Friedrich-Alexander-Universität Erlangen-Nürnberg
Technische Fakultät, Department Informatik

LUC ZUMTAUGWALD
BACHELOR THESIS

EXTRAKTION VON BRANCHENINFORMATIONEN AUS UNSTRUKTURIERTEN TEXTEN

Eingereicht am 8. August 2021

Betreuer:
Prof. Dr. Dirk Riehle, M.B.A.
Julia Krause, M.Sc.
Professur für Open-Source-Software
Department Informatik, Technische Fakultät
Friedrich-Alexander-Universität Erlangen-Nürnberg

Versicherung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Erlangen, 8. August 2021

License

Erlangen, 8. August 2021

Abstract

Nowadays, a huge amount of information is accessible through the internet. Since a large portion of this data exists in the form of unstrucutured text, it is necessary to extract the knowledge and convert it into a structured format. Machine learning models, that are used for this task, need a lot of manually labeled data in order to be trained. However, manually labeling the data is a time consuming task.

The goal of this thesis was to design an extraction pipeline, which combines rulebased approaches with machine learning without the need of manually labeled data. This goal has been fullfilled. It could be shown that it is possible to train a neural network with programmatically labeled data and to use it to extract information from german industry reports. For that, a pipeline which generates training data using pattern matching has been developed. The trained model could achieve results equivalent to a purely rulebased approach.

Zusammenfassung

In Zeiten des Internets sind Informationen für jedermann zugänglich. Da viele dieser Daten jedoch in unstrukturierten Texten vorliegen, ist es notwendig die darin enthaltenen Wissenselemente zu extrahieren und in eine strukturierte Form zu überführen. Maschinelle Lernverfahren, die hierfür verwendet werden, erfordern große Mengen an manuell annotierten Datensätzen. Allerdings benötigt die manuelle Annotierung der Daten zeitliche und personelle Ressourcen, die vielen Unternehmen nicht zur Verfügung stehen.

Das Ziel dieser Arbeit bestand darin, eine Extraktionspipeline zu entwerfen, die regelbasierte Vorgehensweisen mit maschinellem Lernen verbindet und ohne manuellen Annotierungsaufwand auskommt. Dieses Ziel wurde erreicht, denn es wurde gezeigt, dass es möglich ist, ein neuronales Netz mit programmatisch annotierten Daten zu trainieren und damit Brancheninformationen aus Texten zu extrahieren. Dafür wurde eine Regelpipeline entworfen, die mithilfe von Mustervergleichen einen Trainingskorpus für ein neuronales Netz generiert. Das damit trainierte Modell konnte gleichwertige Ergebnisse liefern wie ein rein regelbasierter Ansatz.

Inhaltsverzeichnis

Abbildungsverzeichnis

Quelltextverzeichnis

1 Einleitung

1.1 Problemstellung

Das World Wide Web stellt heutzutage eine unerschöpfliche Wissensquelle dar, die jederzeit mit nur einem Klick an jedem Ort abgerufen werden kann. Zur effizienten Verarbeitung der gespeicherten Informationen durch Computer, müssen diese in strukturierter Form (z.B. in einer Datenbank) vorliegen. Da viele dieser Daten jedoch in unstrukturierten Texten vorliegen, ist es notwendig die darin enthaltenen Wissenselemente zu extrahieren und in eine strukturierte Form zu überführen.

Das Unternehmen DATEV eG stellt seinen Kunden über den Dienst LEXinform eine Vielzahl an Branchenberichten zur Verfügung, die jährlich von verschiedenen Finanzinstituten veröffentlicht werden und Informationen zur künftigen Entwicklung bestimmter Kennzahlen der entsprechenden Branche enthalten. Im Rahmen von betriebswirtschaftlichen Planungsprozessen sind u.a. solche Entwicklungen zu berücksichtigen, weshalb Softwareprodukte zur Planung davon profitieren könnten, wenn die Daten strukturiert und semantisch interpretierbar vorlägen. Da die Informationen momentan jedoch nur in Form von unstrukturierten Textdaten zugänglich sind, können sie nicht elektronisch weiterverarbeitet werden.

Der Großteil der aktuellen Forschung in der Informationsextraktion legt ihr Augenmerk auf überwachte maschinelle Lernverfahren wie bspw. Deep Learning. Trotz des enormen Potentials dieser Technik, hat sie den Nachteil, dass eine große Menge an annotierten Datensätzen erforderlich ist, um ein akzeptables Ergebnis zu erhalten. Die manuelle Annotierung der Daten erfordert zeitliche und personelle Ressourcen, die vielen Unternehmen nicht zur Verfügung stehen.

1.2 Ziel der Arbeit

Das Ziel dieser Bachelorarbeit besteht darin, Wege aufzuzeigen, wie trotz eines Mangels an manuell vorannotierten Textdaten Ergebnisse in der Informationsextraktion erzielt werden können, die dem aktuellen Stand der Technik entsprechen. Hierfür soll ein Prototyp entwickelt werden, der aus unstrukturierten deutschsprachigen Texten in Form von Branchenberichten Informationen über die entsprechende Branche extrahiert. Die Art der gewonnen Information soll sich auf die zeitliche Entwicklung bestimmter Kenngrößen (z.B Umsatzentwicklung) in Prozent beschränken. Die gewonnenen Daten sollen strukturiert und semantisch interpretierbar abgespeichert werden, um im Anschluss zur elektronischen Weiterverarbeitung zur Verfügung zu stehen.

Im Rahmen der Bachelorarbeit soll die folgende zentrale Forschungsfrage (FF) beantwortet werden:

FF: *Wie können regelbasierte Vorgehensweisen mit maschinellem Lernen kombiniert werden, um bei einem geringen manuellen Annotierungsaufwand Informationen aus unstrukturierten Texten zu extrahieren?*

Zur Beantwortung dieser Frage müssen die folgenden Teilfragen (TF) beleuchtet werden:

- **TF 1:** *Welche Informationen sollen aus welchen Datenquellen extrahiert werden und wie kann das Ergebnis evaluiert werden?*
- **TF 2:** *Welche Verfahren der Informationsextraktion bieten sich hierfür an und wie können sie kombiniert werden?*
- **TF 3:** *Wie kann eine Extraktionspipeline in der Praxis umgesetzt werden?*
- **TF 4:** *Wie verhält sich das Extraktionsergebnis?*

Die Relevanz der Beantwortung dieser Fragen begründet sich in der Annahme, dass sich die Vorteile von regelbasierten Systemen mit denen des maschinellen Lernens vereinen lassen, um dem Mangel an manuell annotierten Textdaten entgegenzutreten. Die gewonnenen strukturierten Daten können im Nachgang weiterverarbeitet werden und bieten damit viele Möglichkeiten zum Erkenntnisgewinn. So können die extrahierten Informationen bspw. dafür genutzt werden, einen Wissensgraphen aufzubauen, der kontinuierlich um weitere Wissenselemente wachsen kann und zielgerichtete Abfragen ermöglicht.

1.3 Aufbau der Arbeit

Die Entwicklung des Prototyps und damit auch die Beantwortung der Forschungsfrage orientieren sich am Wasserfallmodell der Softwareentwicklung. Jeder Schritt soll dabei eine Teilfrage behandeln. Abbildung 1.1 stellt den praktischen und den wissenschaftlichen Ansatz gegenüber.

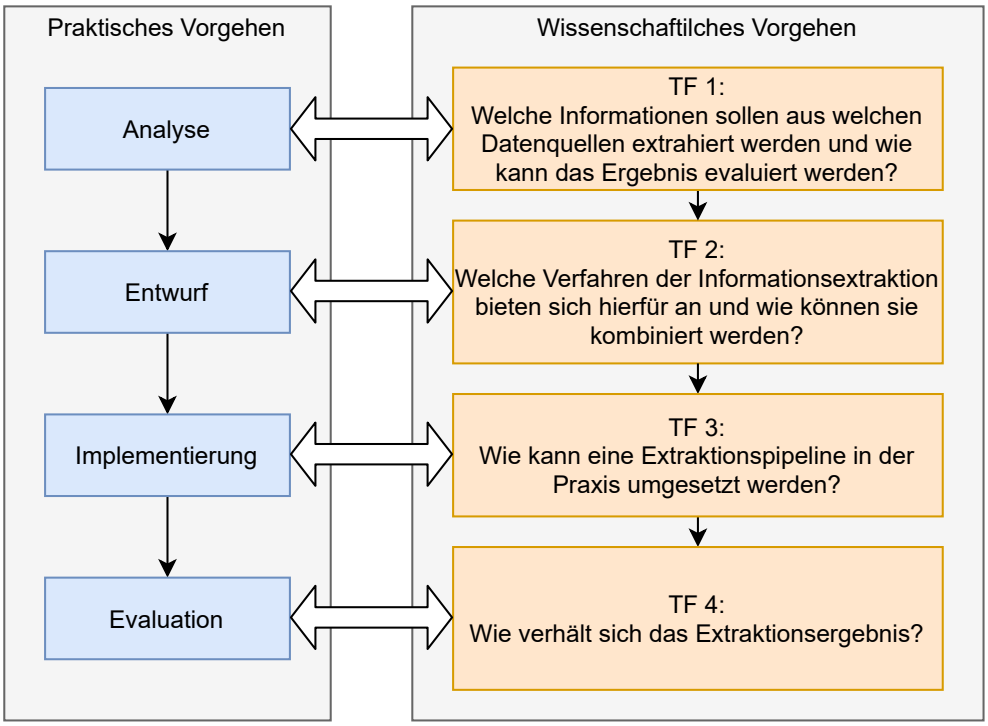

Abbildung 1.1: Methodik

Am Ende jedes Kapitels beantworten wir die jeweilige Teilfrage in Form einer Zusammenfassung.

In der **Analyse** (Kapitel 3) definieren wir zunächst die Anforderungen an die Software. Dies beinhaltet insbesondere die Frage, welche Informationen extrahiert und welche Datenquellen hierfür herangezogen werden sollen. Außerdem stellen wir ein Evaluationsschema für den Prototyp auf, wobei wir gängige Evaluationsmetriken der Informationsextraktion einführen.

Im **Entwurf** (Kapitel 4) beleuchten wir, wie regelbasierte Herangehensweisen zur Informationsextraktion mit Deep Learning kombiniert werden können. Um die ermittelten Verfahren anwenden zu können, müssen wir das Problem entsprechend modellieren. Unter Berücksichtigung der vorhergehenden Überlegungen entwerfen wir die Architektur des Prototyps.

Die **Implementierung** (Kapitel 5) beinhaltet die praktische Umsetzung der zuvor entworfenen Extraktionspipeline und soll insbesondere die Frage klären, welche bestehenden Werkzeuge und Bibliotheken hierbei verwendet werden können. Obwohl sich der Aufbau der Bachelorarbeit am Wasserfallmodell orientiert, ist in der praktischen Umsetzung ein iteratives Vorgehen geplant. Dadurch können mögliche Probleme früher erkannt und die Verzahnung der einzelnen Komponenten besser berücksichtigt werden.

Zur **Evaluation** (Kapitel 6) des Prototyps wenden wir die Extraktionspipeline auf einen Testdatensatz an. Zur Bewertung des Ergebnisses ziehen wir das in Kapitel 3 definierte Evaluationschema heran.

2 Grundlagen

In diesem Kapitel werden die theoretischen Grundlagen dieser Arbeit vermittelt. Zunächst wird eine Übersicht zur maschinellen Sprachverarbeitung gegeben. Anschließend wird das Aufgabengebiet der Informationsextraktion beleuchtet und zuletzt wird auf künstliche neuronale Netze eingegangen.

2.1 Maschinelle Sprachverarbeitung

Maschinelle Sprachverarbeitung, auch bekannt unter der englischen Bezeichnung *Natural Language Processing (NLP)* bezeichnet ein interdisziplinäres Aufgabenfeld, das sich mit dem Verstehen und der Manipulation natürlicher Sprachen durch Computer befasst. Als Teilgebiet der künstlichen Intelligenz agiert es an der Schnittstelle zwischen Neurowissenschaften, Linguistik, Mathematik, Statistik und Informatik (Kedia & Rasu, 2020, S. 11).

Die geschichtliche Entwicklung von NLP als Forschungs- und Andwendungsgebiet lässt sich in drei Phasen aufteilen (Deng & Liu, 2018, S. 2):

In den 1950ern begann die Phase des Rationalismus. Damals wurden Regeln durch Domänen-Experten erstellt, die grammatikalische Strukturen benutzen. Durch die fehlende Fähigkeit der Systeme zu lernen, ergab sich allerdings das Problem, dass deren Anwendungsdomäne sehr eng begrenzt und dadurch keine Übertragung auf Texte anderer Domänen möglich war (Generalisierung).

Ab den 1990ern, im sog. Empirismus, dominierte ein datengetriebener Ansatz in Verbindung mit maschinellen Lernverfahren die Forschung zu NLP. Zur Anwendung kamen statistische Modelle wie *Hidden Markov Models* oder *Suport Vector Machines*. Diese Modelle besitzen Parameter, die durch Lernalgorithmen anhand zuvor im Rahmen des *Feature Engineering* ausgewählter Textmerkmale angepasst werden. Sie eignen sich gut für einfachere Aufgaben, allerdings fehlt ihnen für komplexere Entscheidungen die Tiefe.

Aktuell wird hauptsächlich an *Deep Learning* geforscht, was ebenfalls zum Bereich des maschinellen Lernens gehört. Im Unterschied zu den oberflächlichen

statistischen Modellen werden hier mehrlagige („deep“) neuronale Netze verwendet. Das birgt den Vorteil, dass wichtige Textmerkmale durch das Netz selbst extrahiert und ausgewählt werden und dadurch der kostenintensive Schritt des Feature Engineering wegfällt. Die Lernalgorithmen benötigen dafür jedoch ein Vielfaches an Trainingsdaten.

2.1.1 Linguistische Analyse

Ein essentielles Teilgebiet von NLP ist die linugistische Analyse. Dabei kann zwischen Morphologie und Syntax unterschieden werden. Während bei ersterer einzelne Wörter betrachtet werden, wird bei der syntaktischen Analyse die grammatikalische Struktur untersucht.

Morphologie

Einer der wichtigsten Aufgaben beim NLP ist das **Part-of-speech-Tagging (POS-Tagging)**. Hierbei werden Wörter entsprechend ihrer Wortart mit Tags gekennzeichnet. Da abstraktere Aufgaben, wie bspw. Informationsextraktion von darunterliegenden Informationen profitieren, lassen sich POS-Tagger zum Preprocessing von Texten einsetzen (Mitkov, 2009, S. 220 f.). Die Verfahren zum POS-Tagging lassen sich auch hier in regelbasierte und statistische Lernverfahren unterscheiden. Zu Beginn ihrer Entwicklung funktionierten POS-Tagger noch durch handgeschriebene Regeln, allerdings kamen später auch Systeme zum Einsatz, die manuell annotierte Textkorpora benutzten, um daraus neue Regeln abzuleiten (Martinez, 2012).

Eine weiterer nützlicher Schritt bei der Vorverarbeitung des Textkorpus ist die **Lemmatisierung** von Wörtern. Hierbei werden Wörter in ihre Grundform (Lemma) überführt. Beispielsweise werden Verben in ihre Gegenwartsform und Substantive in ihre Singularform umgewandelt. Das hilft regelbasierten Systemen dabei, verschiedene Varianten von Wörtern zu erkennen (bspw. bei der Klassifikation von Entitäten).

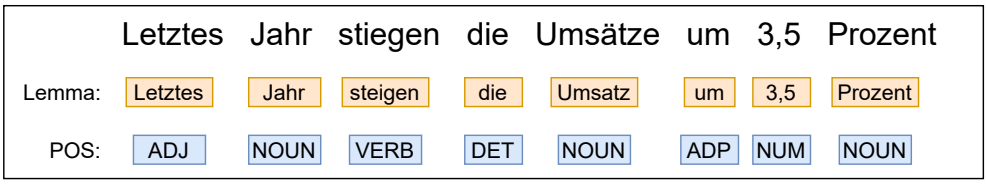

Abbildung 2.1: Beispiel für Part-of-speech-Tags und Lemmata

Abbildung 2.1 zeigt einen Satz, in dem für jedes Wort ein POS-Tag sowie ein Lemma annotiert wurde. Hier wird z.B. deutlich, wie bei der Lemmatisierung das Wort „stiegen“ zu „steigen“ wird. Dadurch können Muster nur mithilfe des

Lemmas alle Vorkommnisse im Text finden, unabhängig davon, in welchem Tempus das Verb steht.

Syntax

Zur Analyse der grammatikalischen Strukturen kommt das **Dependency Parsing** zum Einsatz. Das Ziel besteht dabei darin, einen Satz in eine Baumstruktur zu überführen und dabei jedem Token eine grammatikalische Rolle zuzuordnen.

Die Methoden zum Parsen können in grammatikbasierte und datengetriebene Verfahren aufgeteilt werden. Während sich erstere vom Gebiet der formalen Grammatiken bedienen, verwenden letztere Algorithmen des maschinellen Lernens (Kübler et al., 2009).

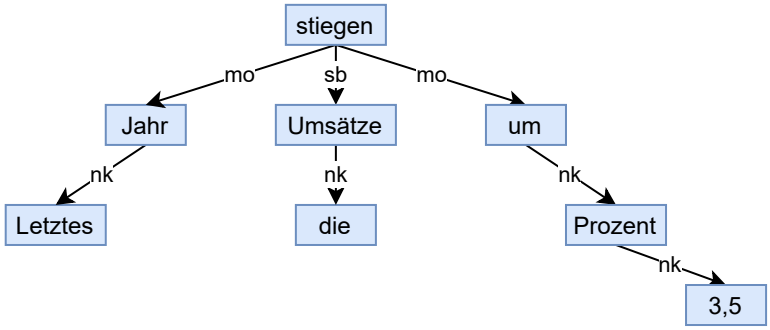

Abbildung 2.2: Beispiel eines Syntaxbaums

Abbildung 2.2 illustriert, wie der Beispielsatz aus dem vorherigen Abschnitt als Abhängigkeitsgraph dargestellt werden kann. An den Pfeilen stehen als Kürzel die grammatikalischen Rollen der jeweiligen Kindelemente innerhalb des Satzes. Diese Kürzel hängen vom verwendeten Parser ab. Im Beispiel wurde die Open Source Programmbibliothek *spaCy*[1] von *Explosion*[2] verwendet. So steht **sb** für *subject*, **mo** für *modifier* und **nk** für *noun kernel element*.

Die Stanford Universität entwickelte eine Syntax zur Mustererkennung namens **Semgrex**[3], mit deren Hilfe Beziehungen zwischen Knoten in einem Abhängigkeitsgraphen gesucht werden können. Seien A und B Knoten in einem Abhängigkeitsbaum und r eine Beziehung, dann werden u.a. die folgenden Operatoren definiert:

A ≫ r B B liegt unterhalb von A und A steht in Beziehung r zu B.

A > r B B ist ein direktes Kindelement von A und A steht in Beziehung r zu B.

[1] https://spacy.io/
[2] https://explosion.ai/
[3] https://nlp.stanford.edu/nlp/javadoc/javanlp/edu/stanford/nlp/semgraph/semgrex/SemgrexPattern.html

Im Beispiel oben würde der Semgrex-Ausdruck {} >*sb* {} „stiegen“ und „Umsätze“ für A bzw. B einsetzen. Diese Suche über dem Syntaxbaum ist vor allem für die regelbasierte Extraktion von Relationen zwischen Enitäten aus Texten interessant (Grishman, 2015).

2.1.2 Reguläre Ausdrücke

Reguläre Ausdrücke sind im praktischen Sinne Textmuster, mit deren Hilfe Texte durchsucht und manipuliert werden können. Sie können verwendet werden, um Textstellen zu finden bzw. durch einen anderen Text zu ersetzen oder um eine Eingabe hinsichtlich eines Musters zu überprüfen. Viele Programmiersprachen wie bspw. Perl, PHP, Java oder Python bieten Parser für reguläre Ausdrücke an, wodurch sich Unterschiede in der Syntax ergeben können (Goyvaerts & Levithan, 2010). Die grundlegenden Elemente eines regulären Ausdrucks sollen im folgenden kurz dargestellt werden (Friedl, 2009):

Alle Zeichenliterale, die keine sonstigen Funktionen haben, werden exakt so gesucht, wie sie angegeben werden. Im Allgemeinen wird zwischen Groß- und Kleinschreibung unterschieden.

Es besteht die Möglichkeit mittels eckigen Klammern Zeichenmengen zu definieren, um z.B. mehrere Schreibweisen eines Wortes abzudecken (*Al[bp]traum*). Eine Menge steht als Platzhalter für genau ein Zeichen und stimmt dann mit einem Text überein, falls das Zeichen an dieser Stelle in der Menge enthalten ist. Für Buchstaben und Ziffern können auch Spannweiten definiert werden (*[0-9][a-z]*). Zudem lässt sich mit Zirkumflex das Komplement der Menge bilden (*[^0-9]* entspricht allen Zeichen außer den Ziffern 0 bis 9).

Um anzugeben, wie oft Zeichen oder Gruppen nacheinander wiederholt werden sollen, dienen die folgenden Quantoren, die hinten angestellt werden:

***** Das Zeichen kommt beliebig oft oder gar nicht vor.

? Das Zeichen kommt höchstens ein mal vor.

\+ Das Zeichen kommt mindestens ein mal vor (a+ entspricht aa*).

Ähnlich wie in der Mathematik können mit Klammern Gruppierungen gebildet werden. Dadurch können bspw. Quantoren auf ganze Zeichenketten angewandt werden. Der Ausdruck *Blumen(topf)?* würde bspw. sowohl mit „Blumen“ als auch mit „Blumentopf“ übereinstimmen. Mit dem Pipe Symbol lässt sich zudem die Vereinigung aus zwei regulären Ausdrücken bilden. So würden durch *(Blumen|Blumentopf)* die selben Textstellen gefunden werden.

Es existieren eine Reihe von vordefinierten Zeichenklassen, die mehrere Zeichen auf einmal abdecken und als Platzhalter anstelle eines Literals verwendet werden können:

. Steht für jedes beliebige Zeichen.

\s Steht für Leer- und Steuerzeichen.

\d Steht für Ziffern.

2.2 Informationsextraktion

Informationsextraktion (IE) beschreibt den Prozess, Entitäten und semantische Beziehungen zwischen diesen aus einem Text zu extrahieren. IE kann aus mehreren Teilschritten aus dem Bereich der maschinellen Sprachverarbeitung bestehen. Dazu gehören u.a. die syntaktische Analyse, Named Entity Recognition und Relationship Extraction (Grishman, 2015).

2.2.1 Named Entity Recognition

Named Entity Recognition (NER) befasst sich mit der Aufgabe *Named Entities* – nachfolgend kurz *Entitäten* genannt – in einem Text zu identifizieren. Hierfür müssen wir zunächst einmal klären was Entitäten sind. Hinsichtlich einer bestimmten Anwendungsdomäne entsprechen Entitäten Teilinformationen, die in der Beschreibung von Fakten oder Ereignissen eine Rolle spielen. Das können einerseits Eigennamen (z.B. für Personen oder Organisationen) sein, aber auch numerische Ausdrücke wie Datums- oder Prozentangaben können als Entitäten betrachtet werden (Nouvel, 2016, S. 3). Komplexere Aufgaben wie Relationship Extraction oder Event Extraction benötigen diese Teilinformationen, weshalb NER bei der Informationsextraktion unerlässlich ist und das Gesamtergebnis maßgeblich davon abhängt, wie gut Entïtäten erkannt werden.

NER kann in zwei Teilprobleme aufgeteilt werden. Einerseits muss erkannt werden, welche Textstellen eine Entität repräsentieren. Wenn man einen Text als eine Sequenz aus Wörtern betrachtet, können bspw. ein Start- und ein Endindex zur Definition einer bestimmten Textspanne benutzt werden. Zum anderen muss der gefundene Bereich klassifiziert, also einem Entitätstypen zugeordnet werden. Hierfür gibt es die folgenden Indikatoren (Nouvel, 2016, S. 78 ff.):

- **Morphologie**
  Die Struktur eines Wortes kann Aufschluss darüber geben, um welchen Entitätstypen es sich handelt. Bestimmte Pre- oder Suffixe sowie Groß- und Kleinschreibung können dabei ebenso nützlich sein wie POS-Tags.
- **Lexikalische Datenbanken**
  Eine weitere Möglichkeit ist die Suche von Entitäten in eigens dafür erstellten Datenbanken, die so vollständig wie möglich sein sollten. Algorithmen die hierfür verwendet werden haben einen moderaten Rechenaufwand. Al-

lerdings ist die Pflege der Datenbanken zu Entiäten mit all ihren Variationen aufwändig.

- **Hinweise im Kontext**
  Es ist oft nicht ausreichend, ausschließlich einzelne Wörter zu betrachten. Falls die beiden vorherigen Indikatoren nicht verfügbar sind oder zu keiner eindeutigen Klassifizierung führen, kann es nützlich sein, den Kontext miteinzubeziehen. Dafür kann beispielsweise das Ergebnis der syntaktischen Analyse benutzt werden.

2.2.2 Relationship Extraction

Relationship Extraction (RE) versucht auf den Ergebnissen von NER aufbauend, semantische Beziehungen zwischen den gefundenen Entitäten zu extrahieren. Da fast alle Informationen Beziehungen darstellen, ist RE für die Informationsextraktion von entscheidender Bedeutung.

Eine Beziehung kann als ein 3-Tupel $\langle E_1, R, E_2 \rangle$ angesehen werden, wobei E_1 und E_2 zwei unterschiedliche Entitäten sind und R einen Beziehungstypen beschreibt. Diese Darstellung eignet sich besonders zur Speicherung in Graphdatenbanken bei der Entitäten als Knoten und Beziehungen als Kanten zwischen diesen interpretiert werden.

Bei **Event Extraction (EE)** handelt es sich um eine komplexe Form von RE (Buyko et al., 2011). Das *Linguistic Data Consortium* definiert ein Ereignis als das Auftreten einer Zustandsänderung mit mehreren Beteiligten (Linguistic Data Consortium, 2005). Das Aufgabenfeld lässt sich dahingehend unterscheiden, ob Ereignisse in einer geschlossenen oder offenen Domäne extrahiert werden sollen. Während bei Letzterem versucht wird, vorher unbekannte Ereignisse in Texten zu finden und nach Ähnlichkeit zu clustern, werden bei der *closed-domain* EE im Vorfeld Schemata definiert, die die Strukturen der Ereignisse vorgeben.

Ereignisse sind unter anderem durch die folgenden Komponenten gekennzeichnet (Xiang & Wang, 2019):

- **Auslöser (Event-Trigger):** Wort (z.B. Verb oder Nomen), das signalisiert, dass es sich um ein Ereignis handelt.
- **Argumente (Event-Argument):** Entitäten, die im Ereignis als Argument auftreten.
- **Rollen (Argument-Role):** Rolle, die ein Argument in einem Ereignis einnimmt.

Um Ereignisse zu extrahieren muss folglich zunächst ein Trigger erkannt und darauffolgend die zugehörigen Argumente mit ihren jeweiligen Rollen identifiziert werden.

2.3 Künstliche neuronale Netze

Künstliche neuronale Netze (KNNs) sind Modelle zur Informationsverarbeitung, die von den biologischen neuronalen Verbindungen im menschlichen Gehirn inspiriert sind. Im Bereich der künstlichen Intelligenz werden KNNs genutzt, um komplexe Probleme zu lösen.

Ein mehrschichtiges Feedforward-Netz ist ein in Schichten aufgebautes Netz aus künstlichen Neuronen. Jedes Neuron ist ausschließlich mit Neuronen der nächsten Schicht verbunden, d.h. es existiert keine Rückkopplung.

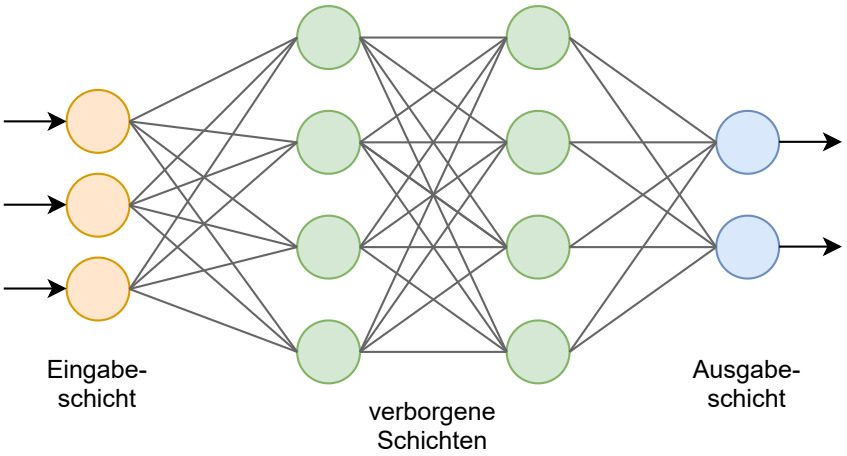

Abbildung 2.3: Vereinfachte Darstellung eines feedforward-Netzes

Im Falle einer Klassifikation hat die Ausgabeschicht so viele Neuronen wie es Klassen gibt.

Die Berechnungseinheiten, aus denen ein KNN aufgebaut ist, nennt man analog zum biologischen Vorbild (künstliche) Neuronen.

Jedes Neuron hat als Eingabe einen Vektor $x \in \mathbb{R}^n$ und berechntet eine Ausgabe $y \in \mathbb{R}$. Alle Eingabekanten des Neurons haben jeweils eine Gewichtung w_i die in der Übertragungsfunktion mit dem zugehörigen Eingabewert x_i multipliziert wird. Mit anschließender Addition des Biases b ergibt das die sog. Netzeingabe $v = (\sum_{i=1}^{n} x_i w_i) + b$. Die Netzeingabe wird in eine Aktivierungsfunktion φ gegeben, welche schlussendlich die Ausgabe des Neurons berechnet. Die Gewichtungen und der Bias sind Eigenschaften eines einzelnen Neurons und werden im Rahmen des Lernprozesses angepasst.

Grundsätzlich kann als Aktivierungsfunktion jede differenzierbare Funktion verwendet werden. Allerdings eigenen sich einige für bestimmte Probleme mehr als andere. Zur Klassifikation mit mehr als zwei Klassen eignet sich insbesondere die

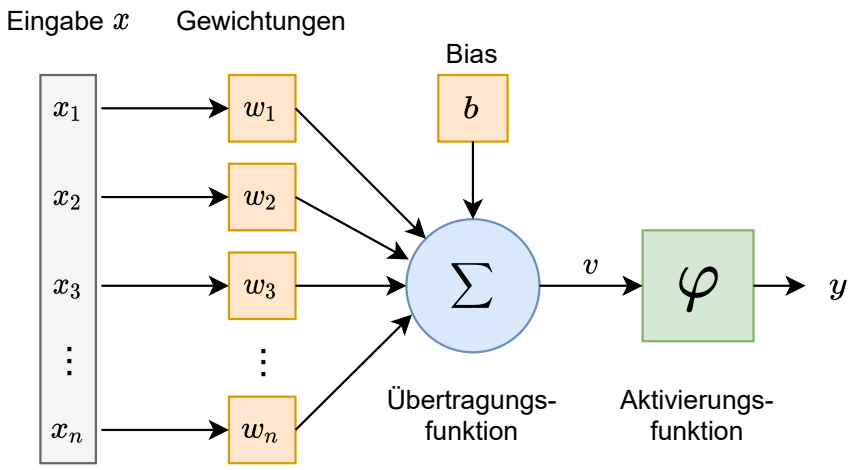

Abbildung 2.4: Künstliches Neuron

Softmax-Funktion, die folgendermaßen definiert ist (Sharma et al., 2020):

$$\sigma(z)_j = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}} \quad \text{für j} = 1, ..., \text{K}$$

K ist hierbei die Anzahl der zur Verfügung stehenden Klassen. In einem neuronalen Netz, das zur Klassifikation eingesetzt wird, entspricht diese Zahl typischerweise der Anzahl an Neuronen in der Ausgabeschicht. Die Funktion sorgt dafür, dass die Ausgabe jedes Neurons einen Wert zwischen 0 und 1 annimmt und die Ausgaben aller Neuronen summiert 1 ergeben. Dadurch ergibt sich eine Wahrscheinlichkeitsverteilung über alle Klassen.

Eine weitere weit verbreitete Aktivierungsfunktion, die häufig in verborgenen Schichten des neuronalen Netzes Anwendung findet, ist die **ReLU-Funktion**. Sie ist definiert als $f(x) = max(0, x)$ und hat den Vorteil das sie effizient berechnet werden kann.

3 Analyse

In diesem Kapitel wird zunächst eine kurze Übersicht der genutzten Datenquellen präsentiert. Im Anschluss werden die Anforderungen an eine prototypische Extraktionspipeline definiert. Zuletzt wird ein Bewertungsmodell konzipiert, welches zur Evaluation herangezogen wird.

3.1 Datenquellen

Als Datenquelle für die Informationsextraktion dienen Branchenberichte die den Kunden von DATEV durch den Dienst LEXinform über verschiedene Finanzinstitute zur Verfügung gestellt werden. Im Rahmen dieser Bachelorarbeit werden 66 Berichte des Deutschen Sparkassen- und Giroverbands e. V. und 129 des Rheinisch-Westfälischen Genossenschaftsverbands verwendet. Diese Berichte liegen alle im PDF-Format vor, was zusätzliche Herausforderungen bei der Extraktion mit sich bringt. Zusätzlich zu den eigentlichen Textdaten existiert für jeden Bericht eine XML-Datei mit Metadaten. Darin enthalten sind Angaben zum Herausgeber, zum Erscheinungsdatum sowie zu den behandelten Branchen.

3.2 Anforderungen

Die Verwendung des Prototyps kann in zwei Phasen aufgeteilt werden. Die erste Phase besteht aus dem Trainieren des neuronalen Klassifizierers mit programmatisch annotierten Testdaten. In der zweiten Phase soll der Prototyp genutzt werden um aus unbekannten Dokumenten Informationen zu extrahieren.

A1: Extraktion von Branchenentwicklungen

Der Prototyp hat die Hauptaufgabe Informationen über Kennzahlen einer Branche und deren zeitliche Entwicklung aus Branchenberichten zu extrahieren. Konkret soll für jede Erwähnung einer Steigerung oder Minderung die zugehörige Kennzahl, der Zeitpunkt (bzw. Zeitraum) und der Wert der Änderung extrahiert werden.

A2: Regelbasierte Annotierung von Trainingsdaten

Im Rahmen dieser Bachelorarbeit soll Informationsextraktion unter der Prämisse mangelnder vorannotierter Datensätze sowie fehlender personeller Ressourcen zur manuellen Annotation untersucht werden. Deshalb soll der Prototyp die Informationen im Text mithilfe einer regelbasierten Herangehensweise selbst annotieren.

A3: Klassifikation durch künstliche neuronale Netze

Die programmatisch annotierten Daten sollen benutzt werden, um ein künstliches neuronales Netz zu trainieren. Dieses Modell soll als Klassifikator dienen und jeden Textdatensatz in eine von mehreren noch zu definierenden Kategorien (z.B. Steigerung, Stagnation, Minderung) einordnen.

A4: Extraktion aus PDF-Dokumenten

Da die zu untersuchenden Branchenberichte als PDF-Dokumente vorliegen, muss der Prototyp in der Lage sein, Text aus diesen zu lesen. Neben Fließtext enthalten PDF-Dokumente weitere Artefakte wie bspw. Grafiken und Tabellen. Da diese mittels NLP nicht effizient weiterverarbeitet werden können, sollen sie von vornherein entfernt werden.

A5: Berücksichtigung von Metadaten

Zur künftigen Verwertung der extrahierten Informationen müssen diesen jeweils die folgenden Metadaten zugeordnet werden:

- **WZ-Code**
  In Deutschland wird jedem Wirtschaftszweig ein sog. WZ-Code zugeordnet, welcher vom Statistischen Bundesamt definiert wird. Dieser wird benötigt, um den Kennzahlenentwicklungen eine konkrete Branche zuordnen zu können
- **Herausgeber**
  Angaben zur Quelle sind wichtig, um die extrahierten Daten zurückverfolgen zu können.
- **Erscheinungsdatum**
  Das Erscheinungsdatum des Branchenberichts ist zur Auflösung von zeitlichen Angaben essentiell.

A6: Strukturierte Speicherung

Die Informationen sollen im strukturiert JSON-Format gespeichert werden, sodass sie leicht wieder gelesen und zugeordnet werden können.

3.3 Evaluationsschema

3.3.1 Klassifikation

Zur Evaluation der Klassifikation durch ein neuronales Netz sollen die in der Informationsextraktion gängigen Evaluationsmetriken angewandt werden, die in diesem Abschnitt dargestellt werden.

Konfusionsmatrix

Grundsätzlich wird bei einer Klassifikation jedem Element (Datensatz) eine von n Klassen aus einer Menge $K = \{k_i \mid 1 \leq i \leq n\}$ zugewiesen. Elemente und Klassen sind im Allegmeinen beliebig. Im Kontext dieser Arbeit beschreiben Klassen verschiedene Kategorien von Branchenentwicklungen.

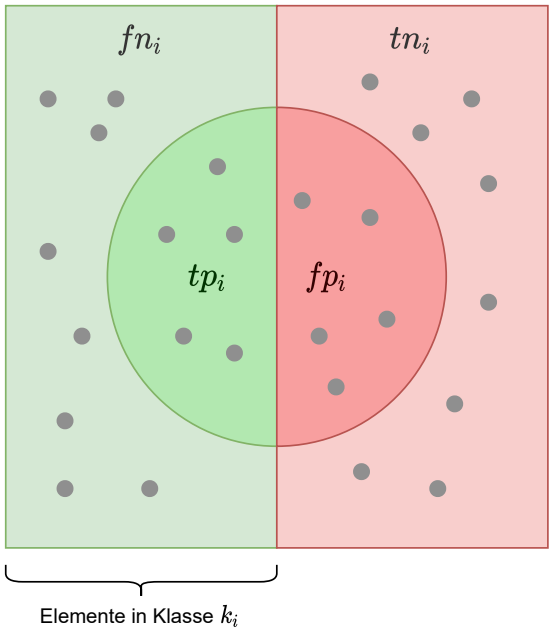

Abbildung 3.1: Bildliche Darstellung einer Konfusionsmatrix

Abbildung 3.1 stellt eine Konfusionsmatrix bildlich dar. Innerhalb des Kreises befinden sich alle Elemente, die der Klassifkator als zur Klasse k_i zugehörig gekennzeichnet hat (positive). Um das Schaubild und die Formeln zu verstehen, müssen zunächst einige Variablen eingeführt werden:

tp_i**:** Anzahl der Elemente **innerhalb** der Klasse k_i, die vom Klassifikator als positiv gekennzeichnet wurden (true positives).

fp_i**:** Anzahl der Elemente **außerhalb** der Klasse k_i, die vom Klassifikator als positiv gekennzeichnet wurden (false positives).

tn_i: Anzahl der Elemente **außerhalb** der Klasse k_i, die vom Klassifikator als negativ gekennzeichnet wurden (true negatives).

fn_i: Anzahl der Elemente **innerhalb** der Klasse k_i, die vom Klassifikator als negativ gekennzeichnet wurden (false negatives).

Evaluationsmetriken

Aus der Konfusionsmatrix leiten sich die nachfolgenden, häufig in der Informationsextraktion eingesetzten Evaluationsmetriken ab. Um das Gesamtsystem (alle Klassen) zu bewerten, werden die Metriken nach dem *micro-average*-Verfahren berechnet. Dabei werden *tp*, *fp*, *tn* und *fn* kumulativ über alle Klassen summiert und anschließend in die Metrik eingesetzt (Sokolova & Lapalme, 2009):

Recall entspricht dem Anteil der Elemente einer Klasse, die vom Klassifikator korrekt gekennzeichnet wurden. Dadurch wird bewertet, wie gut ein Klassifikator die Elemente einer Klasse erkennt:

$$Recall = \frac{tp}{tp + fn} \qquad Recall_\mu = \frac{\sum_{i=1}^{n} tp_i}{\sum_{i=1}^{n}(tp_i + fn_i)}$$

Precision beschreibt, welcher Anteil der als positiv gekennzeichneten Elemente auch tatsächlich zu dieser Klasse gehört. Damit wird evaluiert, wie präzise die Vorhersagen des Klassifikators sind:

$$Precision = \frac{tp}{tp + fp} \qquad Precision_\mu = \frac{\sum_{i=1}^{n} tp_i}{\sum_{i=1}^{n}(tp_i + fp_i)}$$

F_1-Score entspricht dem harmonischen Mittel von Precision und Recall, wobei beide gleich gewichtet werden. Je höher dieser Wert ist, desto besser ist die Gesamtperformance des Extraktionssystems, da sowohl Recall als auch Precision maximiert werden sollten:

$$F_1 = \frac{2 \cdot Precision \cdot Recall}{Precision + Recall} \qquad F_{1\mu} = \frac{2 \cdot Precision_\mu \cdot Recall_\mu}{Precision_\mu + Recall_\mu}$$

Zur Evaluation der Klassifikation von Branchenentwicklungen werden 10% der Daten als Testdatensatz ausgewählt. Anschliesënd werden die Evaluationsmetriken für den neuronalen Klassifkator bzgl. dieser Testdaten berechnet.

3.3.2 Anforderungen

Die Anforderungen sollen nach einem simplen Schema bewertet werden. Dabei stehen die folgenden Bewertungen zur Auswahl:

erfüllt:	Die Anforderung wurde zu 100 % erfüllt.
teilweise erfüllt:	Teile der Anforderung wurden erfüllt.
nicht erfüllt:	Die Anforderung wurde nicht erfüllt.

4 Entwurf

In diesem Kaptitel soll die Frage beantwortet werden, wie ein regelbasiertes Vorgehen mit maschinellem Lernen bei der Informationsextraktion kombiniert werden kann. Vor diesem Hintergrund wird am Beispiel von Brancheninformationen eine Pipeline entworfen, die beide Herangehensweisen vereint.

4.1 Modellierung als Klassifikationsproblem

Bevor über Kombinationsmöglichkeiten von Verfahren nachgedacht werden kann, müssen wir das Extraktionsproblem formal modellieren.

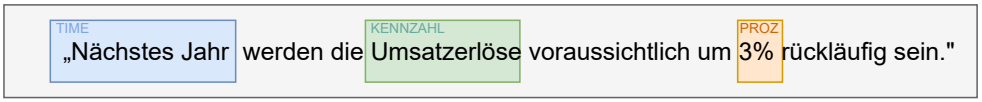

Abbildung 4.1: Beispielhafte Schilderung einer Branchenentwicklung

Wie der beispielhafte Satz in Abbildung 4.1 erkennen lässt, handelt es sich bei Branchenentwicklungen um komplexe Beziehungen, an denen mehrere Entitäten (farbig markiert) beteiligt sind. So werden Informationen zum Zeitpunkt, zur Kennzahl, sowie zum Prozentwert der Steigerung vermittelt. Dieser Sachverhalt wird sehr gut durch den Aufgabenbereich der **Event Extraction (EE)** abgebildet. Da wir bereits wissen welche Informationen wir extrahieren möchten, bewegen wir uns in einer geschlossenen Domäne und müssen demnach Branchenentwicklungen als Ereignis modellieren.

Entitäten

Zur Beschreibung von Ereignissen müssen wir zunächst unterschiedliche Typen von Entitäten definieren, die als Argumente fungieren. Ziel ist die Extraktion von Informationen zur zeitlichen Entwicklung von Kennzahlen. D.h. wir benötigen den Zeitpunkt der Entwicklung, die betreffende Kennzahl sowie den Wert der Entwicklung.

Typ	Kürzel	Beschreibung
Kennzahl	KNZ	Betriebswirtschaftliche Kennzahl
Prozentwert	VAL_PER	Zur prozentualen Angabe von Steigerungen und Minderungen
Absolutwert	VAL_ABS	Absoluter Wert in Euro
Zeitraum	TIME	Beschreibt einen Zeitraum unterschiedlicher Granularität

Tabelle 4.1: Entitätstypen zu Branchenentwicklungen

Tabelle 4.1 zeigt alle Entitätstypen, die zur Abbildung von Branchenentwicklungen notwendig sind. Das angegebene Kürzel wird verwendet, um Wörter bzw. Wortgruppen entsprechend zu annotieren und dient zusätzlich als eindeutige ID für den Entitätstyp.

Ereignisse

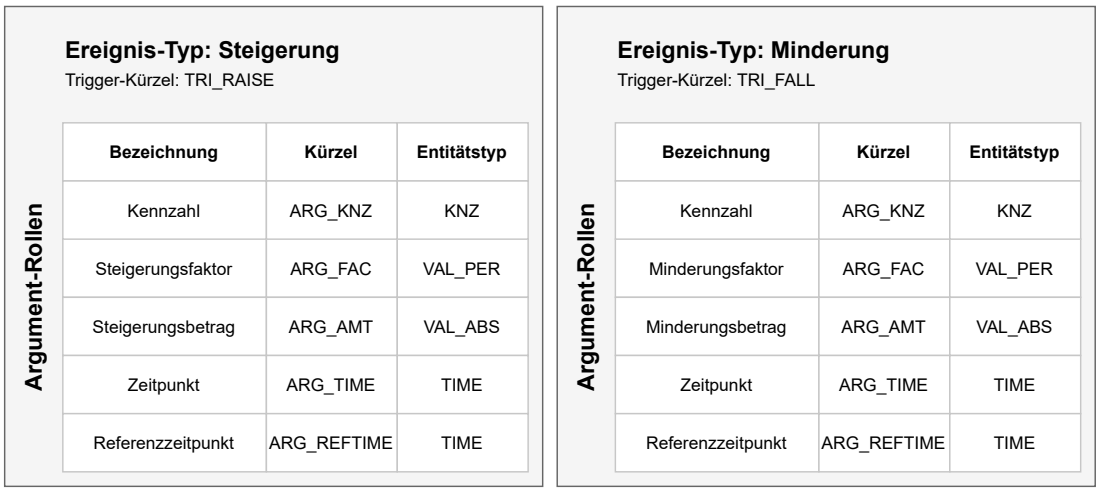

Abbildung 4.2: Ereignisschemata für Steigerung und Minderung

Als nächstes verwenden wir die definierten Entitätstypen, um Schemata für Ereignisse zu entwerfen. Bei den Ereignissen, die uns interessieren, handelt es sich um Steigerungen und Minderungen von Kennzahlen. Diese werden einerseits einem bestimmten Zeitpunkt zugeordnet, können allerdings auch mit Phrasen wie „gegenüber dem Vorjahr“ explizit in eine zeitliche Relation gesetzt werden. Wertänderungen können sowohl prozentual als auch in absoluten Zahlen ausgedrückt werden.

4.2 Architektur

In diesem Kapitel widmen wir uns dem konzeptionellen Aufbau des Prototyps. Wir wollen eine Extraktionspipeline entwerfen, die regelbasierte Vorgehensweisen mit maschinellem Lernen vereint, um manuellen Annotierungsaufwand zu verhindern. Daher bietet es sich an, Textdaten automatisiert durch Regeln zu annotieren und damit ein neuronales Netz zu trainieren, das gefundene Informationen klassifiziert.

Bei der Architektur muss grundsätzlich zwischen drei Anwendungsfällen unterschieden werden. Zunächst müssen die Trainingsdaten aus den PDF-Dokumenten generiert werden. Anschließend wollen wir einen neuronalen Klassifizierer entwerfen und mit den erstellten Daten trainieren. Zuletzt soll das trainierte Modell in Kombination mit Regeln in einer Pipeline auf neue Daten angewandt werden.

4.2.1 Erstellung der Trainingsdaten

In der ersten Phase werden die Rohdaten verarbeitet und mittels Regeln annotiert und so für die Weiterverarbeitung vorbereitet.

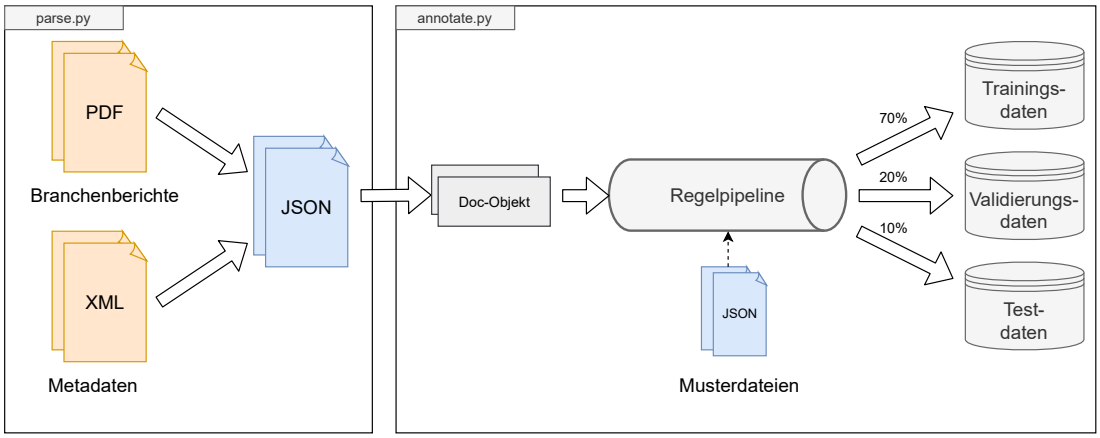

Abbildung 4.3: Erstellung der Trainingsdaten

Abbildung 4.3 zeigt den Fluss der Rohdaten beim Erstellen der Trainingsdaten. Als Erstes müssen wir die vorhandenen PDF-Dokumente auslesen und deren Text extrahieren. Zu jeder PDF-Datei existiert ein XML-Dokument, dass Zusatzinformationen wie bspw. das Herausgabedatum oder den WZ-Code der behandelten Branchen enthält. Diese Metadaten werden zusammen mit dem extrahierten Rohtext in eine einzelne JSON-Datei pro Branchenbericht geschrieben.

Im Anschluss wird die Regelpipeline initialisiert und mit den Daten gespeist. Dort wird jeder Text in mehrere Doc-Objekte[1] umgewandelt, in denen wir die

[1] https://spacy.io/api/doc

linguistischen und regelbasierten Annotationen sowie die Metadaten speichern. Da eine Branchenentwicklung meist innerhalb eines Satzes erwähnt wird und um die Aufgabe zu vereinfachen, erstellen wir pro Satz genau ein Doc-Objekt. Davor werden Sätze dahingehend überprüft, ob sie für NLP geeignet sind, denn bei der Textextraktion aus einem PDF-Dokument werden auch Tabellen oder Legenden mitgelesen, deren Text meist kein Verb enthält. Da wir für unsere Zwecke nur an grammatikalisch vollständigen Sätzen interessiert sind, werden unvollständige Sätze von der weiteren Verarbeitung ausgeschlossen.

Sobald alle Dokumente verarbeitet wurden, teilen wir unsere Daten in drei Datensätze auf. 70% der Daten dienen als Trainingsdaten für den Klassifizierer. Weitere 20% werden während des Trainings zur Validierung und Berechnung von Metriken genutzt. Um den trainierten Klassifizierer zu evaluieren, verwenden wir die restlichen 10% der Doc-Objecte als Testdaten.

Die Regelpipeline ist dafür zuständig, die rohen Textdaten zu verarbeiten und mittels Regeln zu annotieren. Hierbei kommen sowohl von spaCy bereitgestellte als auch eigene Komponenten zum Einsatz. Der Pipeline-Aufbau ergibt sich daraus, dass die einzelnen Komponenten auf die Annotationen der vorherigen Schritte zugreifen müssen.

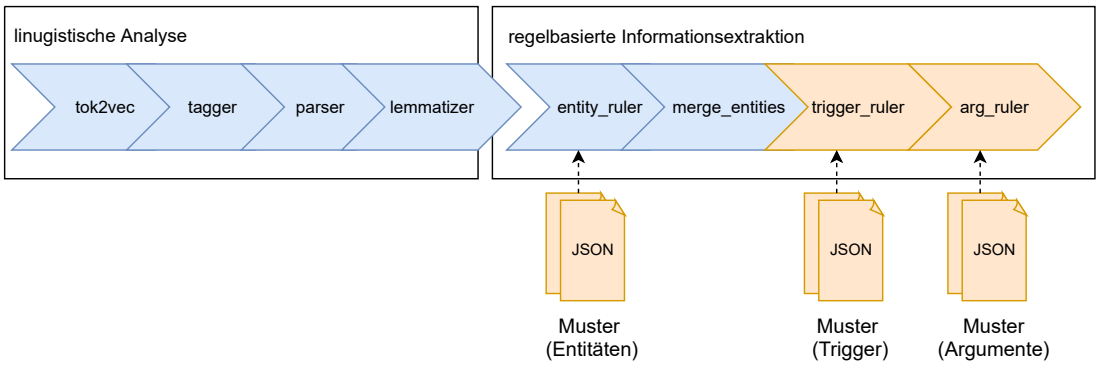

Abbildung 4.4: Regelpipeline

Abbildung 4.4 zeigt die einzelnen Komponenten der Pipeline, wobei eigene Implementierungen orange gekennzeichnet sind. Die Schritte können aufgeteilt werden in die linguistische Analyse und die regelbasierte Informationsextraktion.

Linguistische Analyse

Der erste Teil der Regelpipeline wird durch die von spaCy zur Verfügung gestellte vortrainierte Pipeline **de_core_news_md**[2] realisiert, die zur lexikalischen und syntaktischen Analyse verwendet wird. Als Trainingskorpus wurde von *Ex-*

[2]https://github.com/explosion/spacy-models/releases/tag/de_core_news_md-3.1.0

plosion[3] der TIGER Corpus[4] benutzt. Die dabei erstellten Annotationen werden anschließend in den eigenen Komponenten verwendet, um Regeln zu definieren. Die folgenden Schritte werden bei der linguistischen Analyse durchlaufen:

In der Schicht *tok2vec* werden die Tokens im Text in eine vektorielle Darstellung umgewandelt (sog. Worteinbettungen). Hierzu wird eine neuronales Modell verwendet, das zuvor unüberwacht trainiert wurde.

Der *tagger* ordnet den zuvor generierten Tokens Part-Of-Speech Tags zu.

Im *parser* wird die syntaktische Analyse durchgeführt. Dabei entsteht ein Syntaxbaum, in dem grammatikalische Strukturen abgebildet werden. Jedem Token wird dabei seine Rolle innerhalb des Baums (z.B Subjekt, Objekt etc.) zugeordnet.

Der *lemmatizer* generiert für jedes Token ein Lemma (Wortstamm).

Regelbasierte Informationsextraktion

Der zweite Teil der Pipeline ist für die regelbasierte Extraktion von Brancheninformationen verantwortlich. Wie in Kapitel 4.1 beschrieben, müssen wir Entitäten und Ereignisse, in deren Kontext sie auftreten erkennen.

Die Komponente *entity_ruler* markiert auf Basis von Mustern und Annotationen der linguistischen Analyse gefundene Entitäten. Obwohl für NER heutzutage häufig statistische Verfahren oder neuronale Modelle verwendet werden, ist in unserem Fall ein regebasierter Ansatz vorzuziehen, weil numerische Angaben wie Prozent- oder Zeitangaben mit Mustervergleichen leicht gefunden werden können. Auch die Bezeichnung von Kennzahlen folgt oft einem bestimmten Schema: Die Endungen -quote, -kosten oder -aufwand sind hierbei eindeutige Indikatoren.

Im nächsten Teilschritt *merge_entities* werden Entitäten, die aus mehreren Tokens bestehen zu einem einzigen Token zusammengefasst. Das hilft bei der Extraktion von Ereignissen, da die Argumente jeweils genau einem Token entsprechen und so der tokenbasierte Mustervergleich funktioniert.

Als nächstes wollen wir Ereignisse aus dem Text extrahieren. Hierfür ist es notwendig, zunächst die Trigger und anschließend die Argumente zu identifizieren.

Im *trigger_ruler* werden die Ereignis-Trigger unter Zuhilfenahme der Ergebnisse der syntaktischen Analyse erkannt. Diese Trigger signalisieren, dass eine Branchenentwicklung (Steigerung oder Minderung einer Kennzahl) beschrieben wird. Das können Verben wie „steigen“ oder Substantive wie „Zuwachs“ sein.

[3] https://explosion.ai/

[4] https://www.ims.uni-stuttgart.de/forschung/ressourcen/korpora/tiger/

Zu jedem gefundenen Trigger existieren mehrere Argumente. In der Komponente *arg_ruler* werden diese ebenfalls über Mustervergleiche gefunden und annotiert. Diese Aufgabe ist weitaus komplexer als die Erkennung von Entitäten oder Triggern, weil Sätze beliebig verschachtelt aufgebaut sein können und Argumente in beliebiger Reihenfolge vorkommen können. Aus diesem Grund wird mit den Annotationen dieses Schrittes ein neuronales Modell trainiert.

4.2.2 Trainieren des Klassifizierers

Der Ablauf der Extraktion von Ereignissen durch neuronale Netze lässt sich im Allgemeinen folgendermaßen darstellen: Zunächst erhält ein Ereignis-Klassifizierer als Eingabe ein Wort zusammen mit dem Satz, in dem es vorkommt, und entscheidet, ob dieses Wort einen Ereignis-Trigger repräsentiert. Falls das der Fall ist, wird ein Argument-Klassifizierer auf den gefundenen Trigger und allen Entitäten im Satz angewandt um die Argumente des Ereignisses zu bestimmen (Grishman, 2015).

Für das Finden eines Triggers ist der regelbasierte Ansatz in unserem Fall völlig ausreichend, da die Menge an Wörtern mit denen eine Steigerung oder Minderung ausgedrückt werden kann, begrenzt ist. Bei der Klassifikation von Argumenten kann die Fähigkeit von neuronalen Netzen zur Generalisierung jedoch von Vorteil sein, weil Sätze beliebig strukturiert sein können und die Reihenfolge der Argumente variieren kann. Aus diesem Grund entwerfen wir einen neuronalen Argument-Klassifizierer, der mit den annotierten Datensätzen aus dem vorherigen Schritt trainiert wird.

Trainingspipeline

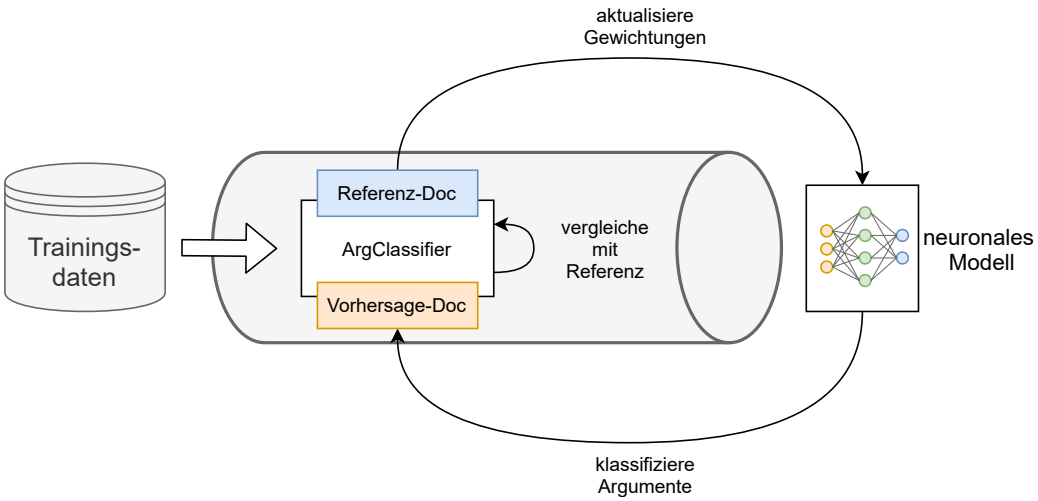

Abbildung 4.5: Trainingsschleife

Die Trainingspipeline besteht aus einer einzigen trainierbaren Komponente, dem Argument-Klassifizierer. Der Klassfizierer ist dafür zuständig, mit dem dahinter liegenden neuronalen Modell sowohl beim Training als auch bei der tatsächlichen Anwendung auf neue Daten zu kommunizieren. In Abbildung 4.5 ist der Trainingsprozess dargestellt. Zu Beginn jeder Trainingsiteration werden Referenzdatensätze aus dem zuvor annotierten Korpus gelesen (Referenz-Doc). Von jedem Datensatz wird anschließend eine Kopie erstellt, in welcher der Klassifikator seine Vorhersagen annotieren soll (Vorhersage-Doc). Dabei werden die Tokenisierung, die Entitäten und die Ereignis-Trigger aus der Referenz übernommen. Einzig die Ereignis-Argumente bleiben in der Kopie leer.

Nachdem das neuronale Modell initialisiert wurde, beginnt die Trainingsschleife mit der Vorhersage der Argumente zu den Ereignis-Triggern im Vorhersage-Doc. Die Vorhersagen werden mit der Referenz verglichen, wobei eine Verlustfunktion den wertmäßigen Unterschied berechnet. Daraufhin werden die Parameter des neuronalen Netzes (Gewichtungen und Bias) so verändert, dass der Verlust verringert wird und die Schleife beginnt von neuem.

4.2.3 Extraktion von Brancheninformationen

Nachdem der neuronale Klassifizierer trainiert wurde, können wir ihn zur Extraktion von Brancheninformationen aus neuen Dokumenten benutzen. Hierfür entwerfen wir eine Pipeline, welche die Mustervergleiche aus der Regelpipeline mit dem Klassifizierer vereint.

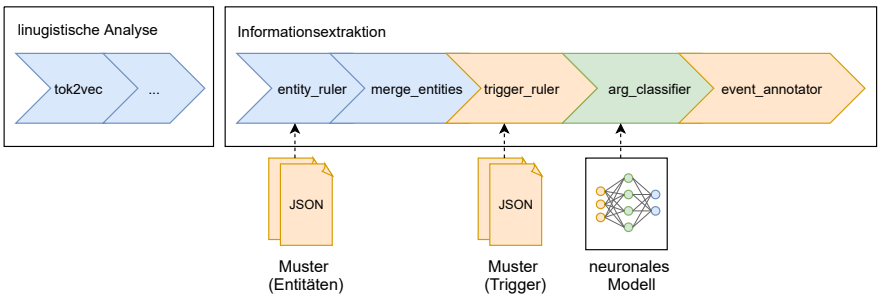

Abbildung 4.6: Vorhersagepipeline

Abblidung 4.6 stellt den Aufbau der Vorhersagepipeline dar. Analog zur Regelpipeline wird auch hier zunächst die linguistische Analyse durchlaufen. Das ist notwendig, damit die regelbasierte Erkennung von Triggern weiterhin funktioniert. Statt des *arg_rulers* aus der Regelpipeline setzen wir hier allerdings den trainierten Klassifizierer *arg_classifier* ein. Zuletzt werden im *event_annotator* unter Berücksichtigung von Vorhersagen der vorherigen Komponenten Ereignisse strukturiert am Doc-Objekt gespeichert.

5 Implementierung

In diesem Kapitel soll dargelegt werden, wie eine Extraktionspipeline in der Praxis implementiert werden kann. Zur Umsetzung wurde die Programmiersprache Python in Verbindung mit der Open Source Programmbibliothek *spaCy*[1].

5.1 Extraktion der Rohdaten

5.1.1 Extraktion aus PDF-Dokumenten

Bevor NLP-Techniken zur Informationsextraktion eingesetzt werden können, muss der Text aus den PDF-Dokumenten extrahiert werden. Dabei handelt es sich um eine komplexe Aufgabe, was daran liegt, dass es sich beim PDF-Format um ein Layoutbeschreibung handelt. Unterschiedliche PDF-Dokumente können unterschiedlich kodiert sein, was dazu führen kann, dass der Text bei der Extraktion die ursprüngliche Struktur verliert (Vajjala, 2020, S. 47).

Es existieren viele Python-Bibliotheken zur Textextraktion aus PDF-Dokumenten, von denen drei getestet wurden, allerdings konnte keine den Text aus den Branchenberichten fehlerfrei extrahieren. Trotzdem ließen sich Unterschiede hinsichtlich der Qualität beobachten.

Auffällig ist weiterhin, dass sich das Ergebnis bei Branchenberichten unterschiedlicher Herausgeber teils massiv unterscheidet. Das liegt einerseits an Differenzen im Layout, andererseits spielen auch die Tools zur Erzeugung der PDFs und somit die verschiedenen Kodierungen eine Rolle.

Pdfminer.six[2] ist eine komplett in Python geschriebene Bibliothek zur Extraktion von Text aus PDF-Dokumenten. Sie liefert annehmbare Ergebnisse und bietet zeitgleich den Vorteil, dass keine zusätzliche Software installiert werden muss. Aus diesem Grund verwenden wir sie zur Extraktion der Rohdaten.

[1] https://spacy.io/
[2] https://github.com/pdfminer/pdfminer.six

5.1.2 Bereinigung des Textes

Wegen der genannten Probleme ist es notwendig, den extrahierten Text zu bereinigen, sodass die Weiterverarbeitung mittels NLP problemlos funktioniert.

Upadhyay und Fujii (2016) stießen in ihrer Forschung auf ähnliche Fehler bei der Textextraktion durch Pdfminer, die sie durch manuelle Korrektur behoben. Da unser Prototyp allerdings komplett auf manuellen Mehraufwand verzichten soll, muss der Text automatisiert bereinigt werden.

Konkret wurden dafür die folgenden Schritte unternommen:

1. Löschen der Silbentrennungszeichen und Zusammenführung der einzelnen Silben
2. Zusammenfassen von mehrerer Leerzeichen zu einem
3. Löschen der Kopf- und Fußzeilen, die den Textfluss unterbrechen
4. Löschen von Aufzählungszeichen
5. Löschen aller Zeilenumbrüche, die durch die Formatierung im PDF-Dokument entstanden waren

```
1  def clean_text(text: str) -> str:
2      '''cleans the text to prepare it for NLP'''
3      text = re.sub(r"-\s*\n(?=[a-zäöüß])", "", text) #remerge
       ↪  words
4      text = re.sub(r"\s\s+", " ", text) # normalize spaces
5      text = re.sub(r"[0-9]*\s*\n\s*\u00A9.*\nBranchenreport.*\n",
       ↪  " ", text) # remove headers and footers on each page
6      text = re.sub(r"\s*\u25ba\s*", ". ", text) # replace listing
       ↪  symbol with dot
7      text = re.sub(r"\s*\n\s*", " ", text) #remove linebreaks
8      return text
```

Quelltext 5.1: Bereinigung des Textes

Die genannten Schritte werden im Modul **parse.py** durch die Funktion *clean_text* realisiert. Dafür werden die betroffenen Textstellen durch reguläre Ausdrücke gefunden und ersetzt, wie in Quelltext 5.1 zu sehen ist.

5.2 Regelpipeline

5.2.1 Regelbasiertes Erkennen von Entitäten

Zur regelbasierten Annotation von Entitäten kommt spaCys EntityRuler[3] zum Einsatz. Dieser erkennt mithilfe von Mustern Entitäten im Text und annotiert die Tokens entsprechend. Die Muster werden in einer separaten Datei im JSON-Format definiert und bei der Initialisierung geladen. In den Mustern können zuvor annotierte linguistische Merkmale verwendet werden, weshalb die Entitätserkennung in der Pipeline nach der linguistischen Analyse steht.

```
{
    "label":"VAL_PER",
    "patterns":[
        [{"POS":"NUM"},{"LOWER":"%"}],
        [{"POS":"NUM"},{"LOWER":{"REGEX":"prozent(punkte)?"}}]
    ]
}
```

Quelltext 5.2: Musterdatei für Prozentangaben

Quelltext 5.2 zeigt die Musterdatei für die Entität *VAL_PER*, welche Prozentangaben beschreibt. Jede Datei entspricht einem Dictionary mit den Schlüsseln *label* und *patterns*. Jedes Muster in *patterns* ist wiederum eine Liste aus Dictionaries, in der jedes Listenelement genau ein Token beschreibt. Zu jedem Token können mehrere Eigenschaften angegeben werden. In unserem Beispiel wird im ersten Muster zunächst nach einem Token mit dem Part-of-speech-Tag (POS) „NUM“, gefolgt von einem Token mit dem nach Kleinbuchstaben konvertierten Text (LOWER) „%“ gesucht. Im zweiten Muster wird statt des Textes selbst ein regulärer Ausdruck (REGEX) angegeben, um Redundanz zu vermeiden. Das führt hier dazu, dass sowohl „Prozent“ als auch „Prozentpunkte“ gematcht werden.

Beim Durchlaufen dieser Pipeline-Komponente werden alle gefundenen Entitäten im Feld **Doc.ents** gespeichert. Dadurch kann die regelbasierte sowie die neuronale Komponente bei der Klassifikation von Ereignis-Argumenten auf die Entitäten zugreifen. Mit den folgenden Codezeilen wird der *EntityRuler* unter der Bezeichnung *entity_ruler* initialisiert und in die Pipeline gehängt:

```
patterns = read_ent_patterns(ent_patterns)
nlp.add_pipe('entity_ruler').add_patterns(patterns)
```

[3] https://spacy.io/api/entityruler

Die Funktion *read_ent_patterns()* liest die Muster aus dem Pfad *ent_patterns* ein und fügt sie dem EntityRuler hinzu.

5.2.2 Regelbasierte Extraktion von Ereignissen

Für die regelbasierte Extraktion von Ereignissen wurden die Komponenten *trigger_ruler* und *arg_ruler* geschrieben, die Ereignis-Trigger bzw. Ereignis-Argumente identifizieren und annotieren. Beide sind als Klassen implementiert und befinden sich im Modul **rulers.py**.

Intern wird in beiden Fällen der von spaCy bereitgestellte DependencyMatcher[4] benutzt. Dieser erlaubt es Muster, ähnlich der Semgrex-Syntax, über dem Syntaxbaum zu definieren und so grammatikalische Strukturen nachzubilden.

```
1 @Language.factory("trigger_ruler", requires=["token.ent_type",
  ↪ "token.dep", "token.pos"])
2 def trigger_ruler(nlp: Language, name: str, trigger_patterns:
  ↪ str):
3     return TriggerRuler(nlp, trigger_patterns)
```

Quelltext 5.3: Factory-Funktion für die Komponente *trigger_ruler*

Durch attributierte Factory-Funktionen (siehe Quelltext 5.3) werden spaCy die eigenen Komponenten bekannt gemacht. Der erste Parameter im Attribut *@Language.factory* ist die Bezeichnung, unter der die Komponente durch spaCy gefunden werden kann. Im Feld *requires* kann angegeben werden, welche Annotationen bereits zuvor existieren sollen. Da wir Muster über dem Syntaxbaum definieren wollen, wird bspw. in jedem Fall die *dep*-Annotation (entspricht der grammatikalischen Rolle) an jedem Token benötigt.

Erkennen von Triggern

Zuerst werden die Ereignis-Trigger identifiziert und danach deren Argumente. Daher gibt es für beides eigene Musterdateien. Der Vorteil davon ist, dass die Trigger-Annotationen in den Mustern der Argumente verwendet werden können. Durch diese Abstraktion sind die Musterdateien weniger komplex und repetitiv.

Das Muster in Quelltext 5.4 beschreibt Satzkonstruktionen, in denen die Verben „zunehmen“ oder „zulegen“ mit getrennten Silben vorkommen. So würde dieses Muster bspw. im Satz „Die Materialkosten nahmen 2020 um 4% zu.“ das Token „nahmen“ als Event-Trigger für einen Steigerung markieren. Die folgenden Schritte werden beim Mustervergleich sinngemäß durchlaufen:

[4] https://spacy.io/api/dependencymatcher

```
{
    "RIGHT_ID": "TRI_RAISE",
    "RIGHT_ATTRS": {"LEMMA": {"REGEX":"([Nn]ehmen|[Ll]egen)"}}
},
{
    "LEFT_ID": "TRI_RAISE",
    "REL_OP": ">>",
    "RIGHT_ID": "raise_indicator",
    "RIGHT_ATTRS": {"LOWER": {"REGEX":"zu"}, "DEP":"svp"}
}
```

Quelltext 5.4: Muster für den Ereignis-Trigger Steigerung

1. Es wird nach einem Token gesucht, dessen Lemma dem regulären Ausdruck ($[Nn]ehmen|[Ll]egen$) entspricht. Sobald dieses gefunden wurde, wird ihm die ID *TRI_RAISE* zugeordnet.
2. Analog wird das Wort „zu“ gesucht, wobei zusätzlich noch die Rolle im Abhängigkeitsbaum des Satzes (DEP) überprüft wird. Da es sich um eine getrennte Wortsilbe handelt, wird dafür das Label *svp (seperable verb prefix)* verlangt. Danach wird dem gefundenen Wort die ID *raise_indicator* gegeben.
3. Zuletzt wird der Semgrex Ausdruck $TRI\_RAISE \gg raise\_indicator$ ausgewertet. Dieser sorgt dafür, dass „zu“ im Abhängigkeitsbaum unterhalb von „nahmen“ gesucht und gefunden wird.

```
def _on_trigger_match(self, matcher, doc, i, matches):
    match_id, token_ids = matches[i]
    _, patterns = self.trigger_matcher.get(match_id)

    for i in range(len(token_ids)):
        token = doc[token_ids[i]]
        label = patterns[0][i]["RIGHT_ID"]
        if label.startswith("TRI"):
            #Add trigger annotation in order for the rulebased
            ↪  and neural arg classification to work
            token._.trigger_type = label
            doc._.trigger_indices.append(token.i)
```

Quelltext 5.5: Annotierung der gefundenen Ereignis-Trigger

Die in Quelltext 5.5 dargestellte Funktion *_on_trigger_match()* wird jedesmal auf-

gerufen, sobald ein Trigger im Text gefunden wurde. Daraufhin wird am entsprechenen Token das Feld *trigger_type* auf den gefundenen Trigger-Typen gesetzt (Zeile 10). Die Startindizes der Trigger werden am Dokument selbst gesetzt (Zeile 15) und sind später für den neuronalen Klassifizierer nützlich, um Paare aus Trigger und Argument-Kandidaten zur Klassifikation zu bilden.

Erkennen von Argumenten

Die Komponente *arg_ruler* ist von entscheidender Bedeutung, denn die durch sie entstandenen Annotationen werden als sog. Gold-Standard benötigt, um den neuronalen Argument-Klassifizierer zu trainieren.

```
1  {
2      "RIGHT_ID": "trigger",
3      "RIGHT_ATTRS": {"_":{"trigger_type": {"IN":["TRI_RAISE",
       ↪  "TRI_FALL"]}}}
4  },
5  {
6      "LEFT_ID": "trigger",
7      "REL_OP": ">",
8      "RIGHT_ID": "ARG_KNZ",
9      "RIGHT_ATTRS":{"DEP":"sb"}
10 }
```

Quelltext 5.6: Muster zur Identifikation der Kennzahl

Analog zum *trigger_ruler* werden auch hier Muster für den *DependencyMatcher* definiert. Quelltext 5.6 zeigt ein Beispielmuster zur Erkennung einer Kennzahl innerhalb einer Steigerung oder Minderung. Im Beispiel wird ein Trigger als Ankerpunkt gewählt, der an dem zuvor annotierten Feld *trigger_type* erkannt wird (Zeile 1-4). Durch den Semgrex-Ausdruck *trigger* > *ARG_KNZ* wird dem direkt darunterliegenden Subjekt (*sb*) die ID *ARG_KNZ* zugewiesen (Zeile 5-10).

Im Satz „Der Personalaufwand fiel 2020 um 3 Prozent.“ würde „fiel“ als *trigger* und folglich „Personalaufwand“ als *ARG_KNZ* erkannt werden, weil letzteres das Subjekt ist. Dadurch kann nur durch die grammatikalische Struktur eine Kennzahl identifiziert werden, ohne dass das Wort selbst analysiert werden muss.

Die gefundenen Argumente müssen so annotiert werden, dass der neuronale Klassifizierer damit arbeiten kann. Quelltext 5.7 zeigt einen Auszug aus der Funktion *_on_arg_match()* des ArgRulers. Jedes Doc-Object hat eine *Extension-Property arg_predictions* in dem die klassifizierten Argumente gespeichert werden. Dabei handelt es sich um ein Dictionary das als Schlüssel einen *offset* (i_t, i_a) verwendet, wobei i_t der Startindex des Triggers und i_a der des Arguments ist. Hier wird

```
 #Fill arg_predictions for neural network training (gold data)
for arg in args:
    offset = (trigger.start, arg.start)
    if offset not in doc._.arg_predictions:
        doc._.arg_predictions[offset] = {}
    for label in ARG_LABELS:
        if label == arg.label_:
            doc._.arg_predictions[offset][label] = 1.0
        else:
            doc._.arg_predictions[offset][label] = 0.0
```

Quelltext 5.7: Annotation der gefundenen Argumente

deutlich, dass die Extraktion von Ereignissen lediglich eine komplexere Form der *Relationship Extraction* darstellt, weil wir durch diesen Offset eindeutig ein Paar identifizieren, dem wir einen Beziehungstypen (z.B ARG_KNZ) zuordnen wollen.

Zu jedem Offset existiert wiederum ein weiteres Dictionary, das für alle Argument-Label eine Zahl zwischen 0 und 1 enthalten soll, die angibt, mit welcher Wahrscheinlichket das jeweilige Label zutrifft. Da wir die Argumente im ArgRuler mittels Mustern eindeutig klassifizieren, erhält hier das zutreffende Label die Zahl 1.0 (entspricht 100%), während die restlichen Klassen mit 0 vorbelegt werden.

5.3 Neuronaler Argument-Klassifizierer

5.3.1 Trainierbare Pipeline-Komponente

Der neuronale Argument-Klassifizierer ist wie die anderen Komponenten auch als Klasse implementiert und befindet sich im Modul *arg_classifier.py*. Er hat eine Referenz auf das neuronale Modell, das während des Trainings angepasst wird und bei der Initialisierung übergeben wird. Zusätzlich erbt er von der Klasse *TrainablePipe*[5], die in spaCy enthalten ist und u.a. die nachfolgenden Funktionen für trainierbare Pipeline-Komponenten vorgibt.

Die Funktion **__call__()** wird zur Laufzeit aufgerufen, sobald ein Doc-Objekt die Komponente passiert. Zunächst werden Vorhersagen des Modells ermittelt und anschließend im Feld *arg_predictions* des Doc-Objektes annotiert.

Quelltext 5.8 ist ein Auszug aus der Methode **update()**, die den Lernalgorithmus implementiert. Zuerst wird die Funktion *begin_update()* des Modells aufgerufen, die sowohl die Vorhersagen als auch einen Backpropagation-Callback zurückgibt. Anschließend werden die Vorhersagen mit der Referenz verglichen und daraus

[5]https://spacy.io/api/pipe

```
docs = [eg.predicted for eg in examples]
predictions, backprop = self.model.begin_update(docs)
gradient, loss = self.get_loss(examples, predictions)
backprop(gradient)
    if sgd is not None:
        self.model.finish_update(sgd)
```

Quelltext 5.8: Aktualisierung des neuronalen Modells

der Verlust und der Gradient berechnet. Durch den Aufruf des Backpropagation-Callbacks wird der berechnete Gradient an die Schichten im Modell weitergeleitet. Zur Aktualisierung der Parameter des Modells wird schlussendlich *finish_update()* am Modell aufgerufen.

Die Methode **score()** berechnet anhand mehrerer Referenzdokumente Precision, Recall und F-Score für die Klassifizierung durch das Modell.

Zum Trainieren des neuronalen Klassifizierers wird der von spaCy implementierte Befehl **spacy train** verwendet. Dieser erwartet eine Konfigurationsdatei (siehe Anhang A), in der Einstellungen für den Trainingsprozess getroffen werden.

5.3.2 Implementierung und Konfiguration des neuronalen Modells

Zur Klassifizierung von Argumenten eines Ereignisses entwickeln wir ein neuronales Modell, dessen Parameter sich während des Trainingsprozesses aktualisieren. Dieses Modell besteht aus mehreren Schichten, die jeweils eine eigene Funktion erfüllen und ihre Ausgabe an die nächste Schicht weitergeben.

Abbildung 5.1 gibt einen Überblick über den Datenfluss durch das Modell, das sich wiederum in zwei Teilmodelle aufteilen lässt.

Das erste Teilmodell **arg_instance_tensor.v1** ist für die Umwandlung der Doc-Objekte in einen sog. Instanztensor zuständig. Als Eingabe wird eine Liste von Doc-Objekten übergeben, aus denen die zu klassifizierenden Instanzen gelesen werden (**get_instances**). Dabei handelt es sich um Paare $(trigger, arg)$ aus Trigger und Argument-Kandidaten (alle Tokens in einem bestimmten Abstand vom Trigger). Die Trigger stammen aus dem durch Regeln annotierten Datensatz. Mithilfe der **tok2vec**-Schicht werden die Doc-Objekte in eine vektorielle Darstellung umgewandelt, sodass das neuronale Netz damit rechnen kann. Zuletzt wird für jede Instanz der trigger-Vektor mit dem arg-Vektor konkateniert und so ein einziger Vektor erzeugt, der als Eingabe für die Klassifizierungsschicht dient. Da Argumente theoretisch auch aus mehr als einem Token bestehen können, werden deren Vektoren in solchen Fällen durch die **pooling**-Schicht zu einem Vektor zusam-

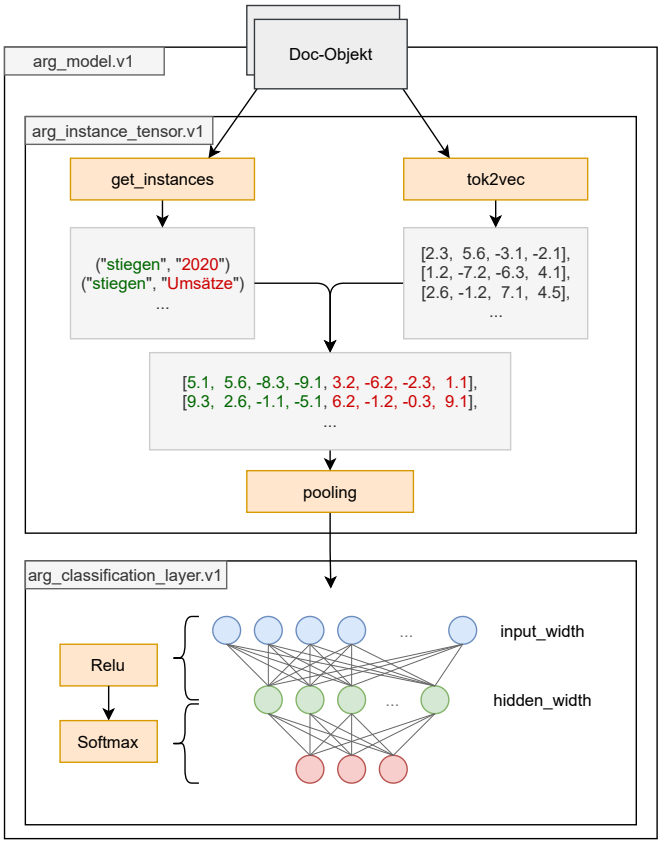

Abbildung 5.1: Modell zur Klassifikation von Argumenten

mengefasst. Der genannte Ablauf wird durch die Funktion **instance_forward()** implementiert.

Das zweite Teilmodell **arg_classification_layer.v1** ist für die eigentliche Klassifizierung der Instanzen verantwortlich. Hierfür wird die Vektorrepräsentation der (trigger, arg)-Paare durch ein dreischichtiges neuronales Netz geschleußt. Die Anzahl der Neuronen in der Eingabeschicht (blau) *input_width* entspricht dabei der Dimension der Instanzvektoren, wobei jedes dieser Neuronen die ReLu-Funktion als Aktivierungsfunktion besitzt. Die verborgene Schicht hat die Breite *hidden_width*, die in der Konfigurationsdatei festgelegt wird. Die Breite der Ausgabeschicht entspricht der Anzahl der Klassen und wird bei der Initialisierung des Modells ermittelt. Als Aktivierungsfunktion wird hier die Softmax-Funktion verwendet, um den verschiedenen Klassen Wahrscheinlichkeiten zuzuordnen.

Die Architekturen des neuronale Modells befinden sich im Modul arg_model.py. Über das Attribut *@spacy.registry.architecture.register* können die Architekturen registriert und in der Konfigurationsdatei *arg_train.cfg* referenziert werden. Zur Implementierung des Modells wurde die *Deep Learning* Programmbibliothek

Thinc[6] genutzt, die ebenfalls von Explosion entwickelt wird und deshalb gut mit spaCy zusammenarbeitet.

```
@spacy.registry.architectures.register("arg_model.v1")
def create_arg_model(
    create_instance_tensor: Model[List[Doc], Floats2d],
    classification_layer: Model[Floats2d, Floats2d],
) -> Model[List[Doc], Floats2d]:
    with Model.define_operators({">>": chain}):
        model = create_instance_tensor >> classification_layer
        model.attrs["get_instances"] =
         ↪  create_instance_tensor.attrs["get_instances"]
    return model
```

Quelltext 5.9: Verkettung der Teilmodelle

Quelltext 5.9 zeigt die Funktion, die für die Erstellung des Gesamtmodells verantwortlich ist. Die beiden Teilmodelle werden als Parameter übergeben und in Zeile 7 verkettet. Das sorgt dafür, dass die Ausgabe von *create_instance_tensor* zur Eingabe von *classification_layer* wird.

```
[components.arg_classifier.model]
@architectures = "arg_model.v1"

[components.arg_classifier.model.classification_layer]
@architectures = "arg_classification_layer.v1"
input_width = 256
hidden_width = 128

[components.arg_classifier.model.create_instance_tensor]
@architectures = "arg_instance_tensor.v1"
```

Quelltext 5.10: Konfiguration des neuronalen Modells in arg_train.cfg

Die Funktion **create_classification_layer()** baut das neuronale Netz, das zur Klassifikation der Argumente verwendet wird. Die Ermittlung der Breiten der Schichten erfolgt über die Konfigurationsdatei (siehe Quelltext 5.10). Hier wird dem Klassifizierer auch das Gesamtmodell übergeben (Zeile 2).

[6] https://thinc.ai/

5.4 Benutzung des Prototyps

Die Benutzung des Prototyps erfolgt über die Konsole.

SpaCy bietet die Möglichkeit in der Datei *project.yml* (siehe Anhang B) Befehle und Workflows zu definieren, die über das Command Line Interface durch den Befehl *spacy project run [BEFEHL]* ausgeführt werden können[7]. Die folgenden Befehle stehen zur Verfügung:

parse:
Extrahiert den Text und die Metadaten aus PDF-Dokumenten bzw. XML-Dateien und speichert die Daten unter ./data.

annotate:
Schickt den Text durch eine Regelpipeline, erstellt so einen annotierten Textkorpus und speichert ihn unter ./corpus.

train:
Trainiert den Argument-Klassifizierer mit den annotierten Daten und speichert das Modell unter ./training.

evaluate:
Evaluiert das neuronale Modell mit durch Regeln annotierten Testdaten und gibt das Ergebnis aus.

predict:
Wendet den Klassifizierer in einer Extraktionspipeline auf neue Daten an und speichert die Ergebnisse unter ./predictions.

[7] https://spacy.io/usage/projects

6 Evaluation

6.1 Klassifikation

6.1.1 Metriken

Zur Evaluation des neuronalen Modells wurde das Skript *evaluate.py* geschrieben, das mit dem Befehl *spacy project run evaluate* ausgeführt werden kann. Darin wird der durch Regeln annotierte Testdatensatz als Referenz für das Modell benutzt. *Recall*, *Precision* und *F-Score* werden durch die Funktion *score()* des Klassifizierers berechnet.

Threshold	Precision_μ	Recall_μ	F-Score_μ
0.00	16.67	100.00	28.57
0.05	96.28	97.47	96.87
0.10	96.50	97.39	96.94
0.20	96.67	97.24	96.96
0.30	96.77	97.15	96.96
0.40	96.88	97.06	96.97
0.50	96.99	96.99	96.99
0.60	97.06	96.87	96.96
0.70	97.15	96.76	96.95
0.80	97.24	96.66	96.95
0.90	97.39	96.49	96.94
0.99	97.70	95.66	96.67
1.00	98.00	94.64	96.29

Tabelle 6.1: Ergebnisse der Evaluation des neuronalen Modells

Tabelle 6.1 zeigt die Resultate der Evaluation. Der Threshold gibt an, ab welcher Wahrscheinlichkeit eine Vorhersage als wahr angesehen wird. Die Metriken sind bei fast allen Thresholds sehr hoch und liegen im Bereich zwischen 0,96 und 0,98. Das deutet darauf hin, dass die Vorhersagen des Klassifizierers sehr eindeutig sind, also für eine Klasse nahe 0 bzw. 1 liegen.

6.1.2 Einschränkungen der Resultate

Bei der Evaluation werden aktuell die durch Regeln annotierten Testdaten herangezogen. Das bedeutet, dass die Metriken nicht die tatsächliche Performance des Modells wiedergeben, sondern durch Resultate der regelbasierten Annotierung verzerrt werden. Um einen unabhängige Evaluation durchzuführen ist eine manuelle Annotierung von Testdaten alternativlos.

6.2 Anforderungen

A1: Extraktion von Branchenentwicklungen

teilweise erfüllt: Die Brancheninformationen werden in den meisten Fällen korrekt extrahiert. Allerding gibt es viele Konstellationen, in denen das Ergebnis fehlerhaft ist. Insbesondere wenn ein Satz mehr als eine Branchenentwicklung enthält, werden die Argumente manchmal den falschen Triggern zugeordnet. Das liegt daran, dass der Klassifizierer sich zu sehr an die Regeln angepasst hat und nicht gut generalisiert. D.h. um neue Konstellationen abzudecken, müssen weitere Regeln hinzugefügt werden.

A2: Regelbasierte Annotierung von Trainingsdaten

erfüllt: Die Trainings- und Validierungsdaten, die zum Trainieren des neuronalen Klassifizierers benutzt werden, werden vollständig programmatisch annotiert. Dabei werden zur Erkennung von Entitäten und Ereignissen Mustervergleiche verwendet.

A3: Klassifikation durch künstliche neuronale Netze

teilweise erfüllt: Zur Klassifikation von Ereignis-Argumenten wurde ein neuronales Modell implementiert. Allerdings wird der Ereignistyp selbst (Steigerung oder Minderung) durch Regeln ermittelt.

A4: Extraktion aus PDF-Dokumenten

erfüllt: Die Textdaten werden aus den PDF-Dokumenten extrahiert und durch reguläre Ausdrücke bereinigt. Sätze, die grammatikalisch unvollständig sind, also aus Tabellen oder Grafiken stammen, werden von der weiteren Verarbeitung ausgeschlossen.

A5: Berücksichtigung von Metadaten

erfüllt: Zu jedem PDF-Dokument werden die zugehörigen Metadaten aus einer xml-Datei gelesen. Die Metadaten werden bei der Extraktion zusammen mit den

extrahierten Informationen gespeichert und können so zugeordnet werden.

A6: Strukturierte Speicherung

erfüllt: Die extrahierten Information werden im Workflow *predict* zusammen mit den Metadaten im JSON-Format abgespeichert.

7 Zusammenfassung

Das Ziel dieser Arbeit bestand darin, eine Extraktionspipeline zu entwerfen, die regelbasierte Vorgehensweisen mit maschinellem Lernen verbindet und ohne manuellen Annotierungsaufwand auskommt.

Dieses Ziel wurde erreicht. Es konnte gezeigt werden, dass es möglich ist, ein neuronales Netz mit programmatisch annotierten Daten zu trainieren und damit Informationen aus Texten zu extrahieren.

In Kapitel 3 haben wir verschiedene Metriken für die Evaluation von Informationsextraktionssystemen vorgestellt. Precision, Recall und F-Score eignen sich, um das Ergebnis einer Klassifikation zu bewerten.

In Kapitel 4 wurden Verfahren präsentiert, mit denen Informationen aus unstrukturierten Texten extrahiert werden können. Das Aufgabengebiet der Event Extraction hat sich zur Modellierung von Brancheninformationen als geeignet erwiesen, weil dadurch Beziehungen zwischen mehr als zwei Entitäten abgebildet werden können. Es wurde eine Pipeline entworfen, die Ereignisse basierend auf Mustervergleichen extrahieren kann und so Trainingsdaten für ein neuronales Netz generiert. Zur Klassifikation von Ereignis-Argumenten wurde ein neuronales Modell entwickelt, das anschliesënd zusammen mit Mustervergleichen in einer kombinierten Extraktionspipeline auf neue Daten angewandt werden kann.

Die praktische Umsetzung des Entwurfs wurde in Kapitel 5 beschrieben. Die Programmiersprache Python in Verbindung mit der Open Source Programmbibliothek spaCy bieten sowohl Werkzeuge für regelbasierte Ansätze als auch für trainierbare Komponenten. Die Programmbibliothek Thinc ermöglicht die Implementierung neuronaler Netze, die von Pipeline-Komponenten angesprochen werden können.

Die Evaluation in Kapitel 6 hat gezeigt, dass sich das neuronale Modell sehr gut an die Trainingsdaten anpasst und bei einem Threshold von 0.5 einen F-Score von 96,99 erreicht. Das hat allerdings auch Nachteile. Das Modell hat sich so sehr an die Daten gewöhnt, dass es die Regeln im Prinzip repliziert. Um das zu verhindern, könnten künftig bspw. Techniken der Datenaugmentation implemen-

tiert werden, um mehr Variation in die Trainingsdaten zu bekommen. Da die Qualität der annotierten Daten maßgeblich beeinflusst, wie gut der Klassifizierer arbeitet, ist es zudem sinnvoll, mehr Muster für die Erkennung von Argumenten zu entwickeln.

Anhang A Konfiguration der Trainingspipeline

```
[paths]
train = './corpus/train.spacy'
dev =  './corpus/dev.spacy'
test = './corpus/test.spacy'

raw = null
init_tok2vec = null

[system]
seed = 342
gpu_allocator = null

[nlp]
lang = "de"
pipeline = ["arg_classifier"]
tokenizer = {"@tokenizers":"spacy.Tokenizer.v1"}

[components]

[components.arg_classifier]
factory = "arg_classifier"
threshold = 0.5

[components.arg_classifier.model]
@architectures = "arg_model.v1"

[components.arg_classifier.model.create_instance_tensor]
@architectures = "arg_instance_tensor.v1"

[components.arg_classifier.model.create_instance_tensor.tok2vec]
@architectures = "spacy.HashEmbedCNN.v1"
pretrained_vectors = null
width = 128
depth = 4
embed_size = 2000
window_size = 1
maxout_pieces = 3
subword_features = true

[components.arg_classifier.model.create_instance_tensor.pooling]
```

```
@layers = "reduce_mean.v1"

[components.arg_classifier.model.create_instance_tensor.get_instances]
@misc = "arg_instance_generator.v1"
max_length = 40

[components.arg_classifier.model.classification_layer]
@architectures = "arg_classification_layer.v1"
input_width = 256
hidden_width_1 = 128
hidden_width_2 = 96

[initialize]

[initialize.components]

[corpora]

[corpora.dev]
@readers = "gold_args_reader.v1"
file = ${paths.dev}

[corpora.train]
@readers = "gold_args_reader.v1"
file = ${paths.train}

[training]
seed = ${system.seed}
gpu_allocator = ${system.gpu_allocator}
dropout = 0.1
accumulate_gradient = 1
patience = 1600000
max_epochs = 0
max_steps = 10000
eval_frequency = 1000
frozen_components = []
dev_corpus = "corpora.dev"
train_corpus = "corpora.train"
before_to_disk = null
logger = {"@loggers":"spacy.ConsoleLogger.v1"}

[training.batcher]
```

```
@batchers = "spacy.batch_by_words.v1"
discard_oversize = false
tolerance = 0.2

[training.batcher.size]
@schedules = "compounding.v1"
start = 100
stop = 1000
compound = 1.001

[training.optimizer]
@optimizers = "Adam.v1"
beta1 = 0.9
beta2 = 0.999
L2_is_weight_decay = true
L2 = 0.01
grad_clip = 1.0
use_averages = false
eps = 0.00000001
learn_rate = 0.001

[training.score_weights]
args_micro_p = 0.0
args_micro_r = 0.0
args_micro_f = 1.0
```

Anhang B Konfiguration der Befehle und Workflows

```yaml
title: "Information Extraction from Industry Reports"
description: "Prototype to extract market developements from german industry reports"

# Variables can be referenced across the project.yml using ${vars.var_name}
vars:
  config: "arg_train"
  sample: "assets/dev/2201039.xml"
  gpu: -1
  ents: "patterns/entities"
  triggers: "patterns/triggers"
  args: "patterns/args"
  model: "training/model-last"

remotes:
  default:

# These are the directories that the project needs. The project CLI will make
# sure that they always exist.
directories: ["assets/dev", "assets/eval", "data/dev", "data/eval", "corpus", "patterns", "training", "configs", "predictions"]

assets:
  - dest: "assets/dev"

workflows:
  all:
    - parse
    - annotate
    - train
    - evaluate
    - predict
  generate_corpus:
    - parse
    - annotate

```

```yaml
commands:
  - name: parse
    help: "Parse XML/PDF documents and convert them to JSON format"
    script:
      - "python ./src/parse.py assets/dev data/dev"
      - "python ./src/parse.py assets/eval data/eval"
    deps:
      - "assets/dev"
      - "assets/eval"
    outputs:
      - "data/dev"
      - "data/eval"

  - name: annotate
    help: "Annotate documents using rules and store them in a DocBin (.spacy)"
    # Make sure we specify the branch in the command string, so that the
    # caching works correctly.
    script:
      - "python ./src/annotate.py data/dev ${vars.ents} ${vars.triggers} ${vars.args} ./corpus"
    deps:
      - "data/dev"
      - "${vars.ents}"
      - "${vars.triggers}"
      - "${vars.args}"
    outputs:
      - "corpus/train.spacy"
      - "corpus/dev.spacy"

  - name: train
    help: "Train the full pipeline"
    script:
      - "python -m spacy train configs/${vars.config}.cfg -o training/ --paths.train corpus/train.spacy --paths.dev corpus/dev.spacy -c ./src/pipeline/custom_functions.py"
    deps:
      - "corpus/train.spacy"
      - "configs/${vars.config}.cfg"
    outputs:
      - "training/model-best"
```

```
72
73  - name: evaluate
74    help: "Evaluate on the test data and save the metrics"
75    script:
76      - "python ./src/evaluate.py ./training/model-best
        ↪  ./corpus/test.spacy --print-details"
77    deps:
78      - "training/model-best"
79      - "corpus/test.spacy"
80    outputs:
81      - "metrics/${vars.config}.json"
82
83  - name: predict
84    help: "Extract information from industry reports and store
      ↪  tham in json-Format"
85    # Make sure we specify the branch in the command string, so
      ↪  that the
86    # caching works correctly.
87    script:
88      - "python ./src/predict.py ./data/eval ./predictions
        ↪  ${vars.ents} ${vars.triggers} ${vars.args}
        ↪  ${vars.model}"
89    deps:
90      - "data/eval"
91      - "${vars.ents}"
92      - "${vars.triggers}"
93      - "${vars.args}"
94      - "${vars.model}"
95
96    outputs:
97      - "predictions/"
98
99  - name: clean
100   help: "Remove intermediate files"
101   script:
102     - "rm -rf training/*"
103     - "rm -rf metrics/*"
104     - "rm -rf data/*"
105     - "rm -rf annotation/*"
```

Literaturverzeichnis

Buyko, E., Faessler, E., Wermter, J. & Hahn, U. (2011). SYNTACTIC SIMPLIFICATION AND SEMANTIC ENRICHMENT—TRIMMING DEPENDENCY GRAPHS FOR EVENT EXTRACTION. *Computational Intelligence*, *27*(4), 610–644. https://doi.org/10.1111/j.1467-8640.2011.00402.x

Deng, L. & Liu, Y. (2018). *Deep learning in natural language processing*. Springer. https://doi.org/10.1007/978-981-10-5209-5

Friedl, J. (2009). *Reguläre Ausdrücke*. O'Reilly Verlag. https://books.google.de/books?id=ZBOuAgAAQBAJ

Goyvaerts, J. & Levithan, S. (2010). *Reguläre Ausdrücke Kochbuch*. O'Reilly. https://books.google.de/books?id=ZeoS2PyifmcC

Grishman, R. (2015). Information Extraction. *IEEE Intelligent Systems*, *30*(5), 8–15. https://doi.org/10.1109/MIS.2015.68

Kedia, A. & Rasu, M. (2020). *Hands-On Python Natural Language Processing: Explore tools and techniques to analyze and process text with a view to building real-world NLP applications*. Packt.

Kübler, S., McDonald, R. & Nivre, J. (2009). Dependency Parsing. *Synthesis Lectures on Human Language Technologies*, *2*(1), 1–127. https://doi.org/10.2200/S00169ED1V01Y200901HLT002

Linguistic Data Consortium (Hrsg.). (2005). ACE (Automatic Content Extraction) English Annotation Guidelines for Events (5.4.3 2005.07.01).

Martinez, A. R. (2012). Part-of-speech tagging. *Wiley Interdisciplinary Reviews: Computational Statistics*, *4*(1), 107–113. https://doi.org/10.1002/wics.195

Mitkov, R. (2009). *The Oxford handbook of computational linguistics* (Reprinted.). Oxford Univ. Press. https://books.google.de/books?id=yl6AnaKtVAkC

Nouvel, D. (2016). *Named entities for computational linguistics*. ISTE. https://doi.org/10.1002/9781119268567

Sharma, S., Sharma, S. & Athaiya, A. (2020). Activation functions in neural networks. *International Journal of Engineering Applied Sciences and Technology*, *04*(12), 310–316. https://doi.org/10.33564/ijeast.2020.v04i12.054

Sokolova, M. & Lapalme, G. (2009). A systematic analysis of performance measures for classification tasks. *Information Processing & Management*, *45*(4), 427–437. https://doi.org/10.1016/j.ipm.2009.03.002

Upadhyay, R. & Fujii, A. (2016). Semantic Knowledge Extraction from Research Documents. *Proceedings of the 2016 Federated Conference on Computer Science and Information Systems*, 439–445. https://doi.org/10.15439/2016F221

Vajjala, S. (2020). *Practical Natural Language Processing* (1st edition). O'Reilly Media, Inc; Safari.

Xiang, W. & Wang, B. (2019). A Survey of Event Extraction From Text. *IEEE Access*, *7*, 173111–173137. https://doi.org/10.1109/ACCESS.2019.2956831