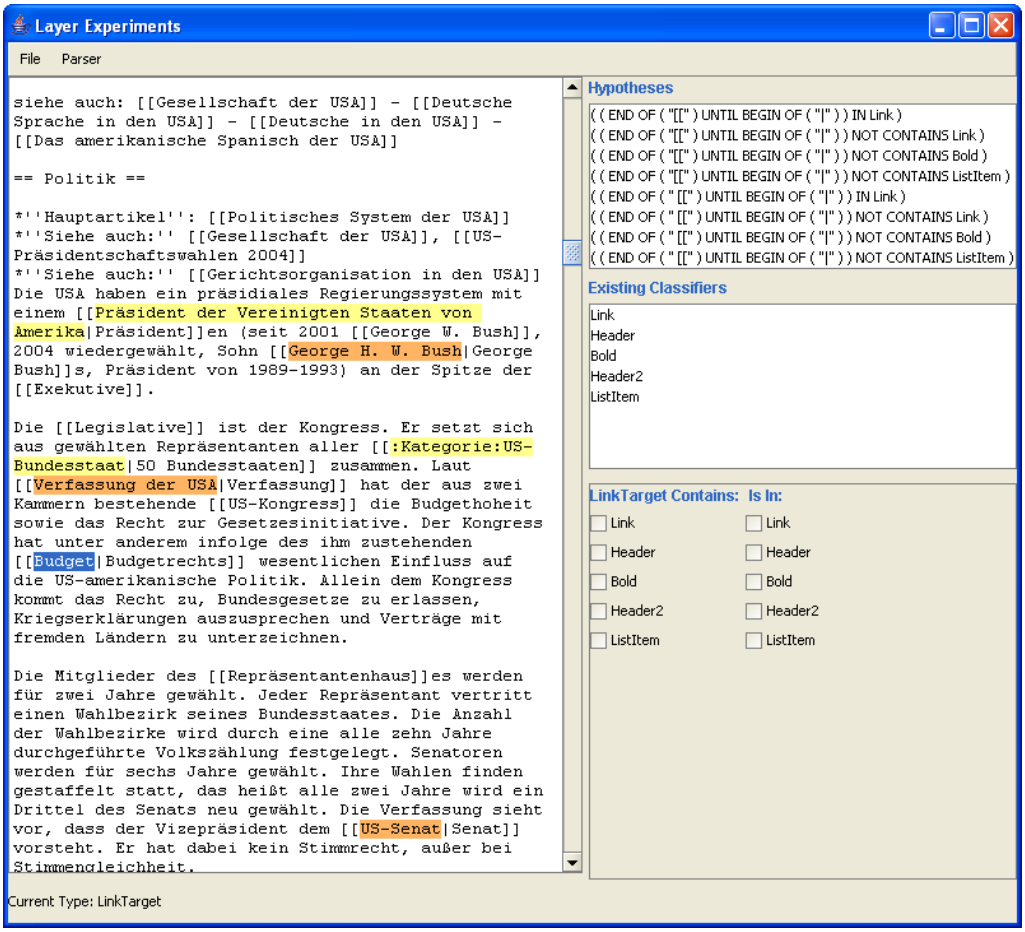


Abbildung 4.6: Benutzeroberfläche

5 Evaluation

In diesem Kapitel werden die erreichten Ergebnisse analysiert und mögliche Verbesserungen dargestellt. Die Ergebnisse werden anhand einer Wiki-Syntax (MediaWiki) dargelegt. Dazu werden die Syntax-Elemente von MediaWiki einzeln aufgeführt und angegeben, ob das System in der Lage ist, dieses Syntax-Element zu Erlernen und wieviele Benutzereingaben dazu nötig waren. In Tabelle 5.1 finden sich die Ergebnisse dieser Untersuchung. Wie man sieht, werden die meisten Syntax-Elemente eines Wikis erkannt. Lediglich Tabellen können mit dem aktuellen System nicht erlernt werden, da hierzu verschachtelte Textkonzepte benötigt werden. Bei den meisten Wiki-Syntaxelementen werden weniger Benutzereingaben benötigt, als dies bei LAPIS der Fall ist. Mit / markierte Textkonzepte konnten von dem jeweiligen System nicht erkannt werden. Es werden also vom vorgestellten System in acht der 14 Fälle weniger, in drei Fällen die gleiche Anzahl sowie in einem Fall mehr Benutzereingaben benötigt. Es ist jedoch zu beachten, dass bei dem vorgestellten System zum Teil eine kluge Reihenfolge der Syntaxelemente nötig war, um diese Textkonzepte zu erlernen. Es wurde jedoch auch bei LAPIS die gleiche Reihenfolge eingehalten.

Ein Ansatzpunkt für zukünftige Verbesserungen ist also, die Reihenfolge der Erlernung der Textkonzepte freier wählen zu können. Darüber hinaus wäre die Erkennung von kontextfreien Sprachen (also Sprachen mit z. B, geklammerten Ausdrücken).

Bei der Entwicklung des System wurden Performance-Aspekte nur rudimentär betrachtet. Für das Parsen eines 164 KB großen Wiki-Textes mit einer vollständigen Wiki-Syntax (bestehend aus zehn Klassifikatoren) werden auf einem 1,6 GHz Intel Centrino (1 GB RAM) nach einer Warmlaufphase der Java VM 1,9 Sekunden benötigt. Auch hier sind weitere Verbesserungen sicherlich möglich.

Syntax-Element	**LAPIS**	**vorg. System**
Header 1	15	7
Header 2	6	3
WebLink	8	2
Unlabelled Link	/	6
Labelled Link	5	5
InterWikiLink	8	3
LinkTarget	1	3
LinkLabel	3	2
Bold	5	2
Italic	3	1
ListItem	2	2
List	2	1
OrderedListItem	2	2
Table	5	/

Abbildung 5.1: Erkannte Elemente einer Wiki-Syntax (MediaWiki) mit Anzahl der benötigten Benutzereingaben

6 Zusammenfassung und Ausblick

In der vorliegenden Arbeit wurde ein System zur einfachen Erstellung von Parsern vorgestellt. Zur Erstellung eines Parsers müssen vom Benutzer lediglich Positiv- bzw. Negativbeispiele für ein einzelnes, zu erlernendes Textkonzept angegeben werden. Das System generiert anschließend anhand dieser Beispiele Hypothesen für Klassifikatoren, die das gewünschte Textkonzept klassifizieren sollen. So kann schließlich für jedes Textkonzept ein Klassifikator erlernt werden. Die daraus resultierende Parser-Definition kann dann abgespeichert werden und in einem Kommandozeilen-Parser wieder verwendet werden. Außerdem kann die aktuell geöffnete Text-Datei anhand der Parser-Definition in eine XML-Datei exportiert werden.

Das vorgestellte System ähnelt insbesondere in der beispielgestützten Erkennen von Textmustern dem Texteditor und Webbrowser LAPIS. Deren Pattern-Beschreibungssprache wurde jedoch im vorliegenden System nicht verwendet, sondern eine eigene Textmuster-Beschreibungssprache implementiert, da die in LAPIS verwendete Beschreibungssprache sich nicht für den direkten Zugriff aus anderen (Java-)Systemen eignet. Außerdem unterscheidet sich die in dem beschriebenen System verwendeten Lernalgorithmen von denen in LAPIS. Zum einen werden in dem hier beschriebenen System weniger Hypothesen generiert, da der Fokus auf strukturierteren Daten liegt als dies in LAPIS der Fall ist. Dadurch ist die Beschreibungsmächtigkeit in LAPIS zwar höher, dies führt jedoch auch dazu, dass mehr Beispiele benötigt werden, um einen gewünschten Text-Klassifikator zu erlernen. Für die meisten Anwendungsfälle ist diese Ausdrucksmächtigkeit ausreichend. Lediglich in Einzelfällen ist es dem beschriebenen System nicht möglich, passende Klassifikatoren zu generieren. Auch ist es durch die geringere Ausdrucksmächtigkeit zum Teil notwendig, eine bestimmte Reihenfolge beim Erlernen der Textkonzepte einzuhalten.

Zusammenfassend lässt sich sagen, dass das vorliegende System ein bereits benutzbares Werkzeug darstellt, welches für die gebräuchlichsten Anwendungsfälle eine einfache und bequeme Möglichkeit darstellt, Parser zu erstellen und Dateien in XML-Dateien zu konvertieren. Weitere Verbesserungen sind in den Punkten Performance und Ausdrucksmächtigkeit möglich. Durch die Erhöhung der Ausdrucksmächtigkeit würde auch das Problem der Einhaltung einer bestimmten Reihenfolge bei der Erlernung von Textkonzepten abgeschwächt werden, es lässt sich jedoch nicht immer vermeiden, bestimmte Textkonzepte vor anderen Textkonzepten dem System bekanntzumachen.

Literaturverzeichnis

[Bri93] Brill, Eric: Automatic Grammar Induction and Parsing Free Text: A Transformation-Based Approach. In: *Meeting of the Association for Computational Linguistics*, 1993, S. 259–265

[CHS99] Chen, Hao ; Hu, Jianying ; Sproat, Richard W.: Integrating geometrical and linguistic analysis for email signature block parsing. In: *ACM Transactions on Information Systems* 17 (1999), Nr. 4, S. 343–366

[Cir01] Ciravegna, F. *LP an Adaptive Algorithm for Information Extraction from Web-related Texts.* 2001

[Her00] Hermjakob, Ulf: Rapid parser development: a machine learning approach for Korean. In: *Proceedings of the first conference on North American chapter of the Association for Computational Linguistics.* San Francisco, CA, USA : Morgan Kaufmann Publishers Inc., 2000, S. 118–123

[LS94] Liu, Rey-Long ; Soo, Von-Wun: A corpus-based learning technique for building a self-extensible parser. In: *Proceedings of the 15th conference on Computational linguistics.* Morristown, NJ, USA : Association for Computational Linguistics, 1994, S. 441–446

[Mil02] Miller, Robert C.: *Lightweight structure in text*, Diss., 2002. – Co-Chair-Brad A. Myers and Co-Chair-David Garlan

[MY00] Min-Yen, K. *Combining visual layout and lexical cohesion features for text segmentation.* 2000

[Sod99] Soderland, Stephen: Learning Information Extraction Rules for Semi-Structured and Free Text. In: *Machine Learning* 34 (1999), Nr. 1-3, S. 233–272

[wik05] Semantisches Web - Wikipedia. In: *Wikipedia* (2005)

Erklärung

Ich versichere hiermit wahrheitsgemäß, die Arbeit bis auf die dem Aufgabensteller bereits bekannte Hilfe selbständig angefertigt, alle benutzten Hilfsmittel vollständig und genau angegeben und alles kenntlich gemacht zu haben, was aus Arbeiten anderer unverändert oder mit Abänderungen entnommen wurde.

Karlsruhe, den 29. Oktober 2005