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
Bachelor Thesis

Philipp Scholz

23.April 2019
Version: Final

Universität Bayreuth

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

NLP-Plattform: Integration und Evaluation von POS-Tagging-Algorithmen
NLP-Platform: Integration and Evaluation of POS-Tagging-Algorithms

Bachelorarbeit
Bachelor Thesis

Philipp Scholz

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 und Stefan Jablonski

23.April 2019

Philipp Scholz
Bachelorarbeit
Bachelor Thesis
NLP-Plattform: Integration und Evaluation von POS-Tagging-Algorithmen
NLP-Platform: Integration and Evaluation of POS-Tagging-Algorithms, 23.April 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 (Deutsch)

Im Themenbereich *Natural Language Processing* (kurz NLP) versucht die Informatik, natürliche Sprachen für Algorithmen zugänglich und interpretierbar zu machen. Ein wichtiger Teil von NLP ist die Identifizierung der syntaktischen Bedeutung von Wörtern (*Parts of Speech*, kurz POS) in gesprochener Sprache, und deren Zuweisung in Form von Tags (POS-Tagging). Für diese Aufgabe existiert eine Vielzahl unterschiedlich leistungsfähiger und robuster Algorithmen.
Im Rahmen dieser Arbeit wurde eine Sammlung solcher POS-Tagging-Algorithmen zusammen mit einem Evaluationssystem als Operatoren im Programm RapidMiner (zur Verfügung gestellt von RapidMiner GmbH) implementiert.

Abstract (English)

The field of Computer Science called *Natural Language Processing* (NLP) attempts to make natural language accessible for machines and algorithms. One of the major parts of NLP is the identification of the syntactic roles of words and symbols (*Parts of Speech, POS* in short) in spoken language, and tagging them as such POS (POS-Tagging). There is a multitude of differently capable and robust algorithms for this task.
In this project, such POS-Tagging algorithms and a system to evaluate them is implemented in form of operators in an extension to the program RapidMiner (by RapidMiner GmbH).

Inhaltsverzeichnis

1 Einleitung

Keine bekannte Lebensform hat eine mit der menschlichen Sprache vergleichbar komplexe Form von Informationsaustausch entwickelt [C R18]. Von selbst ergibt sich die Frage, ob und wie Sprache maschinell verarbeitet werden kann, um sie unter anderem zu interpretieren oder zusammenzufassen. Antworten auf diese Problemstellung liefert das Teilgebiet der Informatik *Natural Language Processing* (Dt. Verarbeitung natürlicher Sprachen, kurz NLP). NLP gliedert sich in viele Teilbereiche: Strukturelle Analyse, Semantik, Phonetik und einige weitere. In dieser Arbeit konzentrieren wir uns auf einen wichtigen Bestandteil der strukturellen Analyse, dem korrekten Identifizieren von syntaktischen Rollen von Wörtern und Symbolen (*Parts of Speech*, kurz POS), bezeichnet als *POS-Tagging* [Smi11].
Ein Algorithmus, der POS-Tagging betreibt (POS-Tagger), nimmt die Sprache in Textform an und gibt ihn üblicherweise mit Tags versehen wieder aus, wie beispielsweise der Text

I like the blue house.

zu folgendem verarbeitet wird:

*I****PRONOUN** *like****VERB** *the****DET** *blue****ADJ** *house****NOUN** .**.**

Wie die tatsächliche Ausgabe eines Taggers exakt formatiert wird und welche Tags auftreten, wird später angesprochen. Wichtiger ist hingegen, dass POS-Tagger diese Tags nicht garantiert korrekt wählen (siehe Abschnitt 2.1). Es ist also von Interesse, deren Performance zu bewerten.

1.1 Problemstellung dieser Arbeit

Die Datenverarbeitungs-Plattform *RapidMiner Studio* (ab hier nur RapidMiner) [Rapb] bietet im Rahmen der Erweiterung *Text Processing* [Gmb18] Funktionen (*Operatoren*), um Texte einzulesen und zu verarbeiten. NLP-Funktionen sind in RapidMiners Textverarbeitungs-Erweiterungen jedoch weitgehend noch nicht implementiert. Ziel dieser Arbeit ist es, in Form einer auf *Text Processing* aufbauenden

Erweiterung sowohl POS-Tagger zu implementieren, als auch ein Evaluationsrahmenwerk, mit dem deren Performance gemessen werden kann.

Die implementierte Erweiterung liefert drei Tagging-Operatoren, einen Evaluationsoperatoren, ein spezialisiertes Übergabeformat für Tagging-Ergebnisse und eine Standardisierung für verwendete POS-Tags. Gleichzeitig sind die Operatoren in ihrem Ein- und Ausgangsformat weitgehend kompatibel mit der Erweiterung *Text Processing*.

1.2 Aufbau der Arbeit

Kapitel 2 betrachtet die Grundlagen von POS-Tagging und einige Implementierungen von Part-of-Speech-Taggern und deren Evaluation in anderen Projekten.

Kapitel 3 erläutert die Methodik und behandelten Probleme in diesem Projekt.

Kapitel 4 beschreibt detailliert die Implementierung und deren Zielsetzung der RapidMiner-Erweiterung.

Kapitel 5 führt Evaluationen der implementierten POS-Tagger anhand eines gewählten Test-Musterergebnisses auf.

2 Hintergrund und Verwandte Arbeiten

In diesem Kapitel sollen einige Grundlagen über NLP und Part-of-Speech-Tagging erklärt werden. Danach soll auf einige Arbeiten verwiesen werden, in denen Evaluationen und Konzeptionen von POS-Taggern stattfinden. Es wird besonders untersucht, *wie* und nach welchen Metriken POS-Tagging-Ergebnisse evaluiert werden.

2.1 Part-of-Speech-Tagging

Betrachtet man das Wort „like“ aus dem einleitenden Beispiel, dann fällt auf, dass es alternativ zum Verb „mögen“ auch als Präposition „wie“ interpretiert werden könnte, auch wenn intuitiv schnell klar wird, dass letztere Variante falsch ist. Diese Uneindeutigkeit (*Ambiguität*) und damit die Aufgabe der *Disambiguation* ist das zentral zu lösende Problem für POS-Tagger [Smi11] [D B97]; Im Gegensatz zum Menschen kann ein Algorithmus Ambiguitäten nicht intuitiv auflösen. Aus diesem Grund arbeiten POS-Tagger nicht zwingend vollständig korrekt.
Während NLP viele andere Analyseaufgaben neben POS-Tagging zusammenfasst, sind diese bei moderneren und komplexeren NLP-Algorithmen nicht mehr strikt von POS-Tagging trennbar, wenn die Performance des Taggers maximiert werden soll [Smi11]. Zusatzinformationen, die parallel erarbeitet werden können, wie z.B. die Satzstruktur (u.A. Identifizierung von Teilsätzen), können das Auflösen von Ambiguitäten erheblich erleichtern. Zur Vereinfachung betrachten wir aber in dieser Arbeit nur den für POS-Tagging relevanten Kontext, welcher typischerweise in folgende zwei Arbeitsschritte unterteilt wird [Smi11]:

Sequenzierung: Zuerst muss bestimmt werden, welche Teile des angegebenen Dokuments jeweils ein Tag erhalten. Hierzu wird jedes Wort und jedes zusammenhängende Satzzeichen als *Token* ausgegeben. Die Reihenfolge der Wörter bleibt als Reihenfolge der Token hierbei erhalten. Ferner können auch Sätze voneinander getrennt werden.

Tagging: Mit Hilfe von statistischen, linguistischen und rechnerischen Methoden wird jedem Token ein POS-Tag zugewiesen.

Es folgen noch ein paar weitere Erläuterungen und Definitionen von Begriffen:

2.1.1 Tagset

Um einheitliche Verarbeitung und Vergleichbarkeit zu ermöglichen, werden die Tags in *Tagsets* definiert, wie zum Beispiel dem des Penn-Treebank-Projekts [Sta03] [MP 93]. Ein Tagset ist nichts weiter als eine Liste von Tags mit spezifischer Bedeutung (z.B. *NN* für *Nomen*) [Hal99].

2.1.2 Korpus und Treebank

Als *Korpus* bezeichnet man eine simple Ansammlung von Text. Die *Penn Treebank* [MP 93] beispielsweise enthält eine Sammlung von Artikeln der Nachrichtenartikel des *Wall Street Journals*.
Korpora, die zu Token-Ketten sequenziert wurden und deren Token mit (POS-) Tags versehen wurden, werden als annotierte Korpora oder *Goldstandard* bezeichnet [Hal99] Enthalten diese Korpora auch noch mit Satzstruktur-Tags (*Parse Trees*) versehen, spricht man von einer *Treebank*. Goldstandards sind in der Regel zu nahezu 100% korrekt annotiert. Da Goldstandards manuell korrigiert werden, können allerdings selten Fehler auftreten. Diese Fehler sind jedoch zu selten, um als relevant angesehen zu werden.

2.1.3 Herangehensweisen ans POS-Tagging

In diesem Abschnitt soll kurz erläutert werden, mit welchen Methodiken Part-of-Speech-Tagging betrieben werden kann bzw. betrieben wird. Wichtig sind hierbei die Begriffe *Lexikon* und *Regel-Modell* (oder kurz *Modell*) [Smi11] [Van14]. Im Lexikon kann ein Tagger die auftretenden Worte nachschlagen und dadurch herausfinden, welche Tags überhaupt möglich sind. Das Modell beschreibt linguistische Regelsätze, die helfen können, Tags auf ihre Wahrscheinlichkeit zu prüfen (z.B. „Tagfolge X→Y ist unwahrscheinlich oder gar unmöglich“ oder „Verben folgen nur selten aufeinander“) und damit Ambiguitäten durch Betrachtung des Kontextes (also der umgebenden Wörter) besser aufzulösen.

Lexikon- und Regelbasiert

Einfache, regelbasierte Tagger werden relativ aufwändig von Hand und mit Hilfe von statistischen und linguistischen Methoden erstellt [Hal99]. Der Aufbau des Lexikons

muss manuell übernommen werden, und das Regelmodell wird ebenfalls händisch definiert. Ein solcher Tagger ist entweder hochspezialisiert oder sehr generisch, was seine Eingaben betrifft [Van14]. Fest kodierte Lexika und Modelle wurden jedoch von maschinellem Lernen abgelöst [Smi11].

Maschinelles Lernen

Maschinelles Lernen in NLP zeichnet sich dadurch aus, dass mit Hilfe von einem (möglichst großen) annotierten Korpus sowohl Lexikon als auch Korpus automatisch generiert werden [Hal99]. Der zugrundeliegende Algorithmus beobachtet hierbei im für ihn sichtbaren Rahmen, welche Worte in welchem Kontext welche Tags erhalten können. Basierend auf diesem Trainingswissen kann der Tagger dann auf Tag-losen Texten verwendet werden. Wichtig ist hierbei anzumerken, dass das Training maßgeblich die Performance des Taggers beeinflusst [Smi11] [Sor13]. Wird der Tagger auf eine bestimmte Sorte von Sprache (z.B. Social Media Posts) trainiert, entwickelt er einen *Bias* in Richtung dieser Sprache. Das heißt, der Tagger ist in seinem Modell und Lexikon stark auf diese Ausdrucksweise und ihren Wortschatz spezialisiert und wird voraussichtlich in einem anderen sprachlichen Umfeld schlechtere Ergebnisse liefern. Ferner liefert ein größerer Unterschied zwischen Trainings- und Testdaten umso weniger akkurate Ergebnisse.

Worttransformation

Um *Bias* und andere Spezialisierungseffekte wie ein zu kleines Lexikon abzuschwächen, wird oft Worttransformation genutzt [Sor13] [D B97]. Dabei kann ein betrachtetes Wort, dessen Syntaktische Rolle dem Tagger nicht bekannt ist, z.B. durch Hinzufügen und Löschen von Prä- und Suffixen (*Nearest-Neighbours*) oder Synonyme (*Analogien*) durch ein Wort ersetzt werden, das für den Tagger leichter zu evaluieren ist. Ein Tagger kann zum Beispiel ein Wort, das Adjektiv und Verb sein kann, durch ein eindeutiges Adjektiv ersetzen und dann mit dem Modell prüfen, ob dieses Tag im Kontext Sinn ergibt.

2.2 Verwandte Arbeiten

Da Part-of-Speech-Tagging aus NLP nicht wegzudenken ist, existiert eine große Menge an Literatur zum Thema. Von Basiswissen bis hin zu extrem detaillierten Papers über neueste Optimierungen kann der Leser geradezu alles finden. In diesem

Abschnitt soll eine kleine Sammlung von Arbeiten geboten werden, die diese Spanne abdecken.

2.2.1 Evaluation des Stanford Taggers

In einem Artikel über den Stanford-Tagger, der zu POS-Tagging fähig ist, wird auch mit Daten von Sektionen aus dem annotierten *Wall Street Journal* Korpus verglichen, um die Qualität der Taggingergebnisse zu prüfen. Hierzu werden unter Anderem die Metriken *Per-Tag-Accuracy* und *Per-Sentence-Accuracy* (siehe Kap. 3.4) genutzt [al03].

2.2.2 P. Paroubek: Evaluating Part-of-Speech Tagging and Parsing

In dieser Arbeit ([Par07]) erklärt Paroubek die Grundlagen von Parsing von Goldstandards sowie der Evaluation von Ergebnissen durch Vergleiche zwischen Tagging-Ergebnissen und dem Goldstandard. Er erläutert auch das Konzept von Tokenization, der Zerlegung des eingelesenen Textes in linguistische Teilstücke, und spricht von Problemen durch eine fehlende Standardisierung dieses Prozesses (siehe 3.1).

2.2.3 N. Smith: Linguistic Structure Prediction

Noah Smith verfolgt in *Linguistic Structure Prediction* [Smi11] das Ziel, eine Brücke zwischen NLP und Maschinenlernen zu schlagen. Er erläutert, inwiefern NLP-Probleme auch Probleme der Mustererkennung sind, und zeigt, wie man das Problemfeld für einen Algorithmus zugänglich formalisieren kann. Dabei startet er mit Definitionen der Teilprobleme selbst, erklärt gängige Lösungsmethoden und auch, wie man über deren Lösungsqualität urteilt. Das Buch richtet sich also sowohl an Linguisten, die Schwierigkeiten mit dem komplexen Thema Maschinenlernen haben, als auch an Mathematiker, die in den meisten Werken über NLP zu viel Abstraktion und zu wenig Detailarbeit sehen.

2.3 Zusammenfassung

Es existiert viel Material zum Thema NLP und POS-Tagging und viele Vorgehensweisen und Klassifizierungen sind sehr standardisiert. Nachdem nun die Grundbegrifflichkeiten von POS-Tagging etabliert sind, können wir nun zum Arbeitskonzept der zu implementierenden Plattform übergehen.

3 Konzept

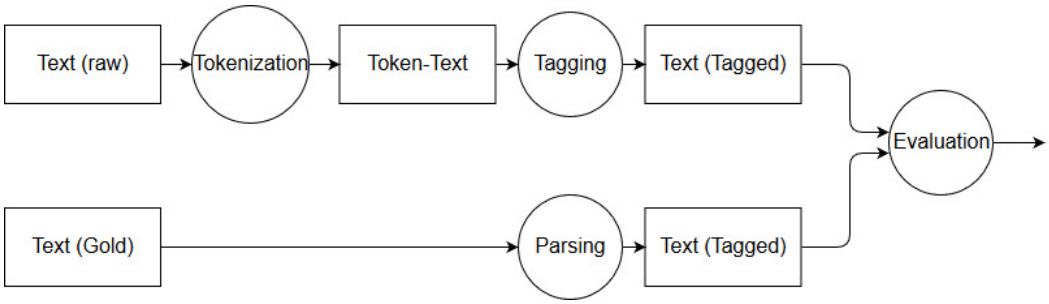

Abb. 3.1: Datenfluss für einen typischen Prozess von Rohdaten bis zur Evaluation

Aus Abbildung 3.1 können die Arbeitsschritte und Daten entnommen werden, die für Tagging und Evaluation modelliert werden müssen. Die folgenden Kapitel behandeln diese Schritte sowie insbesondere deren Eingaben und Ergebnisse.

3.1 Sequenzierung

Bei der Sequenzierung (*Tokenization*) wird zwischen Wort- und Satzsequenzierung unterschieden [Smi11]. Satzsequenzierung beschäftigt sich mit einer Zerlegung des Textes in Sätze, welche für spätere Evaluation nützlich sein kann. Für Part-of-Speech-Tagging ist allerdings vorwiegend die Wort-Sequenzierung wichtig, da jedem Element der Wort-Sequenz ein Token zugewiesen wird. Außerdem sollte eine Satz-Sequenzierung vor dem Tagging nicht unbedingt stattfinden, da Tagger ggf. Kontextinformationen aus anderen Sätzen nutzen könnten, was die Auflösung von Ambiguitäten weiter unterstützen könnte. Allerdings verlangen manche Tagger eine Satzsequenzierung [Par07].

Da Sequenzierunges-Algorithmen nicht standardisiert sind [Par07] und deren Ergebnisse von anderen - und insbesondere von Goldstandards - abweichen können, entstehen Unsicherheiten beim Vergleich dieser Sequenzen.

3.1.1 Tokenization-Unterschiede

Ein Token ist ein Abschnitt, zu dem ein Tag gehört, typischerweise einzelne Wörter oder Satzzeichen. Formal wird aus dem zusammenhängenden eingelesenen Wort beziehungsweise Text *w* eine Menge *M* von *n* Teilwörtern c_i

$$M = \{c_1, c_2, ..., c_n\}$$

generiert. POS-Tagger wie der von NLP4J [16] und der Stanford University [al03] können eine solche Zerlegung selbst übernehmen. Nehmen wir nun an, *M* wäre die Zerlegung von *w* in einem Goldstandard, und ein Tagger produziert ein Ergebnis mit der Zerlegung:

$$M_{tag} = \{d_1, d_2, ..., d_m\}$$

Hierbei sind d_i wieder Teilworte und *m* die Anzahl dieser. Spätestens wenn nun $n \neq m$ gilt, wird klar, dass *M* und M_{tag} nicht mehr einfach iterativ verglichen werden können, da ab der Stelle *p*, wo der Text unterschiedlich geteilt wurde, $c_i \neq d_i$ sein kann, wobei $(p \leq i \leq min\{n, m\})$. Es ist sogar nicht auszuschließen, dass solche Fehler bereits vorher passieren, obwohl weiterhin $n = m$ gilt.
Eines der Ziele bei der Implementierung ist also, den Vergleich von inhomogenen Token-Mengen robust gegen solche Fehler zu machen.

3.1.2 Externe Tokenization

Um alternativ Probleme wie im vorigen Abschnitt zu verhindern, kann alternativ die Sequenzierung extern übernommen werden. D.h. dass den Taggern der Text bereits als Token-Kette übergeben wird. Hierzu muss entweder ein Tokenizer entwickelt werden, der den Taggern Tokens übergibt, die mit ihrem verlangten Format konform sind, oder im Falle eines Goldstandards kann die Token-Zerlegung aus dem annotierten Korpus selbst entnommen werden.
Die meisten (alle implementierten) Tagger unterstützen das Einlesen von bereits zerlegten Texten.

3.2 Ergebnisformat

Abhängig vom Tagger können Ergebnisse sehr informationsreich sein. Der LingPipe-POS-Tagger liefert beispielsweise pro Token (Wort oder Symbol) nicht nur das aus seiner Sicht plausibelste Tag (*First-Best*), sondern auch *n-1* weitere, nachstehende Optionen (*N-Best*), wobei *n* gewählt werden kann [11]. Eine solche Menge an Metainformationen kann möglicherweise von bereitgestellten Datenstrukturen seitens

RapidMiner nur schwer kodiert werden. Eine speziell für Ergebnisse entworfene Datenstruktur bietet hingegen neue Freiheiten:

Satz-Sequenzierung der Tag- und Token-Kette anhand von bestimmten Satzzeichen, die bei der Wort-Sequenzierung immer als eigenes Token eingefügt werden. Dies macht robuster gegen Verschiebungen im Ergebnis, wie in Abschnitt 3.1.1 beschrieben: Wird Satzweise verglichen und entstehen in den Sätzen Verschiebungen durch ungleiche Token-Zerlegungen, dann verschwindet dieser Fehler, sobald ein neues Segment betreten wird, da die neu betrachteten Segmente wieder an gleicher Stelle beginnen.

Anreicherung mit Zusatzinformationen ist möglich, da im Gegensatz zu einer einfachen textuellen Darstellung auf ein Token nicht exakt ein Tag folgen muss. Hier kann beispielsweise pro Token statt nur dem First-Best-Tag eine beliebig lange Liste der *N-Best* Tags stehen.

Ein solches alternatives Format ist jedoch von Operatoren, die nicht speziell dafür vorbereitet sind (wie der Evaluationsoperator), nicht lesbar. Darum müssen Ergebnisse auch noch in einem Text-Format ausgegeben werden.

3.3 Parsing von Ergebnissen

Muss bei der Evaluation ein Text-Format angenommen werden, beispielsweise das Ergebnis in einem Goldstandard, muss entweder das angenommene Format exakt bekannt und definiert sein, oder das Dokument muss auf POS-Tags durchsucht werden. Letzteres bedeutet, dass dem Parser das Tagset des angenommenen Dokuments bekannt sein muss. Hierzu muss also jedes möglicherweise auftretende Tagset definiert werden.

3.4 Evaluation

Um die Performance eines Taggers zu bewerten, müssen Metriken eingeführt werden. Diese entstehen aus Vergleichen zwischen annotierten Korpora und den Tagging-Ergebnissen. Im Rahmen dieses Projekts wird nur der Vergleich von Ergebnissen mit identischen Tagsets betrachtet. Abbildungsfunktionen zwischen Tagsets oder zu einem übergeordneten, generalisierten Tagset sind prinzipiell (mit einem gewissen Informationsverlust) möglich, werden aber hier außen vor gelassen. Betrachtet werden die folgenden Metriken zur Evaluation von Tagging-Ergebnissen:

3.4.1 Per-Tag-Accuracy

Die Per-Tag-Accuracy (ab hier nur *Accuracy*) ist die wichtigste und einfachste Form der Bewertung eines Taggers. Sie ist definiert als [C R18] :

$$\frac{\#\ korrekter\ Tags}{\#\ aller\ Token}$$

3.4.2 Per-Sentence-Accuracy

Analog zur Accuracy pro Tag kann auch Accuracy pro Satz definiert werden:

$$\frac{\#\ vollständig\ korrekter\ Sätze}{\#\ aller\ Sätze}$$

Hierzu müssen jedoch über Wort-Sequenzen hinaus auch Satz-Sequenzen definiert werden. Das fällt jedoch relativ leicht, da man die Sätze einfach nach jedem Satzzeichen-Token abschließen kann.

3.4.3 Confusion Matrix und daraus resultierende Metriken

Konzentrieren wir uns auf bestimmte Tags (Labels), können wir genauere Aussagen treffen (Für allgemeine Definitionen siehe [C R18], für mehrklassige Klassifizierungsprobleme siehe [Gan14]). Zuerst bilden wir die *Confusion Matrix*. In Abbildung 3.2 befindet sich ein Beispiel dafür. An den Positionen (A,B) in der Matrix steht die Zahl der Token, die im Goldstandard mit Tag A und im Tagging-Ergebnis mit Tag B markiert wurden.

True Positives für Label X (kurz TP(X)) sind Vorhersagen, die dem Goldstandard entsprechen (in der Abbildung Gelb markiert).

False Positives (FP(X)) sind alle Vorhersagen X, die dem Goldstandard widersprechen.

True Negatives (TN(X)) sind alle Vorhersagen, die nicht X sind, und auch im Goldstandard nicht X lauten.

False Negatives (FN(X)) sind alle Vorhersagen, die nicht X sind, im Goldstandard aber X.

	GoldLabel_A	GoldLabel_B	GoldLabel_C	
Predicted_A	**30**	20	10	TotalPredicted_A=60
Predicted_B	50	**60**	10	TotalPredicted_B=120
Predicted_C	20	20	**80**	TotalPredicted_C=120
	TotalGoldLabel_A=100	TotalGoldLabel_B=100	TotalGoldLabel_C=100	

Abb. 3.2: Beispiel für eine Confusion Matrix für Labels A, B, C. Entnommen von [Gan14]

Daraus lassen sich Precision(Wie viele der Vorhersagen X waren korrekt?), Recall(Wie viele X im Goldstandard wurden korrekt erkannt?) und der F1-Score(auch F-Score, harmonisches Mittel aus Precision und Recall) berechnen:

$$Precision(X) = \frac{TP(X)}{TP(X) + FP(X)} = \frac{Korrekte\ X}{Vorhergesagte\ X}$$

$$Recall(X) = \frac{TP(X)}{TP(X) + FN(X)} = \frac{Korrekte\ X}{X\ im\ Goldstandard}$$

$$F1\text{-}Score = 2 * \frac{Precision(X) * Recall(X)}{Precision(X) + Recall(X)}$$

Für alle drei dieser Metriken steht der Wert 1 für vollständig richtig und 0 für vollständig falsch.

3.5 Zusammenfassung

In der Implementierung muss dafür gesorgt werden, dass Sequenzierungsunterschiede minimal bis gar nicht vorhanden sind, um Problemen bei der Evaluation vorzubeugen. Dies kann dadurch geschehen, dass die Sequenzierung vorweg stattfindet und den Taggern bereits sequenzierter Text übergeben wird. Es wurde festgestellt, dass zum erfolgreichen Parsen von Goldstandards und Ergebnissen die Tagsets definiert werden müssen. Zusätzlich wurde eine Datenstruktur beschrieben, mit der möglichst viele Informationen zum Tagging-Ergebnis beigelegt werden können. Letztlich wurden die Evaluationsmetriken definiert, mit denen wir die Qualität eines Ergebnisses beurteilen können.

Mit diesen Definitionen und Feststellungen können wir uns nun der tatsächlichen Planung und Implementierung der Erweiterung zuwenden.

4 Implementierung

Dieses Kapitel behandelt die Implementierung der RapidMiner-Erweiterung *postagger*. Zuerst wird grob auf den technischen Rahmen seitens RapidMiner eingegangen, dann werden im Abschnitt 4.2 Ziele der Implementierung angesprochen.
Nach einer Übersicht in Abschnitt 4.3 über die Komponenten, die Funktionen aus Kap. 3 implementieren, werden die Komponenten einzeln im Detail vorgestellt.

4.1 RapidMiner

Die Datenverarbeitungs-Plattform RapidMiner (kurz *RM*) und ihre Erweiterungen sind in Java geschrieben. RapidMiners zentrale Funktionalität ist es, verschiedene Funktionen (*Operatoren*) in einem Modell hintereinanderzuschalten und mit ihnen Daten zu extrahieren und mit verschiedensten Mitteln zu transformieren oder zu untersuchen.

Operatoren funktionieren entfernter betrachtet wie Funktionen in Programmen: Sie nehmen beliebig viele Eingaben mittels *InputPorts* und *Parametern* entgegen, führen eine Aufgabe aus und haben mindestens eine Ausgabe in Form von *OutputPorts*. Alle Operatoren erben von der Superklasse *Operator*. Eine Erweiterung fügt in der Regel solche Operatoren hinzu.

IO-Objekte bzw. IOObjects sind die Datenformate im RM-Modell. Sie werden von der Prozesswurzel sowie von Operatoren ausgegeben und können von Operatoren sowie dem Prozess-Ende entgegengenommen werden. Sie erben von der Klasse *IOObject* und können ebenfalls von Erweiterungen hinzugefügt werden. Operatoren können für ihre Inputs definieren, welche Typ von IOObject sie verlangen. Das Prozessmodell ist nicht lauffähig, wenn diese Spezifikation nicht eingehalten wird.

Eine ausführliche Dokumentation von RM findet sich in [Rapa] und eine Anleitung zum Erstellen von Erweiterungen bei [Rapc]. Die implementierte Erweiterung baut auf der Erweiterung *Text Processing* auf, die mit ihren Operatoren und insbesondere dem Übergabeobjekt *Document* (fortan auch einfach als Dokument bezeichnet) viel Arbeit übernimmt. Das Format Document ist prinzipiell nur ein Container für Texte, wobei mit den Operatoren von *Text Processing* bereits viel auf diesem Text gearbeitet

werden kann. Zusätzlich sind diese Dokumente bereits in der Lage, Sequenzierungen zu kodieren.

Freundlicherweise wurde von der Firma RapidMiner GmbH eine Arbeitslizenz für dieses Projekt zur Verfügung gestellt.

4.2 Ziele

Diese Erweiterung soll u.A. vom auftraggebenden Lehrstuhl weiterverwendet und -entwickelt werden können. Das heißt, eine gute Dokumentation und Strukturierung der Komponenten steht im Vordergrund. Da die Erweiterung nicht unbedingt von Programmierern benutzt werden soll, muss außerdem auf eine niedrige Fehleranfälligkeit der Implementierung geachtet werden.

4.2.1 Erweiterbarkeit

Mit Hilfe von Interfaces, Superklassen und statischen Methoden soll gewährleistet sein, dass durch das Hinzufügen von Komponenten nur minimale bis keine Änderungen an anderen nötig sind.

4.2.2 Robustheit

Verwendung von Operatoren außerhalb ihres Funktionsbereiches soll entweder von vornherein unmöglich sein oder leicht verständliche und eindeutige Fehlermeldungen ausgeben. Auch der Vergleich von nicht identischen Ergebnissen soll weitgehend unterstützt werden und Probleme, wie in Kap. 3.1.1 beschrieben, sollten abgefangen werden.

4.3 Übersicht

Abbildung 4.1 zeigt einen groben Überblick. Es wurden nur Beispiel-Operatoren und -Tagsets eingezeichnet, sodass nur das Konzept verdeutlicht wird. Es folgt eine kurze Erklärung der verschiedenen Komponenten:

Operatoren sind Klassen, die von der RapidMiner-Klasse *Operator* erben, und sind auch in der RapidMiner-Nutzeransicht direkt als solche Operatoren vertreten. Alle Operatoren außer dem *Evaluator* beinhalten einen POS-Tagger und wandeln Ein- als

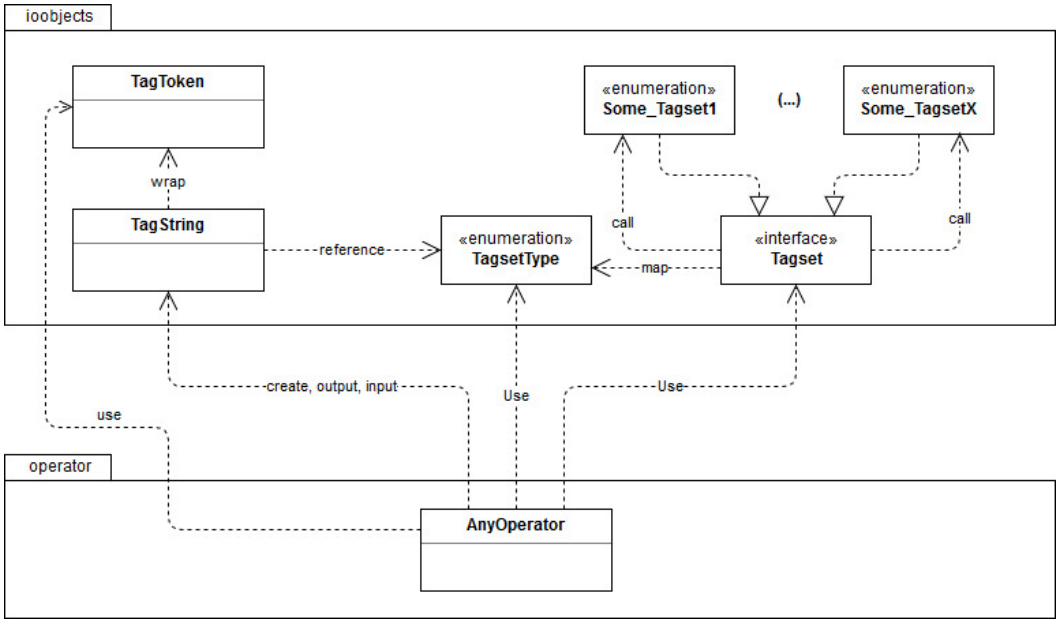

Abb. 4.1: Gekürztes Klassendiagramm der Erweiterung mit Beziehungen

auch Ausgangsformate zwischen dem externen RapidMiner-Modell (*Document* oder *TagString*) und den Vorgaben des Tagging-Algorithmus selbst um. Der Evaluator selbst berechnet die in Kap. 3 vorgestellten Evaluationsmetriken durch Vergleich von zwei Ergebnissen. Die genaue Funktionsweise des Evaluators wird in Abschnitt 4.8 vorgestellt.

Tagsets: Das Interface Tagset muss von jeder Tag-Enumeration, wie zum Beispiel der des Penn-Treebank-Tagsets implementiert werden. Zusätzlich bietet es statische Methoden, die die Werte der Enumeration TagsetType auf die korrespondierenden Enumerationen abbilden, sowie Abfragen über deren Werte anbieten (hierzu wird der Typ sowie ein Tag verlangt). Auf diese Weise kann ein Tagset mit nur seinem Typen adressiert werden und es ist leicht, das Projekt um ein neues Set zu erweitern.

TagString: Diese Klasse bietet, wie in Abschnitt 3.2 angesprochen, ein einheitliches Format für mit Tags versehene Token-Ketten. Sie beschreibt den Tagset-Typen sowie die Zahl an N-Besten Tags pro Token und strukturiert die Tags in Zeilen, die immer dann enden, wenn das letzte Token ein *Separator* ist.

4.4 Tagsets

Das Interface *Tagset* realisiert die Aufführung von POS-Tagsets. Jede Enumeration, die Tagset implementiert, führt eine Gruppe an Tags, wie zum Beispiel die der *Penn*

Abb. 4.2: Klassendiagramm-Abschnitt: Tagsets (ohne Beziehungen)

Treebank [MP 93], auf und wird von einer Konstante in *TagsetType* repräsentiert. Um eine korrekte Evaluation zu ermöglichen, müssen alle Tags, die auftreten können, in der entsprechenden Enumeration vertreten sein (Der Evaluationsmechanismus zählt nur bekannte Tags als potenziell korrekt). Hat ein Tagger eigene definierte Tags, die vom Standard abweichen, sollten auch diese hinzugefügt werden. Die Enumeration muss folgende Funktionen implementieren:

key() liefert den Key des gewählten Tags zurück (falls der Name der Enumerations-Konstante nicht gleich dem Tag ist, würde toString() nicht funktionieren).

isPOS(String key) liefert genau dann TRUE zurück, wenn ein Tag mit dem übergebenen Key existiert und es ein POS-Tag ist.

stringToTagset(String key, boolean strict) liefert die Enumerationskonstante für ein Tag mit dem übergebenen Key zurück, sofern es existiert. Falls STRICT gesetzt ist, wird die Groß- und Kleinschreibung beachtet.

Mapping von TagsetType auf Tagsets

Die überladenen statischen Methoden *IsPOS* und *values* haben jeweils eine Variante mit dem Parameter TagsetType, und eine mit dem Parameter Class. Die TagsetType-Variante ersetzt den Tagset-Typen durch seine korrespondierende Enumeration und ruft dann die andere Variante auf. Die andere Variante operiert dann auf dieser Klasse.

Sinn dieser Aufteilung ist es, dass ein Aufruf nicht mit einer Enumeration oder ihrer Implementierung interagieren muss, sondern sie nur via TagsetType adressiert. Wenn eine Änderung an den implementierenden Tagsets vorgenommen wird, muss diese nur im Interface und in TagsetType bekannt werden. Dadurch senkt sich der Aufwand für eine solche Änderung extrem und die Erweiterbarkeit ist für die Menge der Tagsets gewährleistet.

isPOS(...) (statisch) gibt die Ergebnisse von der Methode isPOS() der entsprechenden implementierenden Enumeration aus.

values(...) (statisch) sammelt alle Keys einer implementierenden Enumeration für die isPOS() mit gesetztem STRICT gilt.

4.5 Eingabe und Tokenization

Zum Einlesen von Text werden bereitgestellte Operatoren von der Erweiterung *Text Processing* (z.B. *read Document*) verwendet. Für diese Aufgabe wurden deshalb keine eigenen Operatoren implementiert. (Satz- und) Wortsequenzierung, wie in 2.1 definiert, wird ebenfalls von *Text Processing* bereits unterstützt. Hierzu kann ein eingelesenes Dokument einfach mit dem Operator *Tokenize* verarbeitet werden. Eine optimale Sequenzierung liefert der Operator, wenn die Einstellung (MODE = LINGUISTIC TOKENS) gesetzt ist.

Dokumente, die bereits in Tokens zerlegt sind, können mit der Methode DOCUMENT# GETTOKENTEXT(): STRING abgerufen werden und geben die Token mit Leerzeichen getrennt zurück. Dieses Format ist dann leicht in andere Formate, wie z.B. Listen, zu zerlegen. Um, wie in Abschnitt 3.1 erklärt, Sequenzierungsprobleme zu verhindern, wird Sequenzierung in den Tagging-Operatoren gezielt weggelassen und stattdessen wird ein Davorschalten des Operators *Tokenize* wie oben beschrieben verlangt.

4.6 Tag-String als Ergebnisformat

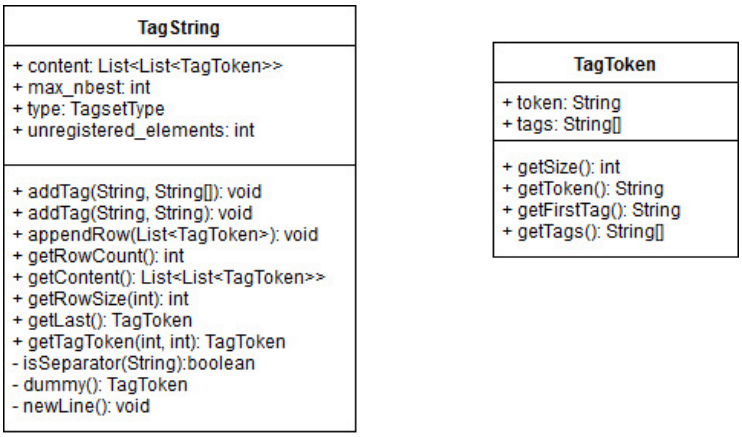

Abb. 4.3: Klassendiagramm-Abschnitt: TagString (ohne Beziehungen)

Das in 3.2 konzeptuell eingeführte Ergebnisformat wird von der Klasse *TagString* realisiert. Solche TagString-Objekte werden vom Parser des Evaluations-Operators

(siehe Abschnitt 4.8.1) und von Tagging-Operatoren erzeugt und vom Evaluator verwendet. TagStrings sind vorwiegend dazu entworfen, Iteration über Tag-Token-Ketten zu erleichtern und Sequenzierungsfehlern zu entgehen.

4.6.1 Metainformationen

Der TagString bietet die Möglichkeit, beim Erzeugen seinen Tagset-Typen [TAGSETTYPE TYPE] festzulegen und zu definieren, wie viele N-Best (Variable [INT MAX_NBEST]) Tags pro Token gespeichert werden sollen. Wird beim Hinzufügen eines Tokens die falsche Menge an Tags hinzugegeben, stellt der TagString von selbst sicher, dass so viele überflüssige Tags entfernt bzw. ungültige Tags hinzugefügt werden, dass die angegebenen MAX_NBEST Tags vorhanden sind. Der Tagset-Typ wird genutzt, um mittels der statischen Methoden von *Tagset* zu prüfen, ob hinzugefügte Tags (bei mehreren pro Token nur das erste) Teil ihres Tagsets sind. Falls nicht, wird das Tag zwar hinzugefügt, aber der Zähler [INT UNREGISTERED_ELEMENTS] wird inkrementiert.

4.6.2 TagToken

Die TagToken-Objekte sind die Elemente der TagString-Struktur. Sie enthalten ein Token und ein Array an Tags, wobei das erste Tag im Array das First-Best Tag ist. Die Bestandteile können mit den entsprechenden GET-Methoden abgefragt werden. Die Methode [GETSIZE(): INT] liefert die Größe des Tag-Arrays zurück.

TagTokens wurden eingeführt, um Informationen hinzufügen zu können, ohne Änderungen im restlichen Code zu veranlassen.

4.6.3 Aufbau des TagStrings

Die verschachtelte Liste CONTENT, die die TagToken-Objekte sortiert, übernimmt nicht nur die Wort-, sondern auch Satzsequenzierung der Tagging-Ergebnisse. Die innere Liste stellt Sätze dar und wird von der äußeren in Reihenfolge gebracht.

Adressierung von Token

Um das Iterieren über CONTENT zu ermöglichen, ohne die Struktur selbst kopieren zu müssen, wurden Hilfsmethoden implementiert. Mit GETROWCOUNT() wird die Zahl an Sätzen und mit GETROWSIZE(INT) die Zahl an TagToken in einem

spezifizierten Satz abgefragt. Mit diesen Informationen kann jedes TagToken via GETTAGTOKEN(INT, INT) ausgegeben werden. Ist die Adressierung falsch, wird DUMMY() aufgerufen, um ein neues TagToken mit ungültigen Tags ("NONE") zurückzugeben. Dadurch sind Iterationen robuster gegen Fehler.

Serialisierung

Falls die Zerlegung in Sätze nicht erwünscht ist, kann sie mit der Methode SERIALIZE() aufgelöst werden. Dann werden alle inneren Listen zu einer zusammengefasst.

Hinzufügen von Token

Beim Hinzufügen von Tagging-Ergebnissen mittels ADDTAG(...) prüft der TagString selbstständig mit Hilfe der Methode ISSEPARATOR(), ob das hinzugefügte Token ein Zeilenseparator ist. Als solche Separatoren werden von der Hilfsmethode alle Token identifiziert, die mit Satzzeichen (Punkt, Fragezeichen etc.) beginnen. Ist dies der Fall, wird automatisch die aktuelle Zeile beendet und eine Neue begonnen.

Den Zeilenumbruch übernimmt der TagString selbst, um eine gleiche Satzsequenzierung verschiedener Ergebnisse zu garantieren, sofern bei diesen die Wortsequenzierung entweder identisch ist oder zumindest jedes Satzzeichen korrekt als eigenes Token abgespalten wurde.

4.7 Implementierte Tagging-Operatoren

Jeder implementierte Tagging-Operator hat folgendes In- und Outputverhalten: Eingelesen wird ein sequenziertes Dokument (es reicht aus, wenn jedes Token durch ein Leerzeichen von Vorgänger und Nachfolger getrennt ist), und ausgegeben dasselbe Dokument, wobei nach jedem Token ein „ “ und das Tag folgen. Zusätzlich wird das Tagging-Ergebnis im Format TagString ausgegeben.

NLP4J (Dokumentation: [16]) ist der laut eigenen Angaben leistungsfähigste der implementierten POS-Tagger. Er kann bei der Initialisierung mit trainierten Modellen und Lexika konfiguriert werden. Das eröffnet die Möglichkeit, diese Modelle per Parameter wählbar zu machen, oder sogar via Text-Parameter eigene Modelle und

Lexika einzufügen. In dieser Implementierung sind Modell und Lexikon jedoch festgesetzt.

LingPipe (Dokumentation: [11])ist, wie der NLP4J-Algorithmus, mit einem Modell initialisierbar. Die drei verschiedenen vortrainierten Modelle, die der Implementierung beigefügt wurden, sind zu Demonstrationszwecken alle in einem Parameter wählbar gemacht worden. Allerdings liefert nur das Modell, das auf dem *GENIA Korpus* [Kim+03] trainiert wurde, POS-Tags zurück, die dem *Penn Treebank* Tagset entsprechen. Da der Evaluations-Testlauf in Kapitel 5 sich nur auf letzteres Tagset bezieht, sind andere Tagsets noch nicht als entsprechende Enumerationen implementiert.

FastTag (Quellcode und Dokumentation: [Wat])ist ein simpler, nicht auf Trainingsdaten beruhender POS-Tagger. Er wurde implementiert, um einen Vergleich zu den anderen, lernenden POS-Taggern zu bieten.

4.8 Evaluations-Operator

Der Operator *Evaluator* ist die Zentrale Komponente dieser Erweiterung. Er muss die Formate von Goldstandards sowie Tagging-Ergebnisse annehmen und interpretieren können. Neben dem eigentlichen Evaluieren ist also auch das *Parsen* eingehender Texte eine wichtige Aufgabe des Operators.

4.8.1 Parsing von Ergebnissen

Werden Ergebnisse als Typ Document ausgegeben oder wird ein Goldstandard mit Operatoren aus *Text Processing* eingelesen, liegt ein Document-Objekt mit einem Kodierungsformat vor. Dieses Dokument muss mittels Parsing-Techniken in das TagString-Format transformiert werden, damit eine Evaluation darüber stattfinden kann.

Verschiedene Formate kodieren POS-Tags unterschiedlich. Eine einfache Notation ist die bereits aus der Einleitung bekannte Variante, bei der nach jedem Token ein „\“ und dann das Tag folgt. Andere Notationen kodieren zusätzlich Satzstrukturinformationen, sind also geschachtelt, um z.B. Teilsätze zu markieren. Hier gibt es kein Symbol, das eindeutig ein POS-Tag ankündigt. Um die Parser-Methode des Evaluationsoperators präziser arbeiten können zu lassen, wurde ein Parameter hinzugefügt, in dem man das eingelesene Format und damit den Modus des Parsers

nennen kann. Es wurden drei Modi (und damit wählbare Parameter) für den Parser implementiert:

Backslash-Notation: [Smi11] In der oben genannten Notation, in der ausschließlich POS via „\“ markiert werden, ist Parsing einfach. In dieser Notation wird jedes Token mit einem Leerzeichen von seinem Vorgänger und Nachfolger getrennt. Letztlich muss nur noch die Struktur

$$WORD\backslash TAG$$

gefiltert werden.

Parenthesis-Notation bzw. Penn-Treebank-Standard: [MP 93] In dieser Notationsvariante wird mittels Klammern die Satzstruktur zerlegt. Wichtig ist, dass Token und Tags immer im Format (TAG TOKEN) bzw. (TOKEN TAG) auftreten. Ferner kann ein Tag also nur auftreten, wenn die Struktur

$$\text{"("}\ WORD\ WORD\ \text{")"}$$

erscheint. Es ist sogar anzunehmen, dass diese Struktur *ausschließlich* Token-Tag-Kombinationen kodiert. Der Parser muss also nur diese Struktur finden und identifizieren, ob eines der beiden Worte ein gültiges POS-Tag ist.

None: Falls die Notation unbekannt ist, versucht der Parser, den Text an allen sinnvollen Symbolen, insbesondere dem Leerzeichen, zu trennen. Alle Substrings werden dann überprüft, ob sie ein POS-Tag sind. Hier können die Token allerdings nicht identifiziert werden und es handelt sich nur um eine Ausweichs-Option.

Ein neuer Parsing-Modus kann der Methode einfach hinzugefügt werden. Zusätzlich kann die Option *„Ignore Brackets“* gewählt werden, falls Klammern für die Formatierung des einzulesenden Textes verwendet wurden (In der Parenthesis-Notation erübrigt sich das), diese werden dann ignoriert.
Der Tagset-Typ des eingelesenen Dokuments und des ausgegebenen TagStrings muss via Parameter angegeben werden.

4.8.2 Iteration über TagStrings

Vergleichsfunktionen müssen nur TagStrings entgegennehmen, da andere Formate vom Parser in solche umgewandelt werden. Aufgrund der automatischen Unterteilung von Token-Ketten in Sätze durch TagStrings kann nicht nur Token- sondern

auch Satzweise verglichen werden. Iteriert wird also erst über die Sammlung an Sätzen, dann über deren individuelle Token.

Bei identischer Wort-Sequenzierung der beiden Ergebnisse erübrigen sich Überlegungen über die Robustheit des Vergleichsverfahrens. Hier kann davon ausgegangen werden, dass an gleichen Positionen im TagString auch gleiche Token platziert sind. Bei der Iteration werden also keine Probleme entstehen.

Bestehen allerdings Sequenzierungsunterschiede (siehe Abschnitt 3.1), kann korrektes Iterieren nicht mehr garantiert werden. Darum wird zuerst überprüft, ob die Satzsequenzierung übereinstimmt. Von einer korrekten Sequenzierung wird ausgegangen, wenn beide zu vergleichenden TagStrings dieselben Satz-Anahlen mit COUNTROWS() zurückliefern. Ist dies *nicht* der Fall, muss die Satzsequenzierung aufgelöst werden. Dazu wird bei den TagStrings SERIALIZE() aufgerufen (Beschreibung siehe Abschnitt 4.6). Danach wird wie zuvor iteriert, wobei beide TagStrings effektiv nur noch einzeilig sind, und Wort-Sequenzierungs-Unterschiede nicht abgefangen werden können. Ist die Satzsequenzierung hingegen korrekt, würden solche Unterschiede beim Iterieren über die Token allerdings nur Probleme innerhalb des Satzes verursachen, da beim nächsten untersuchten Satz die Iteration neu starten kann.

Daraus wird erkennbar, warum bei ungleicher Sequenzierung kein garantiert optimaler Vergleich mehr stattfinden kann. Für korrekte Sequenzierung muss also, wenn möglich, unbedingt schon vorher im Prozess gesorgt werden.

4.8.3 Berechnung der Metriken

Mittels dem erklärten Iterations-Vorgehen aus dem letzten Abschnitt und den Definitionen aus Abschnitt 3.4 kann die Berechnung der Evaluationsmetriken beim Vergleich zweier TagStrings leicht erklärt werden.

Accuracy

Per-Tag-Accuracy berechnet sich aus der Zahl der beim Vergleichen der TagStrings identischen Tags geteilt durch die Gesamtzahl der Token (bei ungleicher Satzlänge wird die größere Anzahl an Token zur Gesamtzahl hinzugerechnet). Die Per-Sentence-Accuracy besteht analog aus der Zahl absolut identischer Sätze, geteilt durch deren Gesamtzahl.

Confusion-Matrix

Die Confusion-Matrix wird wie folgt gebildet: Zuerst muss eine Liste der auftretenden Tags gebildet werden. Diese wird vom Interface *Tagset* statisch angefragt. Die Matrix hat exakt so viele Zeilen und Spalten wie Tags und deren Index repräsentiert das entsprechende Tag in der Tag-Liste. Die Matrix-Positionen MATRIX[X][Y] repräsentieren Token, deren Goldstandard-Tag in der Tag-Liste Index X hat, und deren Ergebnis-Tag Index Y hat. Per Definition sind für Tag mit Index X die entsprechenden Werte also wie folgt zu berechnen (I sei hier der maximale Index der Tag-Liste):

$$True\ Positive\ (X)\ =\ matrix[X][X]$$

$$False\ Positive\ (X)\ =\ \left(\sum_{i=0}^{I} matrix[i][X]\right) - matrix[X][X]$$

$$False\ Negative\ (X)\ =\ \left(\sum_{i=0}^{I} matrix[X][i]\right) - matrix[X][X]$$

True Negatives sind für die berechneten Metriken *Precision, Recall und F-Score* nicht interessant. Precision und Recall sind dementsprechend:

$$Precision(X)\ =\ \frac{matrix[X][X]}{\sum_{i=0}^{I} matrix[i][X]}$$

$$Recall(X)\ =\ \frac{matrix[X][X]}{\sum_{i=0}^{I} matrix[X][i]}$$

Der F-Score wird wie in Kapitel 3 definiert als harmonisches Mittel von Precision und Recall berechnet. Die Rechnung $\frac{0}{0}$ wird abgefangen, sodass stattdessen null als Ergebnis ausgegeben wird.

N-Best Tags

Wurde beim Parameter *Calculate N-Best Distance* ein Haken gesetzt, werden zwei weitere Metriken berechnet:

N-Accuracy: Diese Variante der Accuracy wird analog zur Per-Tag-Accuracy berechnet. Allerdings werden alle Tags pro Token im Tagging-Ergebnis mit dem korrekten Tag im Goldstandard verglichen. Findet sich unter allen Tags ein Korrektes, wird wie bei der normalen Per-Tag-Accuracy von einem korrekten Tag ausgegangen. Dieser Wert ist also mindestens so hoch wie die Per-Tag-Accuracy desselben TagStrings.

N-Distanz: Um genauere Einsicht darauf zu liefern, an welcher Stelle das durchschnittliche korrekte Tag unter den besten n stand, wird die durchschnittliche Distanz eingeführt. Ist das erste Tag korrekt, beträgt die Distanz für dieses Token 1, ist das zweite korrekt, ist sie 2, und so weiter. Wird das korrekte Tag unter den besten n nicht gefunden, lautet die Distanz n+1. Die N-Distanz für einen ganzen TagString ist dann der Durchschnitt aller Distanzen pro Token.

4.8.4 Ausgabe

Da diese Erweiterung zur Weiterentwicklung konzipiert wurde und in eine größere Bilbiothek übernommen werden wird, wurde die Definition eines Evaluations-Formats absichtlich weggelassen.

Als Ersatz liefert der Evaluationsoperator ein Dokument aus, in dem alle berechneten Ergebnisse (und falls gewünscht auch die Confusion-Matrix) textuell wiedergegeben werden. Zusätzlich gibt ein zweiter Output das Ergebnis des Parsers beim eingelesenen Goldstandard aus, sodass überprüft werden kann, ob der Parsing-Prozess korrekt abgelaufen ist.

4.9 Zusammenfassung

Der Evaluationsoperator bietet Einsichten über verschiedene Evaluationsmetriken beim Vergleich zweier Tagging-Ergebnisse, wobei einer davon als Goldstandard betrachtet wird. Sind beide Ergebnisse kein Goldstandard, können die beiden *Accuracy*-Werte als Übereinstimmung der Ergebnisse interpretiert werden. Als Tagging-Algorithmen wurden drei unterschiedlich konfigurierbare POS-Tagger in Operatoren verpackt. Während die Tagger ihre Ergebnisse in Textform ausgeben und der Evaluator diese einlesen können, wurde für eine informationsreichere Übergabe von Daten die Datenstruktur TagString definiert. Zusätzlich wurden die verwendbaren Tagsets als Enumerationen definiert.

Mehrere Methoden wurden angewendet, sodass die Erweiterung leicht um neue Tagsets oder Informationen im TagString vergrößern werden kann. Es wurde auch darauf geachtet, dass der Evaluationsoperator möglichst nicht fehleranfällig ist und im Falle eines Fehlers nützliche Informationen an den Nutzer ausgibt.

5 Evaluation

Tagger (Trainingskorpus)	Performance(eigen)	Performance(test)
NLP4J(WSJ)	97.64%	94,86%
LingPipe(GENIA)	96.9%	77,26%
FastTag(-)	-	30,72%

Tab. 5.1: Liste der Implementierten Tagger.

In diesem Kapitel wird ein Test mit einem fremden Goldstandard an den implementierten Taggern ausgeführt. In Tab. 5.1 sind die Per-Tag-Accuracies der Tagger aufgeführt. Performance(eigen) ist die Per-Tag-Accuracy laut eigenen Angaben (Quellen in den weiterführenden Abschnitten), Performance(test) die Per-Tag-Accuracy in diesem Test. Es ist hierbei jedoch unbedingt anzumerken, dass in diesem Test *keine* Vergleiche zwischen den Taggern gezogen werden können. Hierzu müssten alle Tagger, bei denen dies möglich ist, auf den Korpus trainiert werden, um den *Bias* der Tagger anzugleichen und die Ergebnisse für diesen speziellen Korpus dadurch vergleichbar zu machen.

Allerdings kann man aus einem Vergleich der eigenen Angaben mit den Testergebnissen sehen, wie stark die Performance-Änderung ist, wenn man dem Tagger einen fremden Korpus zuführt. Da auch die Trainingskorpora der Tagger sich unterscheiden, kann allerdings auch diese Änderung nicht als Vergleich zwischen den Taggern genutzt werden.

Die folgenden Abschnitte diskutieren die Auswahl eines passenden Korpus und die Performance (*Scores*) der Tagger auf diesem.

5.1 Goldstandard-Korpus und Aufbau des Tests

Ein annotierter Korpus muss natürlich dem Tagset der einzelnen zu testenden Tagger entsprechen. Zudem sollte aus dem Goldstandard direkt die Sequenzierung auslesbar sein, da sonst Tokenization stattfinden müsste und Sequenzierungsunterschiede die Ergebnisqualität der Tagger trüben könnten.

Ein Korpus, der beiden Kriterien entspricht, ist die *NAIST/NTT TED Treebank* [Neu+14]. Alle Inhalte sind sowohl als *Tokenized* (Token getrennt durch Leerzeichen und Zei-

lenumbrüche) als auch annotiert nach derselben Sequenzierung verfügbar. Die zur POS-Tag-Evaluation überflüssigen Syntaxbaum-Tags können einfach ignoriert werden. Die Notation des Annotierten Textes entspricht der Klammer-Notation aus Abschnitt 4.8.1 und kann somit vom Parser des Evaluationsoperators erkannt werden.

Im Rahmen des Tests wird diese Treebank verwendet. Der RapidMiner-Prozess zum Testen ist in Abb. 5.1 dargestellt. Für die nachfolgenden Tests wurden sämtliche Teilstücke des Testkorpus eingelesen und zu einem Dokument aneinandergereiht. Zur Übersichtlichkeit wurde in der Abbildung allerdings nur einfaches Einlesen von einem Teilstück modelliert.

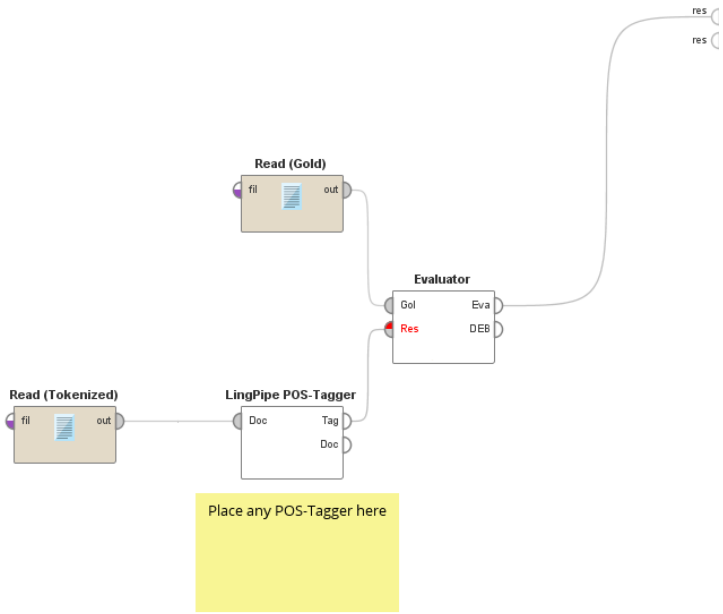

Abb. 5.1: Aufbau des Test-Evaluationsprozesses

Der LingPipe-Tagger im Prozess hat identische Inputs und Outputs zu den anderen Taggern, und kann hier einfach ersetzt werden. Die Einlese-Operatoren lesen für den Tagger die sequenzierten Texte mit der Dateiendung .TOK und als Goldstandard die annotierten Texte in Klammer-Format mit Endung .MRG. Der Evaluationsoperator *Evaluator* wird auf diese Formatierung konfiguriert.

5.2 Evaluation der einzelnen Tagger

5.2.1 NLP4J (WSJ)

Der NLP4J-Tagger erreicht gemäß der Werte in Tab. 5.2 beinahe seine maximale Leistungsfähigkeit(97.64% [16]). Daraus lässt sich schließen, dass der *Bias* in Richtung

Metrik	Performance
Per-Tag-Accuracy	94,86%
Per-Sentence-Accuracy	49,19%
Precision(NN)	96,11%
Recall(NN)	96,85%
F-Score(NN)	96,48%
Precision(VBD)	95,11%
Recall(VBD)	95,7%
F-Score(VBD)	95,4%

Tab. 5.2: Scores des NLP4J-Taggers (Training: WSJ-Korpus)

des WSJ-Korpus, auf dem trainiert wurde, zu keinen sehr hohen Performanceverlusten beim Testkorpus führt, also eine hohe Ähnlichkeit zwischen den Korpora herrscht. Alternativ könnte der NLP4J-Tagger auch hochgradig unanfällig gegen Bias sein, allerdings müssten für solch eine Aussage mehr Tests stattfinden.

Auffällig ist die Per-Sentence-Accuracy von knapp 50%, jedoch klärt eine kurze Rechnung das schnell auf: Bei einer Wahrscheinlichkeit von 94,86% und einer (angenommenen) durchschnittlichen Satzlänge von 13 Wörtern beträgt die Wahrscheinlichkeit, dass der Satz vollständig korrekt ist (also dass 13 Wörter in Folge korrekt sind) $0,9486^{13} \approx 50,36\%$. Bei den eingelesenen 23.158 Wörtern und 1.486 daraus gelesenen Sätzen beträgt die tatsächlich durchschnittliche Satzlänge $\frac{23.158}{1.486} \approx 15.58$. Die kleine Diskrepanz von 2,58 Worten zwischen Annahme und Ergebnis lässt sich mit der ungleichen Verteilung der Satzlängen im Korpus erklären. Dies wird hier nicht weiter überprüft, aber da der Korpus aus Untertiteln von *TED Talks*, also Reden entnommen ist, erscheint das plausibel.

5.2.2 LingPipe (GENIA)

Metrik	Performance
Per-Tag-Accuracy	77,26%
Per-Sentence-Accuracy	10,96%
N-Accuracy(n=5)	77,32%
N-Distanz(n=5)	2,134
Precision(NN)	62,65%
Recall(NN)	77,42%
F-Score(NN)	69,25%
Precision(VBD)	81%
Recall(VBD)	73,36%
F-Score(VBD)	76,99%

Tab. 5.3: Scores des LingPipe-Taggers (Training: GENIA Korpus)

In Tab. 5.3 finden sich die Scoring-Ergebnisse des Ling-Pipe Taggers, konfiguriert mit den Trainingsergebnissen auf Basis des GENIA Korpus. Laut Tests in [Wil09] erreicht LingPipe eine Per-Tag-Accuracy von bis zu 96.9%. Diese wurde hier bei weitem nicht erreicht. Allerdings ist der GENIA Korpus, auf dem trainiert wurde, eine Sammlung an biomedizinischem Fachwissen, daher ist die Leistungsminderung verständlich.

Besonders an diesem Tagger ist, dass er potenziell n beste Tags pro Token ausgibt, daher wurde hier eine N-Accuracy und N-Distanz für n = 5 berechnet. Allerdings fällt bei Inspektion des TagStrings auf, dass die 5 Tags pro Token in diesem Fall fast immer identisch waren (es liegt die Vermutung nahe, dass das zugrundeliegende Modell nicht zu mehr Differentiation in der Lage war). Daher ist die N-Accuracy nur marginal höher und die N-Distanz ist nahe dem Minimum von $(n+1)-Accuracy*n \approx 6-0,7726*6=2,13$.

5.2.3 FastTag

Metrik	Performance
Per-Tag-Accuracy	30,72%
Per-Sentence-Accuracy	0,87%
Precision(NN)	15,92%
Recall(NN)	89,39%
F-Score(NN)	27,02%

Tab. 5.4: Scores des FastTag-Taggers

FastTag erreicht nur eine extrem niedrige Accuracy. Um diese zu erklären, können wir einen Blick auf die Confusion-Matrix-Metriken für das Tag NN werfen: Es fällt sofort auf, dass der Recall-Wert ungewöhnlich hoch ist. Der Tagger rät bei Wörtern, die nicht im Lexikon vertreten sind, immer das Tag NN. Ein hoher Recall bedeutet, dass viele der NN-Tags im Goldstandard erkannt wurden. Auf der anderen Seite sagt die niedrige Precision jedoch aus, dass viele der geratenen NN falsch waren. Es wird also erkennbar, dass dieser Taggingprozess durch viele Rateversuche gekennzeichnet war und vor allem an einem zu kleinen Lexikon gescheitert ist. Zusätzlich verwendet FastTag auch keinerlei Worttransformationsmethoden, die dieses Problem verhindern könnten.

Die Metriken für das Tag VBD wurden ausgelassen, da der Tagger dieses Tag nicht verwendet.

5.3 Zusammenfassung

Bereits die Testergebnisse aus dieser überschaubaren Untersuchung liefern einen hohen Informationsgehalt. Es lässt sich beobachten, wie extrem sich der Bias der beiden trainierten POS-Tagger auf deren Scores auswirkt.

6 Zusammenfassung und Ausblick

Die implementierte RapidMiner-Erweiterung POSTAGGER baut auf der Erweiterung *Text Processing* auf und liefert Möglichkeiten, mit unterschiedlich konfigurierbaren Operatoren Part-of-Speech-Tagging zu betreiben. Die dabei entstehenden Ergebnisse können vom Evaluationsrahmenwerk *Evaluator* eingelesen werden und mit anderen Ergebnissen oder einem Goldstandard verglichen werden. Der Vergleich liefert viele Informationen über die Qualität des evaluierten Ergebnisses. Die gesamte Erweiterung ist so strukturiert, dass ein Hinzufügen von Part-of-Speech-Taggern und Tagsets möglichst einfach und ohne viele Fehlerpotenziale stattfinden kann.

Allerdings ist zum Verwenden der meisten Tagging-Operatoren eine professionelle Lizenz oder Signierung der Erweiterung seitens der RapidMiner GmbH notwendig, da diese Operatoren aus Effizienzgründen stark parallelisiert sind und die Java-Sicherheitsfunktionen von RapidMiner diese Parallelisierung für unsignierte Erweiterungen i.d.R. blockieren.

In Zukunft sind zur Weiterentwicklung viele Optionen denkbar: Die Tagging-Operatoren könnten dynamisch mit externen Modellen und Lexika initialisiert werden, möglicherweise könnte man sogar Operatoren entwerfen, mit denen Modelle und Lexika trainiert und übergeben werden können. Der Evaluations-Operator könnte auch noch um beliebige Metriken erweitert werden. Wie schon im Abschnitt 4.8.4 angesprochen, wurde auch noch kein Ausgabeformat für Evaluationen neben einer einfachen Text-Ausgabe implementiert.

Literatur

[11] *Text Analysis with LingPipe 4*. LingPipe Publishing, 2011 (zitiert auf den Seiten 8, 20).

[16] *Dynamic Feature Induction: The Last Gist to the State-of-the-Art*. 2016 (zitiert auf den Seiten 8, 19, 26).

[al03] C. Manning et al. „Feature-Rich Part-of-Speech Tagging with a Cyclic Dependency Network“. In: *Proceedings of HLT-NAACL 2003* (2003), S. 252–259 (zitiert auf den Seiten 6, 8).

[C R18] Venkat N. Gudivada C. R. Rao. *Computational Analysis and Understanding of Natural Languages: Principles, Methods and Applications*. Elsevier Science, 2018 (zitiert auf den Seiten 1, 10).

[D B97] H. Somers D. B. Jones. *New Methods in Language Processing*. UCL Press, 1997 (zitiert auf den Seiten 3, 5).

[Hal99] H. van Halteren. *Syntactic Wordclass Tagging*. Text, Speech and Language Technology. Springer Netherlands, 1999 (zitiert auf den Seiten 4, 5).

[Kim+03] Jin-Dong Kim, Tomoko Ohta, Yuka Tateisi und Jun'ichi Tsujii. „GENIA corpus—A semantically annotated corpus for bio-textmining“. In: *Bioinformatics (Oxford, England)* 19 Suppl 1 (Feb. 2003), S. i180–2 (zitiert auf Seite 20).

[MP 93] B. Santorini M.P. Marcus M. A. Marcinkiewicz. „Building a large annotated corpus of English: the penn treebank“. In: *Computational Linguistics - Special issue on using large corpora: II* Volume 19 Issue 2, June 1993 (1993), S. 313–330 (zitiert auf den Seiten 4, 16, 21).

[Neu+14] Graham Neubig, Katsuhito Sudoh, Yusuke Oda et al. „The NAIST-NTT TED Talk Treebank“. In: *11th International Workshop on Spoken Language Translation (IWSLT)*. Lake Tahoe, USA, Dez. 2014 (zitiert auf Seite 25).

[Par07] P. Paroubek. „Evaluating Part-of-Speech-Tagging and Parsing“. In: *Evaluation of Text and Speech Systems. Speech and Language Technology, vol. 37*. Springer, Dordrecht, 2007. Kap. 4 (zitiert auf den Seiten 6, 7).

[Smi11] Noah Smith. *Linguistic Structure Prediction*. Morgan und Claypool Publishers, 2011 (zitiert auf den Seiten 1, 3–7, 21).

[Sor13] Anders Sorgaard. *Semi-Supervised Learning and Domain Adaptation in Natural Language Processing*. Morgan und Claypool Publishers, 2013 (zitiert auf Seite 5).

[Van14] F. Van Eynde. *Lexicon Development for Speech and Language Processing*. Text, Speech and Language Technology. Springer Netherlands, 2014 (zitiert auf den Seiten 4, 5).

Websites

[Gan14] Kavita Ganesan. *Computing Precision and Recall for Multi-Class Classification Problems*. 2014. URL: http://text-analytics101.rxnlp.com/2014/10/computing-precision-and-recall-for.html (besucht am 16. Apr. 2019) (zitiert auf den Seiten 10, 11).

[Gmb18] RapidMiner GmbH. *Text processing Plugin Site*. 2018. URL: https://marketplace.rapidminer.com/UpdateServer/faces/product_details.xhtml? (zitiert auf Seite 1).

[Rapa] RapidMiner GmbH. *RapidMiner Documentation*. URL: https://docs.rapidminer.com/ (zitiert auf Seite 13).

[Rapb] RapidMiner GmbH. *RapidMiner Studio Beschreibung*. URL: https://rapidminer.com/products/studio/ (besucht am 21. Apr. 2019) (zitiert auf Seite 1).

[Rapc] RapidMiner GmbH. *RapidMiner: Creating your own Extension*. URL: https://docs.rapidminer.com/latest/developers/creating-your-own-extension/ (zitiert auf Seite 13).

[Sta03] Stanford University. *Penn Treebank P.O.S. Tags*. 2003. URL: https://www.ling.upenn.edu/courses/Fall_2003/ling001/penn_treebank_pos.html (besucht am 4. Apr. 2019) (zitiert auf Seite 4).

[Wat] mark Watson. *FastTag v2*. URL: https://github.com/mark-watson/fasttag_v2 (zitiert auf Seite 20).

[Wil09] Matt Wilkens. *Evaluating POS Taggers: LingPipe Cross-Validation*. 2009. URL: https://mattwilkens.com/2009/01/21/evaluating-pos-taggers-lingpipe-cross-validation/ (besucht am 21. Apr. 2019) (zitiert auf Seite 28).

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, 23.April 2019

Philipp Scholz