Lehrstuhl Angewandte Informatik IV
Datenbanken und Informationssysteme
Prof. Dr.-Ing. Stefan Jablonski

Institut für Angewandte Informatik
Fakultät für Mathematik, Physik und Informatik
Universität Bayreuth

Bachelorarbeit

Klaus Freiberger

Juni 14, 2019
Version: Final

Universität Bayreuth

Fakultät Mathematik, Physik, Informatik
Institut für Informatik
Lehrstuhl für Angewandte Informatik IV

Rapid Miner NLP Tools: Syntax Parsing

Bachelorarbeit

Klaus Freiberger

1. Reviewer Dr. Lars Ackermann
Fakultät Mathematik, Physik, Informatik
Universität Bayreuth

2. Reviewer Prof. Dr.-Ing. Stefan Jablonski
Fakultät Mathematik, Physik, Informatik
Universität Bayreuth

Supervisors Lars Ackermann and Stefan Jablonski

Juni 14, 2019

Klaus Freiberger

Bachelorarbeit

Rapid Miner NLP Tools: Syntax Parsing, Juni 14, 2019

Reviewers: Dr. Lars Ackermann and Prof. Dr.-Ing. Stefan Jablonski

Supervisors: Lars Ackermann and Stefan Jablonski

Universität Bayreuth

Lehrstuhl für Angewandte Informatik IV

Institut für Informatik

Fakultät Mathematik, Physik, Informatik

Universitätsstrasse 30

95447 Bayreuth

Germany

Abstract

Dies hier ist ein Blindtext zum Testen von Textausgaben. Wer diesen Text liest, ist selbst schuld. Der Text gibt lediglich den Grauwert der Schrift an. Ist das wirklich so? Ist es gleichgültig, ob ich schreibe: „Dies ist ein Blindtext“ oder „Huardest gefburn“? Kjift – mitnichten! Ein Blindtext bietet mir wichtige Informationen. An ihm messe ich die Lesbarkeit einer Schrift, ihre Anmutung, wie harmonisch die Figuren zueinander stehen und prüfe, wie breit oder schmal sie läuft. Ein Blindtext sollte möglichst viele verschiedene Buchstaben enthalten und in der Originalsprache gesetzt sein. Er muss keinen Sinn ergeben, sollte aber lesbar sein. Fremdsprachige Texte wie „Lorem ipsum“ dienen nicht dem eigentlichen Zweck, da sie eine falsche Anmutung vermitteln.

Abstract (different language)

Dies hier ist ein Blindtext zum Testen von Textausgaben. Wer diesen Text liest, ist selbst schuld. Der Text gibt lediglich den Grauwert der Schrift an. Ist das wirklich so? Ist es gleichgültig, ob ich schreibe: „Dies ist ein Blindtext“ oder „Huardest gefburn“? Kjift – mitnichten! Ein Blindtext bietet mir wichtige Informationen. An ihm messe ich die Lesbarkeit einer Schrift, ihre Anmutung, wie harmonisch die Figuren zueinander stehen und prüfe, wie breit oder schmal sie läuft. Ein Blindtext sollte möglichst viele verschiedene Buchstaben enthalten und in der Originalsprache gesetzt sein. Er muss keinen Sinn ergeben, sollte aber lesbar sein. Fremdsprachige Texte wie „Lorem ipsum“ dienen nicht dem eigentlichen Zweck, da sie eine falsche Anmutung vermitteln.

Inhaltsverzeichnis

1 Einleitung

2 Natural Language Processing

In diesem Kapitel werden der theoretische Hintergrund und die Methoden des Natural-Language-Processings eingeführt. Nach den Grundlagen werden zwei verschiedene Methoden zum Parsen vorgestellt. Ausführlichere Beschreibung der meisten Punkte ist zu finden in [MJ09].

2.1 Grundlagen

Betrachtet wird ein Satz einer natürlichen Sprache, die im Rahmen dieser Arbeit auf Englisch festgelegt ist. Als Satz wird hier eine Folge von mindestens einem Wort einer Sprache gesehen. Dieser wird durch ein Satzzeichen abgeschlossen. Ein Algorithmus kann so einen natürlich-sprachlichen Satz nicht ohne Weiteres interpretieren. Hierfür bedarf es verschiedener Hilfsstrukturen und zusätzlicher Informationen.
Als ersten Schritt bietet es sich an, den einzelnen Wörtern eines Satzes ihre Wortart zuzuordnen. Mit Wortart, alternativ auch Wortklasse oder im Englischen *part-of-speech* (POS), ist gemeint, wie ein Wort im Satz auftritt. Beispiele für Wortarten sind Nomen, Verb und Adjektiv.
Für die verschiedenen Wortklassen gibt es Abkürzungen, genannt *Tags*. Jedem Wort der Sprache kann mindestens ein *Tag* zugewiesen werden. In dieser Arbeit wird das *Tagset*, die Menge an Wortarten aus der *Penn-Treebank*, verwendet. Aufgelistet sind Tabelle 2.1.

Der Satz

> My dog also likes eating sausage.

würde mit diesem *Tagset* folgendermaßen annotiert werden:

> My/PRP$ dog/NN also/RB likes/VBZ eating/VBG sausage/NN ./.

Das maschinelle zuweisen von *POS-Tags* geschieht über das sogenannte *Tagging*. Das ist nicht weiter Bestandteil dieser Arbeit.
Darüber hinaus zeigt sich, dass in der englischen Sprache oftmals mehrere Wörter eine *Konstituente* bilden. Darunter versteht man eine Gruppe von Wörtern, die sich

Penn-Treebank Part-of-Speech Tags		
Tag	Beschreibung	Beispiel
CC	Koordinierende Konjunktion	*and*
CD	Kardinalzahl	*third*
DT	Artikel	*the*
EX	Existentielles *there*	*there* is
FW	Fremdword	*les*
IN	Präposition, unterordnende Konjunktion	*in*
JJ	Adjektiv	*green*
JJR	Adjektiv, Komparativ	*greener*
JJS	Adjektiv, Superlativ	*greenest*
LS	Listenelement Markierung	*1)*
MD	Modal	*could*
NN	Nomen, singular oder Masse	*table*
NNS	Nomen, plural	*tables*
NNP	Eingenname, singular	*Germany*
NNPS	Eigenname, plural	*Vikings*
PDT	Predeterminer	*both* his children
POS	possesive Endung	*'s*
PRP	Personalpronomen	*me*
PRP$	Possesivpronomen	*my*
RB	Adverb	*extremely*
RBR	Adverb, Komparativ	*better*
RBS	Adverb, Superlativ	*best*
RP	Partikel	*about*
SYM	Symbol	*%*
TO	to	what *to* do
UH	Ausruf	*oops*
VB	Verb, Grundform	*be*
VBD	Verb, Vergangenheitsform	*was*
VBG	Verb, Gerund \Partizip Präsens	*being*
VBN	Verb, Partizip Perfekt	*been*
VBP	Verb, Präsens, nicht 3.Person Singular	*am*
VBZ	Verb, Präsens, 3.Person Singular	*is*
WDT	*wh*-Artikel	*which*
WP	*wh*-Pronomen	*who*
WP$	*wh*-Possesivpronomen	*whose*
WRB	*wh*-Adverb	*be*

Tab. 2.1: Penn-Treebank POS Tags [Mar+93]

```
(S
    (NP (PRP$ My) (NN dog))
    (ADVP (RB also))
    (VP (VBZ likes)
        (S
            (VP (VBGeating)
            (NP (NN sausage)))))
    (. .))
```

Abb. 2.1: Syntaktische Annotation in mehrzeiliger, geklammerter Darstellung

im Satz als Einheit verhalten. Ein Beispiel hierfür ist die Nominalphrase *My Dog* oder die Verbalphrase *likes eating sausage*. Für diese Einheiten gibt es wiederum eine Sammlung an *Tags*, wobei hier ebenfalls die der *Penn-Treebank* verwendet werden. Beschreibung dieser *Tags* findet sich in Tabelle 2.2. Diese Einheiten bilden die syntaktischen Komponenten eines Satzes. Eine Konstituente hat ein *Tag*, das beschreibt um welche Art es sich handelt, und eine Spanne, die angibt welche Wörter in ihr enthalten sind. Die Konstituente, die den kompletten Satz enthält, wird hier oberste Konstituente genannt [MJ09, Kapitel 10].
Zu dieser Menge an Tags gibt es noch die Erweiterung um relationale Tags. Diese liefern eine Zusatzinformation und werden an die eben vorgestellten angehängt. Zum Beispiel bekommt das Subjekt eines Satzes das Suffix *-SBJ*. Das heißt aus einer Nominalphrase *NP* wird, falls sie Subjekt ist, *NP-SBJ* [Rel]. Dieser Erweiterungssatz ist nicht Teil dieser Arbeit, wurde aber der Vollständigkeit halber erwähnt.
In Abbildung **??** wird der Satz mit der syntaktischen Annotation dargestellt. Mit Annotation sind in dieser Arbeit die *Tags* und Klammerungen gemeint. Ein Satz heißt annotiert, wenn alle seine Konstituenten und Wörter mit einem *Tag* und der entsprechenden Klammerung versehen sind.
Wie man am Beispiel erkennen kann, sind im Gegensatz zu den POS-Tags auch diverse Verschachtelungen dieser Komponenten möglich. Um diese Anordnungsstruktur innerhalb einer Sprache zu beschreiben, bietet sich eine Grammatik, genauer eine kontextfreie Grammatik, an [MJ09, Kapitel 10].
Eine Grammatik ist 4-Tupel aus Nichtterminalen, Terminalen, Regeln und einem Startsymbol. Nichtterminal, oder Variablen, sind in unserem Fall sowohl syntaktische, als auch *POS-Tags*. Terminale sind alle Wörter und Satzzeichen der natürlichen Sprache. Regeln, oder Produktionen, haben die Form $\alpha \rightarrow \beta$, wobei α eine Menge von Nichtterminalen und β eine Menge von Terminalen und Nichtterminalen ist. α wird linke Seite und β rechte Seite der Regel genannt. Eine Produktion gibt die Möglichkeit, die linke Seite in die rechte zu überführen. Dieses Überführen wird auch ableiten genannt. Das Startsymbol ist ein Nichtterminal, von dem ausgehend die

Penn-Treebank Syntactic Tags		
Tag	Beschreibung	Beispiel
S	einfacher deklarativer Satz	*There we go.*
SBAR	Satz beginnend mit unterordnender Konjunktion	feels *like we have to move*
SBARQ	Direkte Frage beginnend mit *wh*-Wort oder *wh*-Phrase	*So what's that about?*
SINV	Invertierter deklarativer Satz	neither *am I a pessimist.*
SQ	Invertierte Ja/Nein Frage oder Hauptsatz einer *wh*-Frage	*Will they move on?*
ADJP	Adjektivphrase	*relatively cheap*
ADVP	Adverbphrase	*down here*
CONJP	Konjunktionalphrase	*but also for tissues*
FRAG	Fragment	if *not today, ...*
INTJ	Zwischenruf	*Well*
LST	Listenmarkierung	*1*
NAC	Keine Komponente	via the *Freedom of Information*
NP	Nominalphrase	*the sun*
NX	Markiert Kopf in komplexen NP	fresh *apples and cinnamon*
PP	Präpositionalphrase	*in some way*
PRN	Nebenläufige Phrase	*..., bless his heart, ...*
PRT	Partikel	*up*
QP	Quantifizierende Phrase	*or two a day*
RRC	Reduzierter Relativsatz	titles *not presently in the collection*
UCP	Ungleich Koordinierte Phrasen	She flew *yesterday and on July 4th.*
VP	Verbalphrase	this *is my dog.*
WHADJP	*Wh*-Adjektivphrase	*how great you are*
WHADVP	*Wh*-Adverbphrase	*When* I see it
WHNP	*Wh*-Nominalphrase	*What* they've done
WHPP	*Wh*-Präpositional	*At which*
X	Unbekannt	*The more ..., the less ...*

Tab. 2.2: Penn-Treebank syntaktische Tags [Bie+95]

Regeln angewendet werden. Jede Menge an Terminalen, die mit endlichen Schritten aus dem Startsymbol abgleitet werden kann, zählt zur Sprache, welche von der Grammatik beschrieben wird. Eine Grammatik heißt kontextfrei, wenn jede linke Seite aus genau einem Nichtterminal besteht. [Hof15, Kapitel 4]
Durch die Eigenschaft, dass jedem Wort ein *POS-Tags* zugeordnet wird, kommen Terminale nur in Regeln der Form *POS-Tag* $\rightarrow$ *Terminal* vor.
Für die Verarbeitung unseres Beispielsatzes wurden unter anderem folgende Regeln verwendet:

$$
\begin{aligned}
S &\rightarrow NP\ ADVP\ VP\ . \\
NP &\rightarrow PRP\$\ NN \\
PRP\$ &\rightarrow My \\
NN &\rightarrow dog
\end{aligned}
$$

Da ein englischer Satz nicht unbedingt *S* als oberste Konstituente hat, sondern auch *SQ*, *SBAR* und andere möglich sind, muss in den Grammatiken ein zusätzliches Startsymbol eingeführt werden. Dieses wird zum Beispiel *ROOT* oder *TOP* genannt. Da dieses aber immer nur die oberste Konstituente enthält, wird einfachheitshalber in diesem Kapitel weggelassen. Stattdessen repräsentiert die oberste Konstituente das Startsymbol.
Leitet man aus dem Startsymbol eine Menge an Terminalen ab, ab jetzt ebenfalls Satz genannt, so kann man aus allen verwendeten Regeln einen Syntaxbaum, kurz Baum, zeichnen. Das Startsymbol wird Wurzel genannt und die Terminale Blätter. Alle anderen Nichtterminale dazwischen heißen Knoten. Die linke Seite einer Regel ist der Elternknoten aller Elemente auf der rechten Seite. [Hof15, Kapitel 4]
Für das Beispiel in Abbildung **??** ist der Syntaxbaum in Abbildung 2.2 dargestellt. Syntaxbäume sind äquivalent zur geklammerten Version.

Anhand des vollständigen Regelsatzes der Grammatik einer natürlichen Sprache, kann jedem grammatikalisch korrekten Satz dieser Sprache seine syntaktische Struktur zugewiesen werden. Es gibt für diverse natürliche Sprachen Sammlungen von annotierten Sätzen. Diese werden *Treebank* oder annotierter Korpus genannt. Aus den dort gespeicherten syntaktischen Strukturen kann unter anderem eine Grammatik abgeleitet werden. Dies funktioniert, indem, in jedem Syntaxbaum, die Eltern-Kind-Beziehungen als Regel formuliert wird [MJ09, Kapitel 10].
Ein bekannter annotierter Korpus der englischen Sprache ist die *Penn-Treebank*, aus welcher auch die hier verwendete Annotation stammt. Dieser Korpus wird vom Linguistic Data Consortium, mit Sitz in der Univertität von Pennsylvania, herausgegeben. [Con]
Im Rahmen des *Penn-Treebank-Projekts* wurden von 1989 bis 1992 über 4,5 Millionen Wörter der *Treebank* hinzugefügt. Die Texte hierfür stammen zu einem Großteil aus

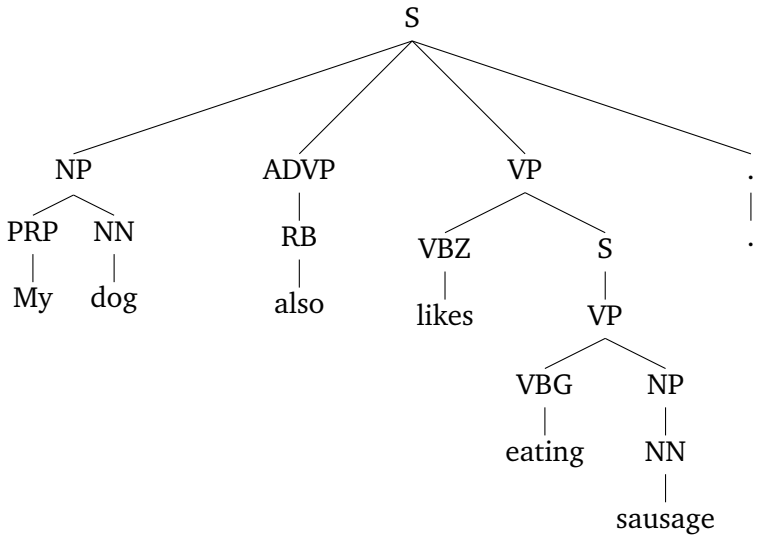

Abb. 2.2: Syntaxbaum zu *My dog also likes eating sausage.*

dem Wall Street Journal. Woher die Sätze einer *Treebank* stammen spielt eine große Rolle, da Parser mit Hilfe von *Treebanks* trainiert werden [Mar+93]. Dazu mehr in Kapitel 2.3.

2.2 Syntaktisches Parsen

Syntaktisches Parsen wird als die "Aufgabe des Erkennens eines Satzes und des Zuweisens einer syntaktischen Struktur"[MJ09, S. 461] definiert. Als Parser wird ein Programm bezeichnet, welches Sätze parst. Anders ausgedrückt, findet der Parser den Syntaxbaum der den Satz abdeckt.
Zum Einstieg in das Kapitel wird das Parsen als Suche betrachtet. Das Suchproblem besteht darin, aus allen möglichen Bäumen, welche sich mit der Grammatik generieren lassen, den korrekten Baum zur Eingabe zu finden. Der Suchraum wird von der Grammatik festgelegt. Ein Baum ist korrekt, wenn das Startsymbol die Wurzel ist und exakt die Eingabe abgedeckt wird. Vereinfachend wird angenommen, dass alle Sätze als oberste Konstituente *S* haben. Deshalb wird dieses als Startsymbol betrachtet. Anhand der zwei Merkmale eines korrekten Baumes kann die Suche gestaltet werden. Es ergeben sich als grundlegende Ansätze die Top-Down und die Bottom-Up Suche.
Beim Top-Down Verfahren wird mit dem Startsymbol *S* begonnen und dieses mit den Regeln der Grammatik Schritt für Schritt erweitert. Bäume, deren Blätter nicht auf die Eingabe passen, werden abgelehnt.
Die Bottom-Up Suche beginnt, dem Namen entsprechend, am anderen Ende des Baumes. Im ersten Schritt gibt es nur die Worte der Eingabe als Blätter. Es wird, mit den rechten Seiten der Grammatikregeln, der Baum von den Blättern zur Wurzel

gebaut. Damit gibt es für die Dauer der Suche im Allgemeinen mehrere Bäume gleichzeitig. Zum Beispiel, wenn aktuell für zwei Terminale die *POS-Tags* gefunden wurden. Mehrere Bäume werden Wald genannt. Die Konstellation des Waldes kann ausgeschlossen werden, wenn es für keine benachbarten Wurzeln keine rechte Seite einer Produktion gibt.
So werden mit der Top-Down Suche keine Bäume konstruiert, die niemals das Startsymbol als Wurzel haben. Dafür entsprechen die Blätter nicht exakt der Eingabe. Beim Bottom-Up Ansatz verhält es sich gegensätzlich.

2.2.1 Ambiguität

Das Hauptproblem, welches sich beim Finden des korrekten Baumes ergibt, ist die Mehrdeutigkeit oder auch genannt *Ambiguität*. Zum einen können Wörter mehrdeutig sein, wie etwa das englische Wort *book*, welches sowohl Verb als auch Nomen sein kann. Zum anderen, und für diesen Kontext relevanter, gibt es die strukturelle Mehrdeutigkeit. Ein Beispielsatz hierfür ist:

> I shot an elephant in my pajamas.

In dieser Satzstruktur kann sich *in my pajamas* sowohl auf den Erzähler, als auch auf den Elefanten beziehen und somit gibt es mehr als einen korrekten Baum.
Diese Art der strukturellen Mehrdeutigkeit ist die Anhangs-Mehrdeutigkeit, bzw. im Englischen *Attachment Ambiguity*. Eine Komponente des Satzes, in diesem Fall die Präpositionalphrase, kann an mehreren Stellen angehängt werden. Kombiniert man die Präpositional- an die Verbalphrase, hat der Satz die Bedeutung, dass der Schütze einen Pyjama trägt. Bindet man diese an das Nomen “Elefant” wird ausgesagt, dass dieser sich im Pyjama befindet. Grammatikalische Korrektheit ist in beiden Fällen gegeben.
Eine weitere Art ist die Koordinations-Mehrdeutigkeit, im Englischen *Coordination Ambiguity*, welche in Verbindung mit Konjunktionen und Satzverbindungen auftritt. Beispielsweise kann der Satzausschnitt *old men and women* unterschiedlich interpretiert werden. *Old* kann sich auf *men and women* oder nur auf *men* beziehen.
Bringt man beide Formen der Mehrdeutigkeit in einen Satz, wie z.B.

> I shot wild elephants and lions in my pajamas.

ergeben sich folgende vier Möglichkeiten (die Klammerung stellt die Zusammengehörigkeit dar):

> I shot [wild elephants] and [lions] in my pajamas.
> I shot [wild elephants and lions] in my pajamas.

I shot [wild elephants] and [lions in my pajamas].
I shot [[wild elephants and lions] in my pajamas].

In den ersten beiden Versionen bezieht sich *in my pajamas* auf *shot*, danach auf *lions*, bzw. *wild elephants and lions*. Auch *wild* wechselt zwischen einem und zwei Bezugswörtern. Die an unterschiedlichen Syntaxbäumen für einen Satz kann also exponentiell wachsen. Von diesen Interpretationen beschreibt nur eine den Inhalt des Satzes so, wie er vom Autor beabsichtigt wurde. Ein Parser braucht weitere Kriterien, um sich zwischen den verschiedenen Optionen entscheiden zu können. Hierzu mehr in Kapitel 2.3. Ohne zusätzliche Informationen kann der Parser lediglich alle grammatikalisch möglichen Bäume erstellen. Um dieses Problem effizient zu lösen bietet sich dynamisches Programmieren an. [MJ09, Kapitel 11]

2.2.2 Dynamische Programmierung

Für das Finden aller Syntaxbäume eines Satzes bietet sich dynamisches Programmieren an. Das Konzept der dynamischen Programmierung ist beispielsweise beschrieben in [GT11].
Der Einsatz dieser Technik empfiehlt sich, da beim Erstellen des Baumes im Allgemeinen Mehrfacharbeit anfällt. Konstruiert der Parser zum Beispiel die Bäume von der Wurzel zu den Blättern und versucht dabei die Eingabe von links nach rechts abzudecken, dann findet er in der Regel einen Baum, der einen Teil des Satzes abdeckt. Da noch nicht die gesamte Eingabe enthalten ist, handelt es sich nicht um einen korrekten Baum. Dennoch wird dieser in der Regel öfters konstruiert, bzw. ist in der tatsächlichen Lösung enthalten. Dass dieser Teilbaum nicht jedes mal neu berechnet werden muss, bietet es sich an ihn abzuspeichern. Dieses Speichern von Zwischenlösungen ist ein Kernkonzept der dynamischen Programmierung. Als Algorithmen für das Parsen mittels dynamischer Programmierung haben sich das *Chart Parsing*, der *Earley*- und der *Cocke-Kasami-Younger*-Algorithmus (kurz *CKY*) etabliert. Letzterer wird nachfolgend ausführlicher präsentiert. [MJ09, Kapitel 11]

2.2.3 Cocke-Kasami-Younger Algorithmus

Zu Beginn des Algorithmus muss die verwendete Grammatik in die *Chomsky-Normalform* (kurz *CNF*) überführt werden. Die *CNF* zeichnet sich dadurch aus, dass die rechte Seite aus genau zwei Nichtterminalen oder einem Terminal besteht. Das Überführen einer kontextfreien Grammatik in diese Form erfolgt nach einem einfachen Regelsatz und wird hier nicht näher erläutert. Auch die Rückführung der Grammatik in *CNF* in die ursprüngliche Grammatik ist mit zusätzlich abgespeicherten

Informationen möglich. Diese Rückführung wird gemacht, da die Ergebnisbäume der *CNF*-Struktur und nicht der eigentlichen Grammatik entsprechen. In *CNF*-Struktur wäre auch eine semantische Analyse komplizierter.
Nach der Überführung in die *CNF* wird für einen Satz mit *n* Wörtern eine obere Dreiecksmatrix der Größe $(n+1) \times (n+1)$ aufgestellt. Die Positionen im Satz werden mit 0 beginnend durchnummeriert. Vor dem ersten Wort steht Index 0, nach dem letzten Index *n*. Die Zelle $[i, j]$ enthält diejenigen Nichtterminale, welche die Eingabe von *i* bis *j* abdecken. Dabei gilt $0 \leq i < n-1$, $1 \leq j < n$ und $i < j$. Die Zelle $[0, n]$ gibt somit an, welche Nichtterminale die komplette Eingabe abdecken.
Die Zellen $[i, i+1]$ bilden die Diagonale der Matrix. Hier wird die Eigenschaft der *CNF* ausgenutzt, dass alle Terminale auf der rechten Seite der Produktionen alleine stehen und somit jedes von mindestens einem Nichtterminal abgedeckt wird. Diese Situation war durch das Verhältnis *POS-Tags* zu Terminalen ohnehin gegeben.
Die Eigenschaft, dass Nichtterminale auf der rechten Seite immer zu zweit auftreten führt dazu, dass für die eine Zelle $[i, j]$ folgendes gilt. Für jede Regel $A \rightarrow B\ C$ die *i* bis *j* abdeckt, gibt es ein *k*, mit $i < k < j$, sodass das *B* den Abschnitt $[i, k]$ und *C* $[k, j]$ abdeckt. Ersteres befindet sich in der selben Zeile, links von der entsprechenden Zelle. Letzteres ist in der selben Spalte, unterhalb zu finden. Das Füllen der Matrix muss von links nach rechts und von unten nach oben geschehen.
Eine gefüllte Matrix für den Satz *Book the flight through Houston* ist in Abbildung **??** dargestellt. Hierfür wurde eine sehr kleine Grammatik verwendet die nur knapp mehr als die verwendeten Wörter abdeckt. Die Grammatik und die Abbildung sind entnommen aus [MJ09, Kapitel 11] Neben der *CNF* Regeln sind auch die ursprünglichen dargestellt.

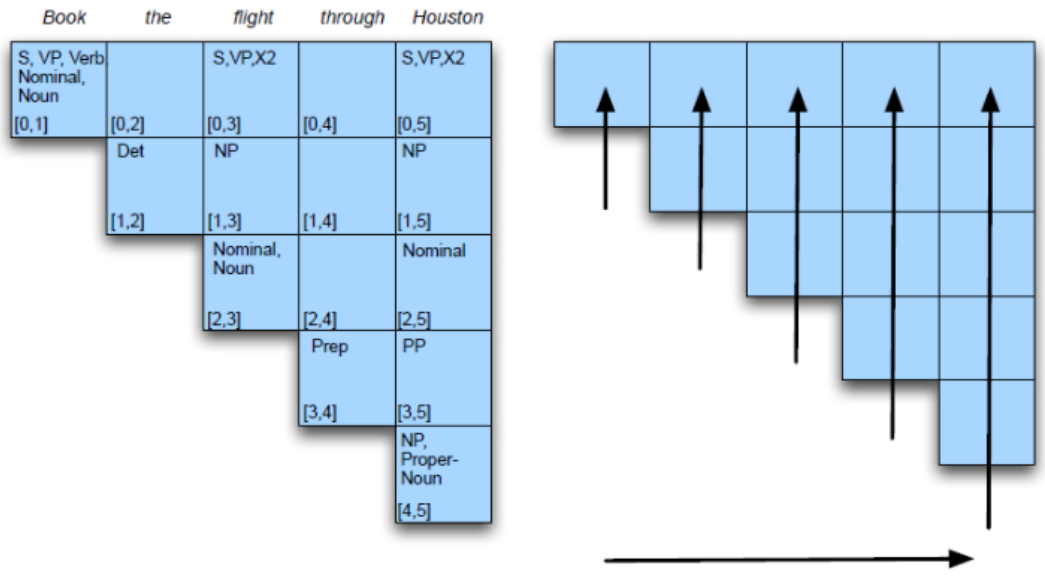

Abb. 2.3: Beispielmatrix für den *CKY*-Algorithmus, entnommen aus ..

$\mathscr{L}_1$ Grammar	$\mathscr{L}_1$ in CNF
$S \rightarrow NP\ VP$	$S \rightarrow NP\ VP$
$S \rightarrow Aux\ NP\ VP$	$S \rightarrow X1\ VP$
	$X1 \rightarrow Aux\ NP$
$S \rightarrow VP$	$S \rightarrow book \mid include \mid prefer$
	$S \rightarrow Verb\ NP$
	$S \rightarrow X2\ PP$
	$S \rightarrow Verb\ PP$
	$S \rightarrow VP\ PP$
$NP \rightarrow Pronoun$	$NP \rightarrow I \mid she \mid me$
$NP \rightarrow Proper\text{-}Noun$	$NP \rightarrow TWA \mid Houston$
$NP \rightarrow Det\ Nominal$	$NP \rightarrow Det\ Nominal$
$Nominal \rightarrow Noun$	$Nominal \rightarrow book \mid flight \mid meal \mid money$
$Nominal \rightarrow Nominal\ Noun$	$Nominal \rightarrow Nominal\ Noun$
$Nominal \rightarrow Nominal\ PP$	$Nominal \rightarrow Nominal\ PP$
$VP \rightarrow Verb$	$VP \rightarrow book \mid include \mid prefer$
$VP \rightarrow Verb\ NP$	$VP \rightarrow Verb\ NP$
$VP \rightarrow Verb\ NP\ PP$	$VP \rightarrow X2\ PP$
	$X2 \rightarrow Verb\ NP$
$VP \rightarrow Verb\ PP$	$VP \rightarrow Verb\ PP$
$VP \rightarrow VP\ PP$	$VP \rightarrow VP\ PP$
$PP \rightarrow Preposition\ NP$	$PP \rightarrow Preposition\ NP$

Abb. 2.4: Grammatik für entnommen aus ..

Ohne zusätzliche Informationen gibt die Tabelle nur an, ob ein Satz grammatikalisch korrekt ist. Ein Satz ist korrekt, wenn in Zelle $[0, n]$ das Startsymbol steht. Um die verwendeten Regeln, also den Baum, bzw. die Bäume zur Eingabe zu erhalten, muss der Algorithmus erweitert werden. Die Erweiterung besteht darin, zu jedem Nichtterminal in jeder Zelle zusätzlich zu speichern, aus welchen zwei anderen Nichtterminalen es entstanden ist. Darüber hinaus darf das gleiche Nichtterminal mehrmals in einer Zelle vorkommen, da es aus unterschiedlichen Nichtterminalen entstanden sein kann. Ein Beispiel zur Tabelle in Abbildung **??** ist in Zelle $[0, 5]$ das Symbol S. Es gibt drei Möglichkeiten dieses zu erzeugen:

$$\begin{aligned} &[0,1] : Verb\ und\ [1,5] : NP \\ &[0,3] : VP\ und\ [3,5] : PP \\ &[0,3] : X2\ und\ [3,5] : PP \end{aligned} \tag{2.1}$$

Damit ergeben sich für die Eingabe drei verschiedene Syntaxbäume. Diese werden erstellt, indem man die Tabelle, beginnend mit dem jeweiligen Startsymbol, rückwärts durchläuft. [MJ09, Kapitel 11]

2.3 Statistisches Parsen

In diesem Abschnitt werden Mittel vorgestellt, welche dem Parser helfen sich zwischen mehreren, grammatikalisch korrekten Bäumen eines Eingabesatzes zu entscheiden. Hierfür benötigt der Parser für jeden errechneten Baum zusätzlich die Wahrscheinlichkeit, mit welcher dieser semantisch korrekt ist. Das heißt, jede Lösung ist mit einer Wahrscheinlichkeit versehen und es kann diejenige, deren Wert am höchsten ist als Lösung gewählt werden.

2.3.1 Probabilistische Kontextfreie Grammatiken

Die Anforderung, einem Baum eine Wahrscheinlichkeit zuzuweisen, lässt sich mit dem Konzept der probabilistischen kontextfreien Grammatiken (kurz *PCFG*) umsetzen. Abgesehen von zwei Erweiterungen haben die Produktionen einer *PCFG* die gleiche Form wie die der ursprünglichen kontextfreien Grammatik. Erstens wird jeder Regel eine Wahrscheinlichkeit p, mit $0 \leq p \leq 1$, hinzugefügt.

$$A \rightarrow \beta[p] \tag{2.2}$$

Diese gibt an, mit welcher Wahrscheinlichkeit die rechte Seite unter der Vorbedingung der linken Seite auftritt.

$$p := P(A \rightarrow \beta|A) \tag{2.3}$$

Es handelt sich um eine bedingte Wahrscheinlichkeit.
Zweitens muss für jedes Nichtterminal die Summe der Wahrscheinlichkeiten aller seiner Produktionen eins ergeben.

$$\sum_{\beta} P(A \rightarrow \beta) = 1 \tag{2.4}$$

Die Wahrscheinlichkeit für einen Baum errechnet sich dann durch das Produkt der Wahrscheinlichkeiten aller verwendeten Regeln. Eine Grammatik ist konsistent, wenn die Summe der Wahrscheinlichkeiten aller möglichen Sätze eins ergibt. Inkosistenz tritt auf, wenn es eine Regel der Form $A \rightarrow A$ gibt.
Mit dieser neuen Art von Grammatik ergeben sich auch neue Algorithmen zum Berechnen der Bäume. Sowohl der *CKY*- als auch der *Earley*-Algorithmus sind um den Faktor der Wahrscheinlichkeit erweiterbar, wobei der *CKY*-Algorithmus mehr Verwendung findet.
Der in Kapitel 2.2.3 vorgestellte *CKY*-Algorithmus wird dahingehend erweitert, dass in jeder Zelle für jedes Nichtterminal die Wahrscheinlichkeit gespeichert wird. Damit erhält man eine dreidimensionale $(n+1) \times (n+1) \times V$ Matrix, wobei V die Anzahl

an Nichtterminalen in der Grammatik, und n wie bisher die Anzahl an Wörtern im Satz, ist.
Einen Nutzen kann man aus der zugefügten Wahrscheinlichkeit nur dann ziehen, wenn ihr numerischer Wert Sinn ergibt. Um diesen Wert für jede Regel zu berechnen, gibt es zwei Möglichkeiten. Zum einen kann dieser aus einer vollständig annotierten *Treebank* errechnet werden. Hierfür wird jedes Auftreten einer Regel und des entsprechenden Nichtterminals gezählt und dividiert:

$$P(A \to \beta) = \frac{Anzahl(A \to \beta)}{\sum_{\gamma} Anzahl(A \to \gamma)} = \frac{Anzahl(A \to \beta)}{Anzahl(A)} \tag{2.5}$$

Die Grammatik wird also anhand der annotierten Sätze der *Treebank* trainiert. Die verwendeten Sätze werden Trainingsdaten genannt.
Falls man keine solche *Treebank* zur Verfügung hat, gibt es eine zweite Möglichkeit die Werte der *PCFG* festzulegen. Hierzu arbeitet der Parser einen unannotierter Textkorpus durch und fügt die entsprechenden *Tags* ein. Zu Beginn haben alle Produktionen eines Nichtterminals die selbe Wahrscheinlichkeit. Der Korpus wird iterativ durchlaufen und nach jedem Durchgang werden die Wahrscheinlichkeiten der Regeln angepasst. Das Anpassen passiert wie in der ersten vorgestellten Möglichkeit, da zu diesem Zeitpunkt eine annotierte *Treebank* vorhanden ist. Das Verfahren endet, wenn die Wahrscheinlichkeiten konvergieren.
Die resultierenden Werte in der Grammatik hängen davon ab, mit welchem Korpus sie berechnet wurden. Hierbei spielen Größe und Textart eine große Rolle. Um beim Parsen eines Textes möglichst gute Ergebnisse zu erhalten, sollte der Parser eine *PCFG* verwenden, welche mit einem Text des selben Genres erstellt wurde. So kann man beispielsweise mit einer *Treebank* aus technischen Handbüchern, unabhängig von ihrer Größe, beim Parsen eines privaten Briefes keine guten Ergebnisse erwarten, da sich die Sprache zu sehr unterscheidet.
Außerdem weist dieses neue Konzept auch zwei Nachteile auf. Das erste Problem der *PCFG* ergibt sich aus der Kontextfreiheit. Die Wahrscheinlichkeit einer Produktion ist immer gleich, egal an welcher Stelle im Satz sie auftritt. In einer natürlichen Sprache ist das aber im Allgemeinen nicht der Fall. Beispielweise kann die gewählte Produktion einer Nominalphrase abhängig davon sein, ob die Phrase Objekt oder Subjekt des Satzes ist. Da diese Information ohne Weiteres nicht in der Grammatik berücksichtigt werden kann, muss der Mittelwert aus beiden Fällen gebildet werden. Dies führt dazu, dass entweder im Falle des Objekts oder des Subjekts häufig die falsche Regel angewendet wird.
Die zweite Schwachstelle ergibt sich daraus, dass die einzelnen Wörter eine zu kleine Rolle spielen. Als Beispiel hierfür dient die bereits erklärte Anhangs-Mehrdeutigkeit aus Kapitel 2.2. Wiederum wird eine Präpositionalphrase betrachtet, welche ent-

weder an eine Verbal- oder eine Nominalphrase angebunden wird. An welche von beiden hängt allein von der *Treebank* ab. Dort kommt entweder die Produktion

$$VP \rightarrow \alpha \;\; NP \;\; PP$$

oder die Kombination aus

$$VP \rightarrow \alpha \;\; NP$$

und

$$NP \rightarrow NP \;\; PP$$

öfter vor. α steht für eines der hier möglichen Verb-Nichtterminale, welches aber in beide Fällen immer das selbe ist und deswegen keine Rolle spielt. Wesentlich bessere Resultate sind hier erzielbar, wenn man das Verb aus *VP*, das Nomen aus *NP* und die Präposition aus *PP* in die Entscheidungsfindung mit einbezieht. Es kann aus der *Treebank* die Information gewonnen werden, ob die gegebene Präposition sich öfter auf das Nomen oder das Verb bezieht. Angenommen es gibt eine Präposition, die ausschließlich mit Verben in Verbindung steht. In der *Treebank* sind es nun aber die Nominalphrasen, an welche öfter Präpositionalphrasen gebunden werden. Dann wird die eben angenommene Präposition mit einer *PCFG*, welche mit der *Treebank* errechnet wurde, immer falsch zugeordnet.
Diese Schwäche der *PCFG* macht sich ebenso bei Koordinations-Mehrdeutigkeit bemerkbar. Beim Verbinden von Phrasen durch Konjunktionen kann wieder die konkrete Konjunktion und die Beziehung zu den entsprechenden Wörtern der zu verbindenden Phrasen betrachtet werden. Hierzu dient wieder das Beispiel aus Abschnitt 2.2: *old men and women*. Es könnte eine *Treebank* die Information liefern, dass die Wörter *men* und *women* per *and* öfter direkt miteinander verbunden werden, als dass nur eines von beiden das Adjektiv *old* zugeordnet bekommt.
Eine mögliche Verbesserung der *PCFG* ergibt sich durch das Erweitern der Nichtterminale um die Information wessen Kind es ist. Es wird ein *NP* als *NP^S* geschrieben, wenn *S* der Elternknoten ist oder als *NP^VP*, wenn es *VP* ist. Hierdurch ergeben sich zwei neue Regeln in der Grammatik und damit auch zwei neue Wahrscheinlichkeiten. Mit diesem Konzept kann dem Nichtterminal, obwohl die Kontextfreiheit im grammatikalischen Sinne nicht verletzt wird, ein Kontext gegeben werden. Allerdings wird, wenn jedes Nichtterminal für jeden möglichen Elternknoten eine neue Produktion erhält, die Grammatik enorm aufgeblasen. Nebeneffekt dieser neuen Größe ist, dass bei gleich bleibender *Treebank* für jede Regel weniger Trainingsdaten zur Verfügung stehen und damit die erhaltenen Wahrscheinlichkeitswerte ungenauer sind. Für einen gegebenen Korpus muss ein Split-and-Merge Algorithmus ausgeführt werden, der errechnet wie weit eine Aufteilung der Nichtterminale sinnvoll ist. [MJ09, Kapitel 12]

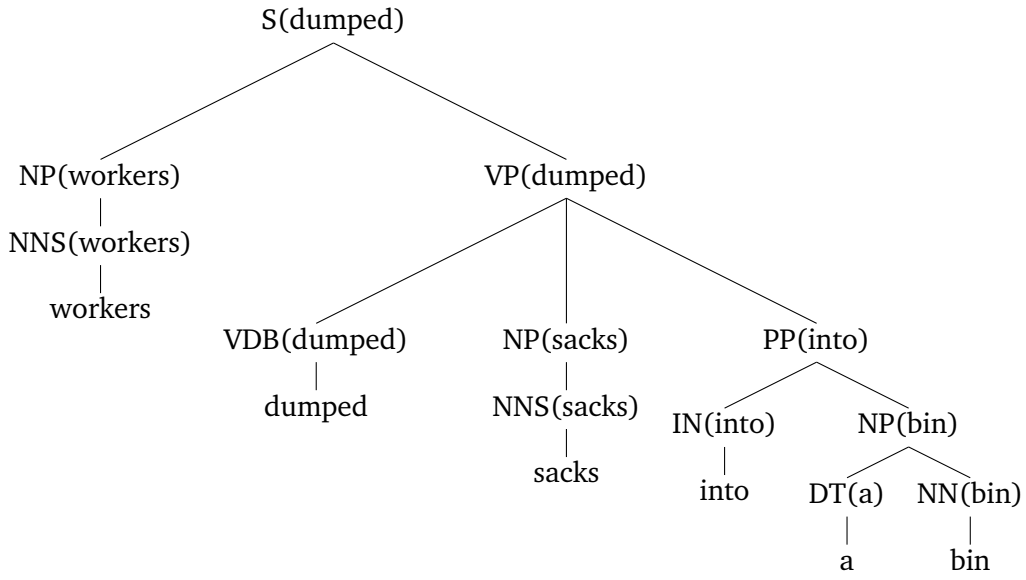

Abb. 2.5: Lexikalisierter Baum, entnommen aus

2.3.2 Probabilistische Lexikalisierte Kontextfreie Grammatiken

Ein weiterer Ansatz, um bessere Parserergebnisse zu erhalten, ist das Lexikalisieren der Grammatik. Hierfür muss der Begriff des Kopfes, im Englischen *Head*, eingeführt werden. Die Idee ist, dass jede syntaktische Einheit einen Kopf in Form eines Wortes aus dieser Einheit besitzt. Es wird das Wort genommen, welches "im Satz am grammatikalisch wichtigsten ist" [MJ09, S. 443]. Da jedes Nichtterminal einer syntaktischen Einheit oder einem POS-Tag entspricht, kann jedem Nichtterminal und jeder Produktion der Grammatik ein Kopf zugewiesen werden. Für das Finden des richtiges Wortes gibt es verschiedene Schritte. Begonnen wird, indem jedes POS-Nichtterminal sein entsprechendes Kind als *Head* wählt. Dieses Wort wird dann im Baum nach oben weitergegeben. Für jede syntaktische Einheit gibt es Regeln, anhand derer einer der Köpfe der Kinder ausgewählt und als eigener eingesetzt wird. Ein Beispiel ist der lexikalisierte Baum für den Satz *workers dumped sacks into a bin*, dargestellt in Abbildung **??**.

Zusätzlich zum Kopfwort kann auch noch dessen POS-Tag abgespeichert werden. Somit wird beispielsweise *S(dumped)* zu *S(dumped, VBD)* und *NNS(workers)* zu *NNS(workers, NNS)*. Die Wahrscheinlichkeit der Regel

$$VP(dumped, VBD) \rightarrow VBD(dumped, VBD)\ \ NP(sacks, NNS)\ \ PP(into, IN) \tag{2.6}$$

ergibt sich wieder aus dem Inhalt der *Treebank*, nämlich durch die Formel

$$\frac{Anzahl(VP(dumped, VBD) \rightarrow VBD(dumped, VBD)\ NP(sacks, NNS)\ PP(into, IN))}{Anzahl(VP(dumped, VBD))} \tag{2.7}$$

Dieser Wert ist null, wenn der Korpus keinen Satz mit *dumped sacks into* enthält. Existiert kein Satz in welchem sich *VP(dumped, VBD)* finden lässt, so ist dieser Wert nicht definiert.
Aufgrund dieses Problems bedarf es anderer Berechnungsvorschriften, wie zum Beispiel *Collins Model 1*, vorgestellt in [Col97]. Es wird jede Produktion folgendermaßen betrachtet: *H* ist der Kopf, L_i sind die Nichtterminale links und R_i die rechts davon. Alle Nichtterminale bleiben weiterhin lexikalisiert. Das Kopfwort wird mit *h* bezeichnet. Damit ergibt sich die Form

$$A \rightarrow L_n ... L_1 H R_1 ... R_m$$

Zusätzlich wird an der Stelle L_{n+1} und R_{m+1} das Nichtterminal *STOP* eingefügt um anzuzeigen, dass hiernach die Regel zu Ende ist. In drei Schritten wird die Wahrscheinlichkeit der Produktion errechnet:

1. Es wird der Kopf mit der Wahrscheinlichkeit $P_H(H|A, h)$ generiert.

2. Alle Elemente rechts vom Kopf werden mit $\prod_{i=1}^{m+1} P_R(R_i|A, h, H)$, also einschließlich dem *STOP*, generiert.

3. Alle Elemente links von *H* werden mit $\prod_{i=1}^{n+1} P_L(L_i|A, h, H)$ generiert.

Für die Regel 2.6 errechnet sich die Wahrscheinlichkeit über

$$\begin{aligned} P = & P_H(VBD|VP, dumped) \\ & \times P_R(NP(sacks, NNS)|VP, dumped, VBD) \\ & \times P_R(PP(into, IN)|VP, dumped, VBD) \\ & \times P_R(STOP|VP, dumped, VBD) \\ & \times P_L(STOP|VP, dumped, VBD) \end{aligned} \tag{2.8}$$

Im Gegensatz zu vorher muss nicht die Kombination aus *NP(sacks, NNS)*, *PP(into, IN)* und *VBD(dumped, VBD)* vorhanden sein. Es genügt, wenn jedes einzeln, als Kind von *VP(dumped, VBD)* auf der entsprechenden Seite des *Heads* in der *Treebank* zu finden ist. Darüber hinaus wird das Modell um die Information der Distanz erweitert. Es wird zusätzlich berücksichtigt, wie weit ein Nichtterminal vom Kopf der Regel entfernt ist. Der Vollständigkeit halber ist außerdem zu erwähnen, dass es neben

Model 1 noch zwei weitere Modelle gibt. Diese Modelle bilden die Grundlage für den *Collins Parser*. [Col97][MJ09, Kapitel 12]

3

Konzept des Evaluations-Rahmenwerks

In diesem Kapitel wird das Konzept vorgestellt, nach dem das Evaluations-Rahmenwerk entwickelt wurde. In Abbildung **??** wird dargestellt, wie die Leistung eines Parsers evaluiert wird. Der Input-Text ist der rohe Text, den der Parser mit syntaktischer Annotation versehen soll. Das Parser-Modell ist die Grundlage auf der ein Parser arbeitet. Der *Goldstandard* ist der Input-Text mit - als korrekt angesehener - Annotation. Dieser wird mit der Ausgabe des Parsers verglichen. Daraus lassen sich Kennzahlen zur Leistung des Parsers errechnen, die am Ende ausgegeben werden. Um mehrere Parser miteinander zu vergleichen, werden die Resultate der einzelnen in einer Tabelle zusammengefasst und dargestellt. Alle Elemente werden nachfolgend ausführlich vorgestellt.

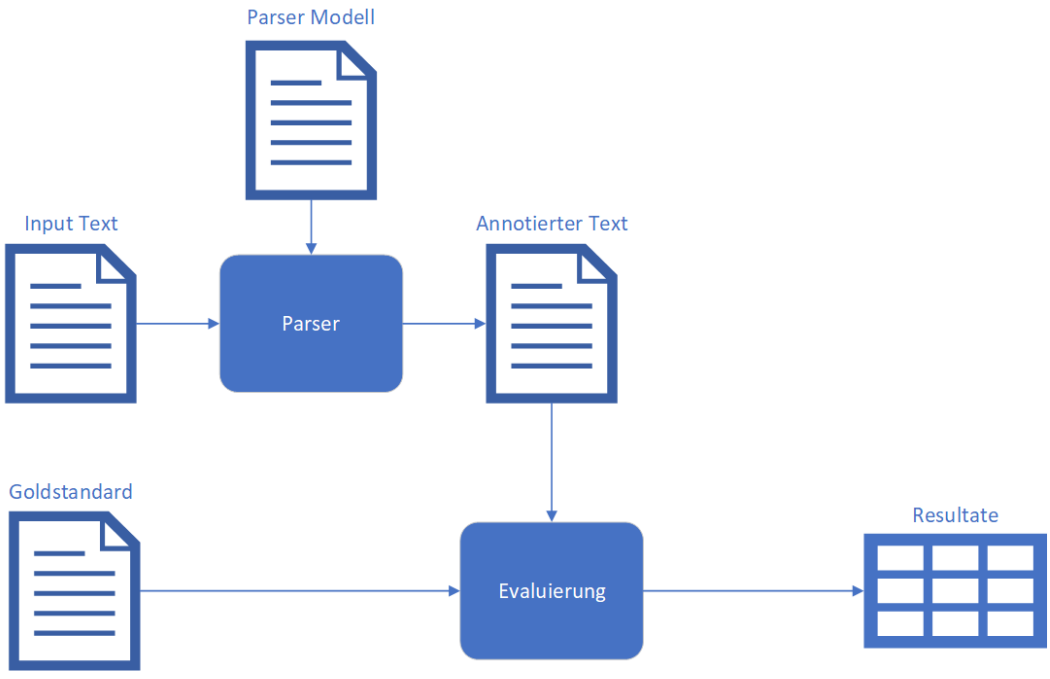

Abb. 3.1: Konzeptueller Aufbau

3.1 Input Text

Die Eingabe des Parser ist nicht der eigentliche natürlich-sprachliche Text. Ein Element des Satzes, dem ein *POS-Tag* zugewiesen wird, heißt *Token*. In den Eingabesätzen müssen bereits alle *Token* durch Leerzeichen getrennt. Ein solcher Satz wird *sequenziert* genannt. In den meisten Fällen haben die *Token* bereits die benötigten Leerzeichen vor und hinter sich. Ein Beispiel, indem das nicht der Fall ist, ist *he's*. Hier erkennt der Parser nur dass es sich um die Wörter *he* und *is* handelt, wenn ein Leerzeichen vor dem Apostroph eingeschoben wird. Bekommt man aus einem Korpus keine Version des Satzes, bei dem dieser Arbeitsschritt schon erledigt ist, muss man einen *Tokenizer* vorschalten. Das ist ein Programm, welches Eingabesätze sequenziert. Hierfür gibt es unterschiedliche Anbieter, wie zum Beispiel Stanford [Sta] oder OpenNLP [Ope]. Allerdings wird in der Implementierung kein *Tokenizer* verwendet.
Die Sätze der Eingabe müssen über ein Textdokument übergeben und zeilenweise getrennt sein. Der Parser erstellt für jede Zeile einen Syntaxbaum.

3.2 Parser Output

Als Ausgabe gibt der Parser die annotierten Sätze in der Reihenfolge, in der sie eingegeben wurden, zurück. Es wird hierfür wieder ein Textdokument erstellt. Beispielsweise lautet die annotierten Version des Satzes

> It goes 150 miles an hour .

```
( (S (NP (PRP It)) (VP (VBZ goes) (NP (NP (CD 150) (NNS miles))
(NP (DT an) (NN hour)))) (. .)) )
```

Das äußerste Klammerpaar hat kein führendes Nichtterminal und könnte deshalb weggelassen werden. Es enthält typischerweise das Startsymbol der Parser, also *ROOT*, *TOP* oder ähnliches. Dieses Nichtterminal wird aber für die Evaluierung aus dem Ergebnisbaum herausgenommen, weil es keine syntaktische Information liefert. Da der Satz des *Goldstandards* ebenfalls diese unannotierten Klammern besitzt, werden sie aus beiden Bäumen nicht gelöscht sondern im Evaluationsschritt ignoriert.

3.3 Parser Modell

Das Modell des Parser ist in der Regel eine lexikalisierte oder unlexikalisierte *PCFG*. Die Wahrscheinlichkeitswerte einer *PCFG* sind ausschlaggebend für die Leistung der Parser. Deshalb erzielt ein Parser, je nachdem welches Modell ihm zu Grunde liegt, sehr unterschiedliche Ergebnisse. Aus diesem Grund ist dieses Modell nicht fest im Parser verankert, sondern wird ihm als Datei übergeben. Somit kann man selbst Modelle anhand einer *Treebank* erstellen lassen um dann beispielsweise auf einem Parser die unterschiedlichen Modelle zu vergleichen. Alle verwendeten Parser bringen ein trainiertes Modell mit.

3.4 Goldstandard

Als *Goldstandard* werden die korrekten Bäume bezeichnet. Diese wurden entweder von Menschenhand erstellt oder durch einen Parser produziert und dann von Menschen nachkorrigiert. Es muss für ein eingegebenes Textdokument, welches vom Parser bearbeitet werden soll auch ein Textdokument geben, das für jeden dieser Sätze die annotierte Version enthält. Die gratis verfügbaren *Treebanks* weisen oft ein unterschiedliches Sortiment an Dateien und Formaten auf.
Der in dieser Arbeit verwendete Korpus heißt “The NAIST-NTT TED Talk Treebank” [Neu+14] und stammt aus dem Jahr 2014. Er umfasst etwa 23.000 Wörter in 1.200 Sätzen, welche aus 10 Reden gewonnen wurden. Zur Annotierung wurde der Berkeley Parser verwendet und dessen Resultat per Hand verbessert. Als *Tagset* wurde die Vorlage der *Penn Treebank* verwendet. In dieser *Treebank* sind für jeden Satz eine Rohversion, eine sequenzierte und eine annotierte Version enthalten. Rohversion ist der unveränderte, natürlich-sprachliche Satz. Da es sich bei den Texten um Gesprochenes handelt, sind noch weitere Informationen, wie zum Beispiel Zeitmessung eines Satzes und ähnliches vorhanden. Das wird hier nicht weiter verwendet.
Bei Verwendung anderer *Goldstandards* können diverse Probleme auftreten. Zwei davon sind hier kurz durchdacht aber nicht implementiert worden.
Falls das Problem auftritt, dass nur die korrekten Lösungsbäume zur Eingabe verfügbar sind, so kann man sich zum Beispiel mit dem Python Toolkit NLTK [Nlt] die Blätter jedes Lösungsbaums ausgeben lassen. Hierdurch erhält man, den in seine Token aufgeteilten Satz.
Ist der Lösungsbaum nicht einzeilig, sondern erstreckt sich ähnlich zu Abbildung **??** über mehrere Zeilen, so muss hier auch eine Vorverarbeitung geschehen. Entweder kann anhand der Klammerung erkannt werden wo die Grenzen des Baumes liegen oder es müssen alle Zeilen, die mit einer Art Leerzeichen (Tabulator, u.ä.) beginnen, an die vorherige gehangen werden.

3.5 Parser

Alle Parser erhalten die Eingabe in selber Form, allerdings kann jeder Parser noch eine individuelle Vorverarbeitung benötigen. Dazu mehr in Abschnitt 4.3. Im Rahmen dieser Arbeit sind drei Parser verglichen worden. Das sind der *Stanford-Parser* [KM03], der *Berkeley-Parser* [Pet+06] und der Parser aus dem *OpenNLP Paket* [Ope]. Jeder verwendet zum Annotieren die *Penn Treebank Tags* und liefert die Ausgabe standardmäßig ohne die zusätzlichen relationalen *Tags*.

3.6 Evaluierung

Für die Evaluierung wird das Ergebnis der Parser zeilenweise mit dem *Goldstandard* verglichen. Zu diesem Zweck werden die Konstituenten der Bäume betrachtet. Eine Konstituente beschreibt einen Knoten im Baum und enthält die Information über das Nichtterminal, den Start- und den Endpunkt. Die Punkte geben den Index des ersten und letzten Wortes dieses Teilbaums an. Mit jedem Wort wird der Zähler inkrementiert. Für den Beispielsatz *They learn much faster.* (Baum in Abbildung 3.2) sind die Konstituenten in Tabelle 3.1 dargestellt.

Falls es, für eine Konstituente des Parsers, im *Goldstandard* eine mit gleichem

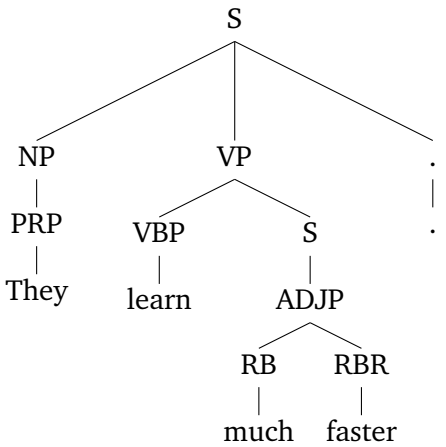

Abb. 3.2: Syntaxbaum zu *They learn much faster.*

Nichtterminal und gleicher Spanne gibt, wird diese als richtig bewertet. Hier muss bei der Ausführung entschieden werden, welche Nichtterminale gewertet werden. Zum einen kann man alle in das Resultat einbeziehen. Zum anderen können die *POS-Tags* herausgelassen und nur die syntaktischen *Tags* aus Tabelle 2.2 bewertet werden. Der Grund ist, dass man so Parser einbringen kann, die als Eingabe einen mit *POS-Tags* versehenen Text bekommen. Diese *Tags* verfälschen dann die Korrektheitsrate der Parser und müssen ignoriert werden. Da in dieser Arbeit ausschließlich Parser genutzt werden, die als Eingabe sequenzierten Text bekommen, betrachten

Nichtterminal	Text	Start	Ende
S	They learn much faster .	0	5
NP	They	0	1
PRP	They	0	1
VP	learn much faster	1	4
VBD	learn	1	2
S	much faster	2	4
ADJP	much faster	2	4
RB	much	2	3
RBR	faster	3	4
.	.	4	5

Tab. 3.1: Konstituenten zu Abbildung 3.2

wir hier den Fall in dem alle *Tags* relevant sind. Der Evaluierungsmechanismus ist angelehnt an [KT07], bzw. [Bla+91]. Als Bewertungsmetriken werden folgende vier Werte errechnet: *Precision*, *Recall*, F_1 und *Relative Cross Brackets* (kurz: RCB). Siehe Formeln (3.1) bis (3.4).

$$Precision = \frac{\#\ korrekte\ \ Konstituenten}{\#\ \ Konstituenten\ \ im\ \ Parseroutput} \tag{3.1}$$

$$Recall = \frac{\#\ korrekte\ \ Konstituenten}{\#\ \ Konstituenten\ \ im\ \ Goldstandard} \tag{3.2}$$

$$F_1 = \frac{2 \times Precision \times Recall}{Precision + Recall} \tag{3.3}$$

$$RCB = \frac{\#\ Parser\ Konst.\ die\ Goldstandard\ Konst.\ kreuzen}{\#\ \ Konst.\ \ im\ \ Parseroutput} \tag{3.4}$$

Die *Precision*, im Deutschen *Genauigkeit*, gibt an, wieviele der Konstituenten des Parsers korrekt sind. *Recall*, oder *Trefferquote*, beschreibt zu welchem Teil die Konsituenten des *Goldstandards* mit den Parser-Konstituenten abgedeckt wurden. F_α ist das gewichtete harmonische Mittel aus beiden. Im Rahmen dieser Arbeit wurde nur F_1 betrachtet, d.h. beide werden gleich gewichtet. Beim *RCB*-Wert handelt es sich um einen Indikator, der ausschließlich für Syntaxbäume eingesetzt wird und vorgestellt wurde in [Bla+91]. Er gibt an, wieviele der Konstituenten des Parsers sich mit denen des Standards kreuzen. Kreuzen ist in diesem Kontext folgendermaßen definiert: Seien *k1* und *k2* zwei Konstituenten mit unterschiedlichen Start- und Endpunkten. Außerdem gilt o.B.d.A., dass *k1* den niedrigeren Startpunkt hat. Dann kreuzen sie, falls der Endpunkt von *k1* größer als der Startpunkt und kleiner als der Endpunkt von *k2* ist. Anders ausgedrückt, sie sind nicht ineinander enthalten aber teilen sich

mindestens ein Wort. Aus der Definition ergibt sich, für *k1* aus dem Parser und *k2* aus dem *Goldstandard*, dass *k1* nicht korrekt sein kann. Angenommen es wäre korrekt, so müsste es ein *k1'* im *Goldstandard* mit identischen Grenzen geben. *k1'* beginnt vor *k2* und hört auch vor diesem auf und kann somit kein Eltern- oder Kindknoten von diesem sein. Da sie sich aber mindestens ein Wort teilen würde für dieses Wort gelten, dass es Blatt von zwei verschiedenen Teilbäumen wäre, also zwei Eltern hätte. Hier ergibt sich also ein Widerspruch. Aus den Formeln folgt damit außerdem: $RCB \leq 1 - Precision$. Für alle vier Kennzahlen liegt der Wertebereich zwischen 0 und 1. *Precision*, *Recall* und F_1 sollten möglichst hoch sein und *Relative Cross Brackets* möglichst niedrig.

4 Implementierung

Dieses Kapitel beschreibt, wie das Konzept aus Kapitel 3 umgesetzt wurde. Es wird sowohl auf die neu implementierten Operatoren, als auch auf die verwendeten, bereits existenten RapidMiner-Operatoren eingegangen.

4.1 RapidMiner Studio

Die Plattform für das Parser-Framework wurde auf RapidMiner Studio [Rms] festgelegt. Hierbei handelt es sich um eine Data Science Plattform, in der Operatoren im Steckkasten-Prinzip miteinander verbunden werden. Zwischen diesen Operatoren werden Daten in unterschiedlicher Form übergeben. Die Operatoren selbst führen verschiedene Aufgaben aus. Das eigentliche RapidMiner Studio bietet hauptsächlich Operatoren für die Verarbeitung von Tabellen an. Allerdings sind über den RapidMiner Marketplace auch eine Vielzahl von Erweiterungen erhältlich. Im Rahmen dieser Arbeit wird die Text Processing Extension (Version 8.1.0) [Tex] verwendet. RapidMiner bietet zusätzlich die Möglichkeit eigene Operatoren zu programmieren, was im Rahmen dieser Arbeit auch genutzt wurde. Das RapidMiner Studio, die Erweiterungen und die eigenen Operatoren wurden in Java geschrieben.

4.2 Verwendete Operatoren

Hier werden die bereits bestehenden Operatoren beschrieben, welche im Prozess verwendet wurden.

Zum Einlesen des Parser-Modells wurde der Operator **Read File** aus dem Ordner *Utility\Files* verwendet. Dieser hat als Parameter den Pfad für die zu öffnende Datei. Er bietet einen Output-Port, über den die geöffnete Datei später als `com.rapidminer.operator.nio.file.SimpleFileObject` verarbeitet werden kann. Das Dateiformat der Modelle ist *.bin* für den OpenNLP-Parser, *.gr* für den Berkeley-Parser und *.ser.gz* für den Stanford-Parser. Wichtig ist hier aber nicht die geöffnete Datei, da alle Parser diese selbst nochmal öffnen. Die Parser bekommen als Übergabe den Pfad und dieser kann aus dem `SimpleFileObject` entnommen werden.

Aus der Text-Processing-Extension wird der Operator **Read Document** verwendet. Er wird an zwei Stellen verwendet. Zum einen wird über ihn der Input-Text für den Parser eingelesen. Zum anderen wird der *Goldstandard* für die Evaluierung damit in den Prozess gebracht. In beiden Fällen wird nur der Parameter *file* auf den entsprechenden Dateipfad gesetzt und alle anderen Parameter auf ihrem Default-Wert belassen. Auch sein Input-Port wird nicht verwendet. Über den Output-Port wird die geöffnete Datei als `com.rapidminer.operator.text.Document` an die nachfolgenden Operatoren übergeben.

Der letzte verwendete Operator ist **Execute Script** aus dem Ordner *Utility\Scripting*. Dessen Aufgabe ist das Ausführen von Java Code. Verwendung findet er lediglich in Kombination mit dem Stanford-Parser. Die verwendeten Packages von Stanford werden hier einmal initialisiert, da es sonst einen Fehler seitens RapidMiner gibt. Hierfür wird ein Objekt der Klasse `edu.stanford.nlp.trees.TreebankLanguagePack` erstellt und wieder auf `null` gesetzt. Weder die Input- noch die Output Ports des Operators werden genutzt, wodurch er vor den anderen Operatoren ausgeführt wird, und die Stanford-Packages korrekt initialisiert werden. Um diese Strategie der Fehlerumgehung nutzen zu können, sind zwei zusätzliche Schritte notwendig. Zum einen muss die Datei *stanford-parser.jar*, welche die entsprechenden Packages beinhaltet, in den lib Ordner von RapidMiner Studio gelegt werden. Diesen findet man typischerweise unter *Programme\RapidMiner\RapidMiner Studio\lib*. Das zweite und schwerwiegendere Problem ist der Bedarf einer Large License von RapidMiner. Mit der Lizenz in Kombination mit dem vorherigen Schritten wird der Fehler nicht geworfen. Anderweitig wird die Lizenz nicht benötigt. Für die Dauer dieser Arbeit wurde mir diese von RapidMiner kostenlos zur Verfügung gestellt.

4.3 Eigene Operatoren

Hier werden die Operatoren und alle dazugehörigen Klassen, die für dieses Parser-Framework entwickelt wurden, vorgestellt.

Zur Entwicklung einer eigenen Erweiterung stellt RapidMiner eine Anleitung zur Verfügung [Rmg]. Als Entwicklungsumgebung wurde Eclipse (Version: Oxygen.3a Release (4.7.3a)) in Kombination mit dem Gradle Plugin gewählt.

Alle entwickelten Operatoren erben von der Klasse `com.rapidminer.operator.Operator` und die Methode `doWork()` wird aufgerufen, wenn der Operator im Workflow durchlaufen wird.

Die Übergabe von Daten zwischen Operatoren erfolgt über Klassen, die das Interface `com.rapidminer.operator.IOObject` implementieren, wie z.B. `ExampleSet` oder `Document`.

Im Rahmen dieser Arbeit wurden folgende Operatoren erstellt: **Berkeley Parser**, **OpenNLP Parser**, **Stanford Parser**, **Compare Results** und **Show Results**

Alle Parser-Operatoren haben die gleichen Ports zur Ein- und Ausgabe.

ioobjectInputGrammar Über diesen Port wird das Modell des Parsers eingelesen. Dieses wird in `doWork()` auf ein `SimpleFileObject` gecastet, um den Dateipfad auslesen zu können. Er wird mit dem Read-File-Operator verbunden.

ioobjectInputText Hier liest der Operator den Input-Text für den Parser ein. Das `IOObject` wird auf ein `Document` gecastet, um den Inhalt als String zu bekommen. Dieser String wird dann nach Zeilen getrennt, da jeder Parser immer eine Zeile eingegeben bekommt. Er wird mit dem Read-Document-Operator verbunden.

nameOutputExt Es wurde ein PortExtender verwendet. Das heißt, falls man diesen Port verbindet wird der selbe Port nochmals erstellt. Somit kann man die Parser Ausgabe öfters verwerten. Dies kann man z.B. nutzen, um in der Evaluation unterschiedliche Paramter zu setzen. Extender müssen im Konstruktor mit der Methode `start()` aktiviert werden. Über die von diesem Extender erstellten Ports wird der Name des Parsers in einem `Document` nach außen gegeben. Hierfür gibt es einen Konstruktor der Klasse Document dessen Übergabeparameter ein String ist.

resultOutputExt Dieser Extender gibt den geparsten Text in einem `Document` aus. Hier sind alle einzeln verarbeiteten Zeilen wieder zu einem String zusammen gefügt worden.

4.3.1 Berkeley-Parser

Der Code für diesen Parser ist aus der Klasse `edu.berkeley.nlp.PCFGLA.BerkeleyParser` entnommen. Diese Klasse wird beim Kommandozeilenaufruf des Parsers angesprochen. Die `main`-Methode sowie `outputTrees(...)` wurden leicht abgewandelt übernommen.

Die Funktionalität der `main` wurde in die `doWork()` geschrieben. Über den eigentlichen Kommandozeilenaufruf können Optionen übergeben werden, die in ein Objekt

der Klasse `edu.berkeley.nlp.PCFGLA.BerkeleyParser.Options` geparst werden. Hier wird stattdessen ein Objekt dieser Klasse erstellt und die Optionen manuell gesetzt. Der Wert des Attributs `grFileName` wird auf den Dateipfad des eingelesenen Modells gesetzt. Bei allen anderen werden die Default-Werte beibehalten.
Die Ein- und Ausgabe in der Originalklasse war ein `BufferedReader` und ein `PrintWriter`. Das wurde abgewandelt, da die Eingabe jetzt zeilenweise als String zur Verfügung steht und die Ausgabe auch in einen String geschrieben werden soll. Der Parser bekommt seine Eingabe jetzt über eine `for`-Schleife, in der über das Array der Eingabesätze iteriert wird. Zuvor wurden per `while`-Schleife und die BufferedReader Methode `readLine()` der Parser-Input erhalten.

Die Methode `outputTrees(...)` wurde als Methode des Operators aufgenommen. Hierzu wurde der Rückgabetyp von `void` auf `String` geändert und der Übergabeparameter zur Datenausgabe von `PrintWriter` auf `String` abgeändert. Das führt dazu, dass statt `outputData.write("...")` nun `outputText += "..."` verwendet wird. Außerdem wird die Möglichkeit, die Ausgabe als .png zu erhalten, entfernt. Zuletzt wird der Methode ein `return outputText` angehängt. In der `doWork()` wird dieser String als `Document` verpackt und über den PortExtender an die Output-Ports geliefert.

4.3.2 OpenNLP-Parser

OpenNLP bietet eine Anleitung zum Integrieren des Parsers in eine Anwendung mit Hilfe seiner API. Zu finden unter [Ope]. Der Code aus dieser Anleitung wurde in die `doWork()` des Operator geschrieben. Dieser Code wurde von einem zusätzlichen `try-catch`-Block umgeben, um eine `FileNotFoundException` zu fangen. Diese Exception wird beim Öffnen der Datei des Parser-Modells geworfen. Per `for`-Schleife werden die Sätze der Eingabe in den Parser geschickt.

Über die String Methode `outputText.replaceAll("\\(TOP ", "( ");` wird das Startsymbol des Parsers entfernt. Um zu verhindern, dass das Auftreten des Wortes "TOP" im Eingabesatz gelöscht wird, wurde die Klammer mit in den regulären Ausdruck aufgenommen. Worte des Satzes haben in der Formatierung immer eine schließende Klammer nach sich, aber nie eine öffnende vorher.

4.3.3 Stanford-Parser

Beim Herunterladen des Parsers wird eine Demo-Klasse mitgeliefert, in welcher der Code zum Starten des Parsers aufgeführt wird. Hieraus wurden die notwendigen Befehle in die Methode `doWork()` des Operators übernommen. Zu beachten ist hier

noch, dass der Parser den Satz nicht als `String` entgegen nimmt, sondern die Token des Satzes in einem `String`-Array.

Das Startsymbol dieses Parser lautet “ROOT” und wird analog zum OpenNLP-Parser aus 4.3.2 entfernt.

4.3.4 Compare Results

In diesem Operator findet die Evaluierung statt. Es wird die Ausgabe von genau einem Parser mit dem *Goldstandard* verglichen und die Kennzahlen errechnet.

Hierzu gibt es drei Input-Ports, deren `IOObject` jeweils zu einem `Document` gecastet wird. Der erste Port erhält den Namen des Parsers, der zweite die Ausgabe der Parser und der dritte den *Goldstandard*. Das Ergebnis wird in einem einzeiligen `com.rapidminer.example.ExampleSet` ausgegeben. Dadurch kann man später mehrere Evaluierungen in einer Tabelle untereinander anzeigen lassen.

Der Operator hat zwei `Boolean`-Parameter. `PARAMETER_REMOVE_SUFFIX` gibt an, ob den syntaktischen Tags die Suffixe entfernt werden sollen. Das heißt, dass NP-SBJ als NP gewertet wird. `PARAMETER_COUNT_ONLY_SYNTACTIC_TAGS` entscheidet ob alle Tags oder nur die syntaktischen in die Bewertung zählen. Die Parameter wurden entsprechend dem Extension Manual [Rmg] erstellt. Default-Wert bezüglich der Suffixe ist `true` und bezüglich der syntaktischen Tags `false`.

Für die Evaluierung benötigt dieser Operator zwei zusätzliche Hilfsstrukturen. Zum einen das Enum **PennTag**. Hier sind alle Tags der Penn Treebank aufgelistet. Jedes Literal hat einen String als Attribut, der die Schreibweise angibt. Somit kann über die statische Methode `stringToPennTag(String s, boolean removeSuffix)` ein String auf ein Enum-Literal abgebildet werden. Hierzu werden alle Tags durchlaufen und deren Attribut mit dem Übergabestring verglichen. Das entsprechende Literal wird zurückgegeben. Für den Fall, dass dem String kein Tag entspricht, wird das zusätzlich eingeführte Literal `Empty`, mit leerem String als Attribut, zurückgegeben. Das passiert beispielsweise dann, wenn im Parser oder *Goldstandard* zusätzliche Tags verwendet werden. Eine Konstituente, die als Typ `Empty` hat, wird immer als falsch gewertet. `removeSuffix` sorgt dafür, dass die Tags ab dem ersten Bindestrich abgeschnitten werden und somit nur das ursprüngliche betrachtet wird.
Zum anderen wird die Klasse **ParseTreeNode** verwendet. Ein Objekt dieser Klasse entspricht einer Konstituente aus Kapitel 3.6. Realisiert wurde dieses Konzept über ein `PennTag` und zwei `int` Membervariablen für den Typ und die Grenzen. Über die Methode `equals(Object obj)` wird getestet, ob sowohl Tag, als auch Grenzen des übergebenen `ParseTreeNodes` mit dem aufrufendem übereinstimmen.

Der Operator selbst bedient sich wiederum zweier Hilfsmethoden. Mit

```
private static String formatSentence(String s)
```

wird jede Zeile so formatiert, dass vor und hinter jeder Klammer ein Leerzeichen ist. Außerdem gibt es keine aufeinanderfolgenden Leerzeichen. Dadurch kann der String nach Leerzeichen geteilt werden. Durch

```
private static List<ParseTreeNode> parseSentence(String s, boolean removeSuffix,
boolean countOnlySyntacticTags)
```

wird eine formatierte Zeile `s` in eine Liste von Konstituenten umgewandelt. Hierfür wird ein `Stack<ParseTreeNode>` genutzt. Der Satz wird nach Leerzeichen in Klammern, Nichtterminale und Terminale aufgeteilt. Diese werden mit einer `for`-Schleife durchlaufen. Damit ergeben sich vier Fälle.

- Fall 1: Öffnende Klammer. Es wird ein neuer `ParseTreeNode` erstellt und dessen Startpunkt auf die aktuelle Wortposition im Satz gesetzt. Diese Konstituente wird auf den Stack gelegt.

- Fall 2: Nichtterminal. Die oberste Konstituente des Stacks wird geholt, ihr Typ auf das gelesene Nichtterminal gesetzt und sie selbst wieder auf den Stack zurückgelegt.

- Fall 3: Terminal. Der Wortzähler wird inkrementiert. Das eigentliche Terminal wird nirgends abgespeichert, da diese in der Parser-Ausgabe und im *Goldstandard* identisch sind.

- Fall 4: Schließende Klammer. Der oberste `ParseTreeNode` wird vom Stack geholt und sein Endpunkt auf den aktuellen Stand des Wortzählers gesetzt. Gilt `countOnlySyntacticTags == true`, so werden nur diejenigen `ParseTreeNodes` an die Ausgabe gehängt, deren `PennTag` zu den syntaktischen Tags gehört. Falls nicht, so wird jede Konstituente der Ausgabeliste hinzugefügt.

Diese Methode gibt alle zu bewertenden Konstituenten des Satzes zurück.

In der `doWork()` des Operators werden sowohl Parser-Ausgabe, als auch *Goldstandard* nach Zeilen aufgeteilt. Für jede Zeile der beiden Versionen werden die beschriebenen Hilfsmethoden aufgerufen. Die Parser-Konstituenten werden durchlaufen und im *Goldstandard* nach passendem Gegenstück gesucht. Hierdurch bekommt man die Anzahl an korrekten Konstituenten. Für den *RCB*-Wert werden nochmals die `ParseTreeNodes` des Parsers durchlaufen und alle gezählt, die mindestens eine

Konstituente des *Goldstandards* kreuzen. Diese beiden Zahlen, sowie die Anzahl an Parser- und *Goldstandard*-Konstituenten werden für jede Zeile aufsummiert. Dadurch können, nach dem Vergleich aller Zeilen, entsprechend der Formeln (3.1) bis (3.4), die Kennzahlen für den Parser errechnet werden.
Zuletzt wird ein `ExampleSet` mit den Attributen `Name`, `Precision`, `Recall`, `F1`, `Crossing Brackets`, sowie der totalen Anzahlen der Konstitueten, aus denen die Kennzahlen berechnet wurden, erstellt. Diese eine Zeile mit den Werten des Parser, wird in dieses Set eingefügt, welches dann an den OutputPort übergeben wird.

4.3.5 Show Results

Dieser Operator dient lediglich dem Zusammenführen von mehreren Parser-Evaluationen.

Es wird für die Input-Ports wieder ein PortExtender verwendet, um beliebig viele Compare-Results-Operatoren anschließen zu können. Der Output-Port liefert eine Tabelle, in der pro Zeile die Ergebnisse eines Parsers stehen.

In der `doWork()` werden alle Input-Ports durchlaufen und die Werte in die Ergebnistabelle übertragen. Diese wird zuletzt an den Output-Port geliefert.

4.4 Prozess zum Evaluieren der Parser

Mit den vorgestellten Operatoren lässt sich im RapidMiner Studio nun ein Prozess zum Evaluieren der Parser erstellen. Die Anordnung der Operatoren zur Evaluierung des Berkeley-Parsers wird in Abbildung **??** präsentiert. Dieser Aufbau ist für alle Parser gleich. Lediglich bei **Read Grammar** muss beachtet werden, dass das entsprechende Modell geladen wird. Um mehrere Parser zu vergleichen, muss das Konstrukt aus **Compare Results**, und allen links davon liegenden Operatoren, für die anderen Parser erstellt werden. Die Ausgänge der **Compare Results**-Operatoren werden mit **Show Results** verbunden.
Abbildung **??** zeigt einen Teil der Ausgabe des Prozesses beim Vergleich aller Parser. Nach dem Namen der Parser werden die vier Metriken aufgelistet. In der Abbildung wurden rechts vier Spalten entfernt, welche die Zahlen angeben, aus denen sich die Metriken berechnen. Diese Zahlen sollen unter anderem dabei helfen die Größe der geparsten Datei einzuordnen.

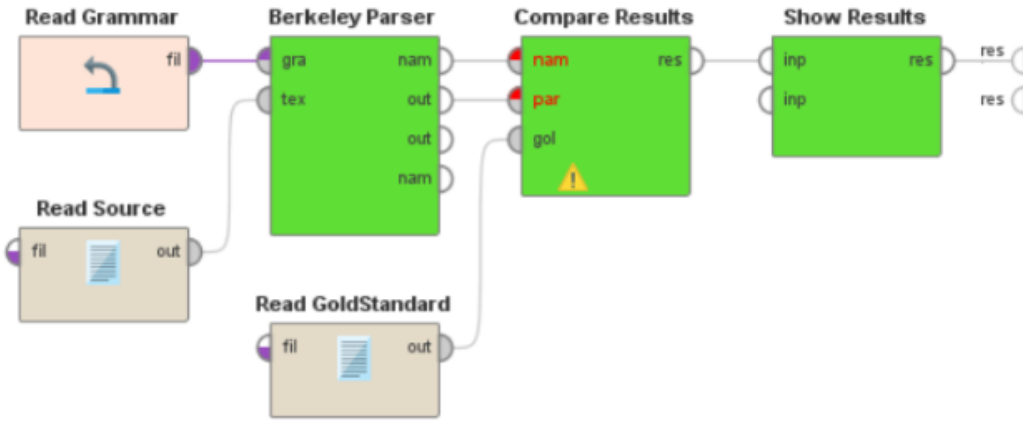

Abb. 4.1: RapidMiner-Prozess zur Evaluierung des Berkeley-Parsers

Name	Precision	Recall	F1	Crossing Brackets
Berkeley Parser	0.919	0.920	0.919	0.014
Stanford Parser	0.882	0.888	0.885	0.037
OpenNLP Parser	0.894	0.896	0.895	0.014

Abb. 4.2: Teil der Ausgabe des RapidMiner-Prozesses beim Vergleich aller Parser

5 Evaluierung der Parser

In diesem Kapitel wird die Funktionsweise der Parser vorgestellt. Danach wird die Leistung der Parser anhand der Ted-Talks-Treebank getestet und eine weitere Parser-Evaluation betrachtet.

5.1 Stanford-Parser

Der Stanford-Parser bringt ein Modell namens *englishPCFG* mit. Es handelt sich dabei um eine unlexikalisierte PCFG für die englische Sprache.

5.2 Berkeley-Parser

5.3 OpenNLP-Parser

5.4 Leistung der Parser

Dieser Korpus enthält zehn Texte unterschiedlicher Länge, die alle von jedem Parser bearbeitet werden. In Tabelle 5.1 sind die Ergebnisse aufgeführt. Für die Messung wurden alle Tags gewertet, nicht nur die syntaktischen. In einer Zeile stehen die Werte für die entsprechende Datei. Für den totalen Wert wurde die Anzahl der unterschiedlichen Konstituenten über alle zehn Dateien aufsummiert und anhand dieser Zahlen nach entsprechender Formel berechnet. Somit wird die unterschiedliche Größe der Texte berücksichtigt.
Der Berkeley-Parser nimmt mit Precision- und Recall-Resultaten von über 93% den ersten Platz ein. Der OpenNLP-Parser liegt, mit weniger als 1% Vorsprung in beiden Werten, vor dem Stanford-Parser. Beide weisen einen gleich hohen Anteil an kreuzenden Konstituenten auf.
Für die Interpretation dieser Ergebnisse muss beachtet werden, dass die Modelle nicht mit den selben Texten trainiert wurden. Es handelt sich um die vortrainierten Modelle der Parser-Anbieter. Um aussagekräftigere Ergebnisse zu erhalten, müssen eigene Modelle erstellt werden.

	Berkeley Parser				Stanford Parser				OpenNLP Parser			
Dateiname	Precision	Recall	F_1	RCB	Precision	Recall	F_1	RCB	Precision	Recall	F_1	RCB
SheaHembrey_2011	93,3%	93,2%	93,2%	2,7%	89,4%	89,2%	89,3%	4,4%	89,9%	89,3%	89,6%	3,6%
RobertLang_2008	93,8%	93,4%	93,6%	2,4%	89,4%	89,1%	89,3%	3,5%	89,2%	89,1%	89,2%	3,3%
JessaGamble_2010G	94,3%	94,4%	94,4%	2,2%	88,3%	89,6%	88,9%	3,8%	90,3%	90,3%	90,3%	2,9%
MihalyCsikszentmihaly_2004	93,8%	93,0%	93,4%	3,8%	87,8%	85,2%	86,4%	6,0%	89,6%	89,1%	89,3%	5,5%
YvesBahar_2009	91,9%	92,0%	91,9%	1,4%	88,2%	88,8%	88,5%	3,7%	89,4%	89,6%	89,5%	1,4%
KatherineFulton_2007	92,8%	92,3%	92,5%	3,5%	85,7%	83,4%	84,5%	4,8%	87,2%	87,3%	87,2%	6,0%
HannaRosin_2010W	93,1%	92,8%	92,9%	4,1%	90,0%	89,3%	89,7%	4,2%	89,3%	88,5%	88,9%	4,8%
HansRosling_2010S1	92,4%	92,6%	92,5%	2,9%	88,9%	89,6%	89,2%	3,6%	87,4%	87,2%	87,3%	4,4%
StefanaBroadbent_2009G	92,8%	92,1%	92,5%	3,8%	88,5%	88,0%	88,2%	5,6%	87,9%	87,8%	87,8%	5,5%
AnthonyAtala_2009P	94,5%	94,3%	94,4%	2,0%	89,0%	87,9%	88,4%	3,9%	90,3%	90,4%	90,3%	3,4%
Total	93,4%	93,1%	93,3%	3,0%	88,7%	87,9%	88,3%	4,4%	89,1%	88,8%	89,0%	4,4%

Tab. 5.1: Evaluation der Parser für die Ted-Talks-Treebank

5.5 Weitere Evaluationen

6 Verwandte Arbeit

Für RapidMiner ist kein weiteres Parser-Evaluations-Rahmenwerk bekannt. Dagegen wird in [Kan+11] und [Kan+09] *U-Compare* vorgestellt. Es handelt sich dabei um ein umfassendes Rahmenwerk zur Evaluation verschiedener NLP-Techniken. Es ist unter anderem möglich Tokenizer, POS-Tagger und syntaktische Parser zu vergleichen. Stand 2009 waren sowohl der Stanford-, als auch der OpenNLP-Parser im Framework integriert. Das Programm wird angeboten vom *National Centre for Text Mining (NaCTeM)* [Nac], mit Sitz in der Universität von Manchester.
U-Compare baut auf *UIMA* [Uim] auf. Dies ist ein einem Rahmenwerk zum Erstellen von Anwendungen für die Verarbeitung unstrukturierter Informationen, was zum Beispiel natürlich-sprachlicher Text ist. Die Arbeitsweise von *U-Compare* ist ähnlich zur hier vorgestellten RapidMiner Erweiterung. Es werden Daten eingelesen und durch verschiedene Komponenten verarbeitet. Diese Komponenten sind vergleichbar mit RapidMiner Operatoren. Per Drag-and-Drop können mehrere Komponenten hintereinander zu einem Ablaufplan angeordnet werden. Eine Komponenten kann beispielsweise ein Tokenizer, ein POS-Tagger, ein Parser oder auch eine Kombination aus allen drei sein. Es gibt die Möglichkeit diese Ablaufpläne zu evaluieren. So können mehrere Tokenizer, POS-Tagger und Parser angegeben werden und *U-Compare* errechnet für alle möglichen Kombinationen die *Precision*, *Recall* und F_1 Werte. [Kan+09] [Uco]
Somit ist hiermit ein mächtiges Werkzeug zur NLP-Evaluierung auf einer anderen Plattform bereits vorhanden, welches die Funktionalität dieser Arbeit größtenteils beinhaltet. Dennoch ist die Entwicklung eines NLP-Evaluations-Rahmenwerks für RapidMiner, in Hinblick auf die *RapidMiner Community* mit einer Größe von über 500.000 Mitgliedern (Stand 07.06.2018), ein interessantes Projekt.

7 Schluss

Zuletzt wird ein Fazit gezogen und ein Ausblick gegeben.

Literatur

[Bie+95] Ann Bies, Mark Ferguson, Karen Katz et al. „Bracketing guidelines for Treebank II style Penn Treebank project“. In: *University of Pennsylvania* 97 (1995), S. 100 (zitiert auf Seite 6).

[Bla+91] Ezra Black, Steven Abney, Dan Flickenger et al. „A procedure for quantitatively comparing the syntactic coverage of English grammars“. In: *Speech and Natural Language: Proceedings of a Workshop Held at Pacific Grove, California, February 19-22, 1991*. 1991 (zitiert auf Seite 23).

[Col97] Michael Collins. „Three generative, lexicalised models for statistical parsing“. In: *arXiv preprint cmp-lg/9706022* (1997) (zitiert auf den Seiten 17, 18).

[GT11] Michael T Goodrich und Roberto Tamassia. *Data structures and algorithms in Java*. John Wiley & Sons, 2011 (zitiert auf Seite 10).

[Hof15] Dirk W Hoffmann. *Theoretische Informatik*. Carl Hanser Verlag GmbH Co KG, 2015 (zitiert auf Seite 7).

[Kan+09] Yoshinobu Kano, William A Baumgartner Jr, Luke McCrohon et al. „U-Compare: share and compare text mining tools with UIMA“. In: *Bioinformatics* 25.15 (2009), S. 1997–1998 (zitiert auf Seite 37).

[Kan+11] Yoshinobu Kano, Makoto Miwa, K Bretonnel Cohen et al. „U-Compare: A modular NLP workflow construction and evaluation system“. In: *IBM Journal of Research and Development* 55.3 (2011), S. 11–1 (zitiert auf Seite 37).

[KM03] Dan Klein und Christopher D. Manning. „Accurate Unlexicalized Parsing“. In: *Proceedings of the 41st Annual Meeting on Association for Computational Linguistics - Volume 1*. ACL '03. Sapporo, Japan: Association for Computational Linguistics, 2003, S. 423–430 (zitiert auf Seite 22).

[KT07] Taavet Kikas und Margus Treumuth. *Automatic Parser Evaluation*. 2007 (zitiert auf Seite 23).

[Mar+93] Mitchell Marcus, Beatrice Santorini und Mary Ann Marcinkiewicz. „Building a large annotated corpus of English: The Penn Treebank“. In: (1993) (zitiert auf den Seiten 4, 8).

[MJ09] James H Martin und Daniel Jurafsky. *Speech and language processing: An introduction to natural language processing, computational linguistics, and speech recognition*. Pearson/Prentice Hall Upper Saddle River, 2009 (zitiert auf den Seiten 3, 5, 7, 8, 10–12, 15, 16, 18).

[Neu+14] Graham Neubig, Katsuhito Sudoh, Yusuke Oda et al. „The naist-ntt ted talk treebank“. In: *International Workshop on Spoken Language Translation*. 2014 (zitiert auf Seite 21).

[Pet+06] Slav Petrov, Leon Barrett, Romain Thibaux und Dan Klein. „Learning accurate, compact, and interpretable tree annotation“. In: *Proceedings of the 21st International Conference on Computational Linguistics and the 44th annual meeting of the Association for Computational Linguistics*. Association for Computational Linguistics. 2006, S. 433–440 (zitiert auf Seite 22).

Websites

[Con] Linguistic Data Consortium. *About LDC*. URL: https://www.ldc.upenn.edu/about (besucht am 4. Juni 2019) (zitiert auf Seite 7).

[Nac] *NaCTeM*. URL: http://nactem.ac.uk/ (besucht am 7. Juni 2019) (zitiert auf Seite 37).

[Nlt] *Natural Language Toolkit*. URL: http://www.nltk.org/ (besucht am 4. Juni 2019) (zitiert auf Seite 21).

[Ope] *Apache OpenNLP Developer Documentation*. URL: https://opennlp.apache.org/docs/1.9.1/manual/opennlp.html (besucht am 4. Juni 2019) (zitiert auf den Seiten 20, 22, 28).

[Rel] *Penn Treebank II tag set*. URL: https://www.clips.uantwerpen.be/pages/mbsp-tags (besucht am 6. Juni 2019) (zitiert auf Seite 5).

[Rmg] *Creating your Own Extension*. URL: https://docs.rapidminer.com/latest/developers/creating-your-own-extension/ (besucht am 5. Juni 2019) (zitiert auf den Seiten 26, 29).

[Rms] *RapidMiner Studio*. URL: https://rapidminer.com/products/studio/ (besucht am 5. Juni 2019) (zitiert auf Seite 25).

[Sta] *Stanford Tokenizer*. URL: https://nlp.stanford.edu/software/tokenizer.html (besucht am 4. Juni 2019) (zitiert auf Seite 20).

[Tex] *Text Processing*. URL: https://marketplace.rapidminer.com/UpdateServer/faces/product_details.xhtml?productId=rmx_text (besucht am 5. Juni 2019) (zitiert auf Seite 25).

[Uco] *Evaluation and comparison mechanism*. URL: http://nactem.ac.uk/ucompare/documentation/evaluation-and-comparison-mechanism/ (besucht am 7. Juni 2019) (zitiert auf Seite 37).

[Uim] *Apache UIMA*. URL: http://uima.apache.org/ (besucht am 7. Juni 2019) (zitiert auf Seite 37).

Abbildungsverzeichnis

Tabellenverzeichnis

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides statt, dass ich die vorliegende Arbeit selbständig verfasst und dabei keine anderen als die angegebenen Hilfsmittel benutzt habe. Sämtliche Stellen der Arbeit, die im Wortlaut oder dem Sinn nach Publikationen oder Vorträgen anderer Autoren entnommen sind, habe ich als solche kenntlich gemacht. Die Arbeit wurde bisher weder gesamt noch in Teilen einer anderen Prüfungsbehörde vorgelegt und auch noch nicht veröffentlicht.

Bayreuth, Juni 14, 2019

Klaus Freiberger