Technische Universität Darmstadt Fachgebiet Knowledge Engineering

**Bachelorarbeit**

# Informationsextraktion aus Lebensläufen

Aleksandrs Galickis

30. September 2009

Betreuer: Eneldo Loza Mencía
Prüfer: Prof. Dr. Johannes Fürnkranz

## Ehrenwörtliche Erklärung

Hiermit versichere ich, die vorliegende Bachelorarbeit ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die den Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Frankfurt am Main, den 30. September 2009

**Zusammenfassung**

Die vorliegende Arbeit befasst sich mit der Erstellung eines Informationsextraktionssystems zur Erkennung von Namen und Adressen in tabellarischen Lebensläufen. Hierzu wird der Ansatz der Überführung eines Problems der Informationsextraktion in ein Klassifizierungsproblem gewählt und angewandt. Der Klassifizierer wird mithilfe von Verfahren aus dem Bereich des Maschinellen Lernens, insbesondere Support Vector Machines, trainiert. Ferner werden Vor- und Nachteile der Software-Bibliotheken GATE und MinorThird im Bezug auf die Entwicklung des Systems betrachtet. Es wird gezeigt, dass die Komponentenarchitektur von GATE für die Entwicklung von solchen Anwendungen geeignet ist. Anhand der durchgeführten Experimente zeigt sich außerdem, dass wenngleich das System nicht vollautomatisch Arbeiten kann, es sich jedoch als Arbeitshilfe durchaus eignet und den Benutzer bei der manuellen Eingabe unterstützen kann.

# Inhaltsverzeichnis

# Abkürzungsverzeichnis

| | |
|---|---|
| $P$ | Durchschnittliche Precision bei Cross-Validation, s. S. 28 |
| $R$ | Durchschnittlicher Recall bei Cross-Validation |
| $\sigma_P$ | Standardabweichung der Precision bei Cross-Validation |
| $\sigma_R$ | Standardabweichung des Recall bei Cross-Validation |
| AS | Annotation Set in GATE, s. S. 22 |
| IE | Informationsextraktion |
| JAPE | Java Annotation Patterns Engine, s. S. 18 |
| ML | Machine Learning |
| NLP | Natural Language Processing |
| PR | Processing Ressources in GATE, s. S. 18 |
| SVM | Support Vector Machines |

# Tabellenverzeichnis

# Abbildungsverzeichnis

# 1 Einführung

Sowohl Organisationen als auch Individuen werden zunehmend mit immer größer werdenden Datenmengen konfrontiert. Um dieser Datenflut Herr zu werden, wurden zahlreiche Techniken und Verfahren aus den Bereichen wie Maschinelles Lernen, Information Retrieval oder Information Extraction entwickelt. Vielerorts haben diese Techniken auch den Sprung aus dem „Forschungslabor“ in die Praxis geschafft, vor allem wenn sie zum Kerngeschäft gehören. In anderen Bereichen, wo ihr Einsatz eine Arbeitserleichterung bedeuten könnte, sind sie dagegen nahezu unbekannt. In der vorliegenden Arbeit wird ein Lösungsansatz zu einem Problem vorgestellt, das ursprünglich im Arbeitsalltag eines real existierenden Unternehmens entstanden ist. Dieses Problem soll jedoch als Testfeld für bestimmte Verfahren, Techniken und Software-Produkte dienen, die im Folgenden vorgestellt werden. Der Rest dieses Kapitels erläutert das entstandene Problem. Außerdem werden die für diese Arbeit gesetzten Ziele und Fragestellungen beschrieben.

## 1.1 Ausgangslage

PASS IT-Consulting ist ein multinationales Consulting-Unternehmen mit Sitz im unterfränkischen Aschaffenburg. Das Unternehmen bietet seinen Kunden verschiedene Dienstleistungen in Bereichen Individualsoftware, IT- und Management-Consulting an. Seit 2007 bin ich bei PASS als Werkstudent tätig.

Im Rahmen meiner Werkstudententätigkeit wurde ich gebeten, das folgende Problem zu analysieren. Die Mitarbeiter von PASS IT-Consulting werden ständig mit Profilen von freien Beratern („Freelancern“) konfrontiert. Die Freelancer reagieren auf Ausschreibungen oder Anzeigen, bewerben sich auf eigene Initiative oder nehmen Kontakt auf Fachmessen auf. Die Berater können potentiell in zukünftigen Projekten eingesetzt werden. Damit PASS ihre Fähigkeiten einschätzen kann, stellen Sie ein „Beraterprofil“ zur Verfügung, das i.d.R. elektronisch vorliegt und aus einem kurzen tabellarischen Lebenslauf sowie einer Auflistung der bisherigen Referenzen besteht. Heute werden die Daten aus diesen Profilen manuell ins hauseigene CRM-System übertragen. Die Eingabe gestaltet sich recht zeitaufwendig, weswegen es interessant wäre, die Möglichkeiten einer automatischen Verarbeitung zu evaluieren.

## 1.2 Formalisierung der Problemstellung

Bei der beschriebenen Aufgabenstellung handelt es sich um eine Anwendung aus dem Bereich Informationsextraktion (IE). Ein IE-System ist in der Lage, die relevanten Informationen (hier Profildaten wie Name, Adresse etc.) in einem unstrukturierten Dokument zu finden und sie daraus zu extrahieren. Man kann IE-Systeme definieren als Systeme, „die gezielt domänenspezifische Informationen aus freien Texten aufspüren und strukturieren können, bei gleichzeitigem 'Überlesen' irrelevanter Informationen“ ([CEE+04, S. 502]). Da die zu extrahierenden Daten im Voraus definiert sein müssen, unterscheidet sich IE grundlegend von der Suche nach relevanten Dokumenten anhand von Suchanfragen (Information Retrieval) daher insofern, als die Aufgabe jedes Mal domänenspezifisch zu definieren ist (vgl. [TAC06, S. 2]).

Im Zusammenhang mit IE-Systemen spricht man oft von *Templates,* die die zu extrahierenden Informationen zusammenfassen. Templates bestehen aus einzelnen *Slots* (Datenfeldern). Jeder Slot steht für eine einzelne informative Angabe. Ein Beispiel für ein bereits ausgefülltes Template für Adressen könnte folgendermaßen aussehen (Darstellung nach [CEE+04, S. 503]):

$$\begin{bmatrix} \mathit{Strasse} & : & \text{Zeil} \\ \mathit{Hausnummer} & : & 3 \\ \mathit{PLZ} & : & 60316 \\ \mathit{Stadt} & : & \text{Frankfurt} \end{bmatrix}$$

Die einzelnen Slots sind hier *Strasse, Hausnummer, PLZ, Stadt.* Zusammen geben Sie eine Adresse an. Ein IE-System würde versuchen, diese Slots auszufüllen.

In den letzten 15 Jahren wurden zahlreiche Lösungsansätze entwickelt. Einer davon beruht auf der Möglichkeit, ein IE-Problem als ein Klassifizierungsproblem darzustellen. In einem Klassifizierungsproblem werden nach [Mit97, S. 54] *Instanzen* betrachtet, die als Mengen von Attribut-Wert-Paaren (*Features* bei [TAC06]) darstellbar sind. Dabei ist eine Instanz einer *Klasse* aus einer abzählbaren Menge von Möglichkeiten zuzuordnen. Die Umformulierung erfolgt folgendermaßen: Aus jedem minimalen Dokumentfragment[1] wird eine Instanz erzeugt, die dann mit Features versehen wird, wie etwa dem Textinhalt, Nachbarfragmenten etc. (zur Featurerzeugung s.a. Abschnitte 3.2.1 und 4.2). Die Klassen entsprechen den Template-Slots aus dem Extraktionsszenario, wobei es darüberhinaus einer zusätzlichen Klasse bedarf, die für den nicht extrahierten Text steht (üblicherweise `NONE` genannt).

Der Vorteil des Klassifikationsansatzes besteht darin, dass viele bekannte Klassifikationsalgorithmen verwendet werden können, z.B. Naive-Bayes-Classifier oder Entscheidungsbäume. Es sind jedoch vor allem Hyperebenen-Klassifikatoren wie Support Vector Machines (SVM), die hier die besten Erfolgsaussichten zu haben scheinen (vgl. [TAC06, S. 26]).[2]

[1] Dies können sinnvollerweise Tokens sein, die von einem lexikalischen Scanner ausgegeben wurden.

[2] Andererseits sollte der Naive-Bayes-Ansatz nicht völlig abgeschrieben werden, denn trotz seiner Ungenauigkeit im Vergleich zu SVM, kann er dank seiner Schnelligkeit und Einfachheit etwa zum Testen verwendet werden.

## 1.3 Motivation und Zielsetzung

Die Perspektive, mithilfe der IE-Methoden echte Zeit- und Kosteneinsparungen zu erreichen, ist für jedes Unternehmen attraktiv. Aus diesem Grunde ist die Evaluierung eines IE-Systems, das eine praxisnahe Aufgabe löst, besonders interessant. Dies ist auch die primäre Zielsetzung dieser Arbeit: auswerten, wie gut sich die oben beschriebene Technik (IE durch Überführung in ein Klassifizierungsproblem) für die Datenextraktion aus tabellarischen Lebensläufen eignet.

Ich möchte jedoch auch weitere Fragestellungen einbeziehen, die unmittelbar mit der Beantwortung der obigen Frage zusammenhängen. Jedes IE-System ist einzigartig, da ja seine Struktur stark domänenabhängig ist. Nichtsdestotrotz wäre es eine große Arbeitserleichterung, ein „Rezept“ zu haben, nach dem ein IE-System für einen neuen Anwendungsbereich entwickelt werden kann. In enger Beziehung hierzu steht die Frage nach den Evaluierungsmetriken, mit deren Hilfe die Güte bzw. die Erreichung eines Vorgabeniveaus schnell beurteilt werden kann. Selbstverständlich würde eine detaillierte Diskussion verschiedener Entwurfsansätze den Rahmen dieser Arbeit sprengen, trotzdem werde ich eine Vorgehensweise skizzieren, die mir vernünftig erscheint und die – zumindest in meinem Fall – schnell zu guten Ergebnissen geführt hat.

Gute Werkzeuge sind wichtig, deshalb möchte ich einen Teil der Arbeit auch der Diskussion der zwei Programmbibliotheken widmen, die ich bei der Entwicklung eines Softwareprototyps verwendet habe, GATE und MinorThird. Vor allem bei der letzteren ist die Frage interessant, inwiefern sie sich zur Entwicklung von IE-Anwendungen eignet. Dies ist folgendermaßen motiviert: MinorThird wird von seinen Entwicklern als ein Konkurrenzprodukt zu einer anderen etablierten Bibliothek, WEKA, positioniert; auch hat es im Gegensatz zu WEKA neben reinen Klassifikationsalgorithmen auch IE-spezifische Funktionalitäten. Ein vergleich zweier Bibliotheken gehört zwar ebenfalls nicht zu den Zielen dieser Arbeit, eine Handlungsempfehlung (einsetzen oder nicht einsetzen?) bzgl. MinorThird kann jedoch nach mehreren Monaten intensiven Einsatzes durchaus ausgesprochen werden.

## 1.4 Aufbau der Arbeit

Der Rest der vorliegenden Arbeit ist wie folgt strukturiert. Im Kapitel 2 werden die Grundlagen von IE-Systemen vorgestellt. Anschließend werden im Kapitel 3 die verwendeten bzw. selbst erstellten Software-Werkzeuge vorgestellt. Im Kapitel 4 werden die getroffenen Entscheidungen bezüglich Feature-Extraktion und Performanz-Evaluation beschrieben. Das Kapitel 5 fasst die experimentell ermittelten Ergebnisse zusammen. Das letzte Kapitel 6 gibt einen Überblick über die Ergebnisse der vorliegenden Arbeit und gibt einen Ausblick auf weitere Bereiche, die untersucht werden könnten.

# 2 Einführung in IE-Systeme

In diesem Kapitel wird ein kurzer Überblick über IE-Systeme, deren Aufbau und Komponenten gegeben. Vorteile der Systeme, die Verfahren aus dem Bereich des maschinellen Lernens (ML) verwenden, werden diskutiert. Schließlich wird eine in der Praxis oft verwendete Alternative vorgestellt, mit der das Lebenslauf-Problem ebenfalls gelöst werden kann.

## 2.1 Aufbau eines IE-Systems

Ein IE-System ist klassischerweise modular aufgebaut: „An information extraction system is a cascade of transducers or modules that at each step add structure [...] by applying rules“ ([Hob93]).Die Module fallen je nach Aufgabe in eine der folgenden Kategorien (Liste nach [TAC06, S. 8]), die man als Bearbeitungsphasen ansehen kann:

1. Dokument-Präprozessor;
2. Syntax-Parser;
3. semantische Darstellung, um ein Teil-Template zu generieren;
4. Diskurs-Analyse;
5. Generierung des Ausgabe-Templates.

Welche Module es im Einzelnen sein können, wird in der Abbildung 2.1 zusammengefasst. Weiter folgt eine Übersicht über die Kategorien.

### Dokument-Präprozessor

Darunter versteht man jegliche Komponenten zur Aufbereitung des Dokuments wie bspw. lexikalische Analyzer, Stemmer, Part-of-Speech-Tagger und andere. An dieser Stelle kann auch die Namenserkennung durchgeführt werden, z.B. durch Nachschlagen in einem Wörterbuch (diese Aufzählung folgt weitgehend der Beschreibung bei [CEE+04] und bei [TAC06]).

Nach [Hob93] kann man in diesem Schritt den Text in Zonen einteilen, Sätze bestimmen, irrelevante Sätze herausfiltern und den Parse-Vorgang vorbereiten, indem z.B. Nominalphrasen bereits erkannt werden.

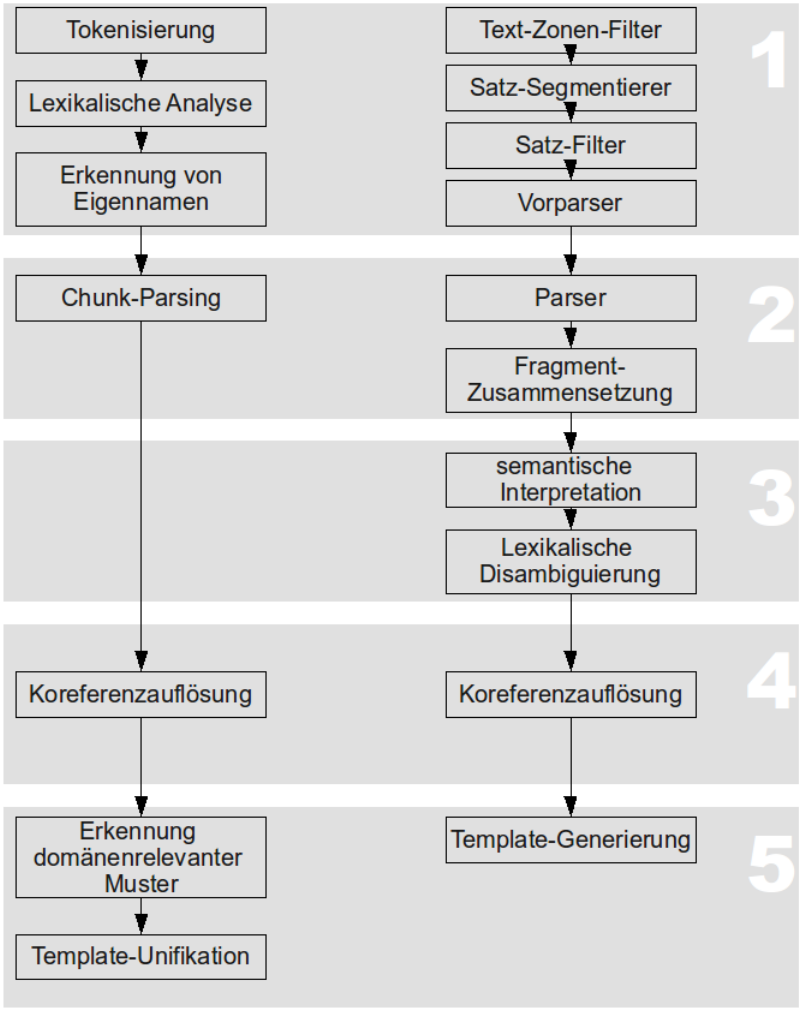


Abbildung 2.1: Module eines IE-Systems nach [CEE+04] und [Hob93]. Die Ziffern in grauen Rechtecken entsprechen den allgemeinen Phasen (s. Text).

### Syntaktische Analyse

In diesem Schritt wird die syntaktische Struktur des Dokuments erfasst. Anfangs wurden Systeme entworfen, die versuchten, Sätze vollständig zu parsen. Dieser Ansatz war wenig erfolgreich und heute ist man dazu übergegangen, eine unvollständige Syntax-Analyse durchzuführen. Dies rührt von der Erkenntnis, dass ein IE-System kein vollständiges Wissen um die syntaktische Struktur des Dokuments braucht, um die Extraktion vorzunehmen. Eine solche Technik ist das Chunk-Parsing (s. [Abn91]): Mithilfe einer kontextfreien Grammatik werden kleine syntaktisch zusammenhängende Bestandteile eines Satzes ermittelt (sog. „Chunks“); in einem nächsten Schritt wird der Zusammenhang zwischen den Chunks innerhalb eines Satzes bestimmt. Durch diesen Ansatz verspricht man sich eine bessere Modularisierung, Separation of Concerns, bessere Nutzung der Ressourcen etc.

### Semantische Darstellung

Nachdem die syntaktische Struktur bestimmt wurde, kann daraus die semantische Struktur abgeleitet werden, was z.B. in Form einer logischen Darstellung erfolgen kann. Desweiteren kann mithilfe der semantischen Informationen zwischen Homonymen unterschieden werden.

### Diskursanalyse

Wichtig ist auch die Analyse des Dokuments über die Satzgrenzen hinaus, weil man nicht davon ausgehen kann, dass alle relevanten Informationen sich in einem Satz oder sonstig isoliert analysierten Fragment befinden. Sie müssen also sukzessiv gesammelt werden. Hierunter fällt auch die Koreferenzauflösung, deren Aufgabe es ist, „festzustellen, ob unterschiedliche linguistische Objekte auf dieselbe Template-Instanz Bezug nehmen.“ ([CEE+04, S. 506]).

### Template-Generierung

Im letzten Schritt erfolgt die Generierung eines ausgefüllten Template. Hierzu sind noch weitere Inferenzschritte notwendig, um bestimmte Restriktionskriterien bzgl. des Ausgabeformats zu erfüllen. Beispiele davon können sein: Vereinheitlichung des Datumsformats, der Nachkommastellen bei Prozentangaben; Auswahl der Slots aus einer vordefinierten Menge etc.

## 2.2 Machine-Learning-Ansätze

Die oben beschriebene IE-System-Architektur geht von Regeln aus (sei es in Form von regulären Ausdrücken, ausführbaren Code oder in einer anderen Repräsentation), die der Systemdesigner von Hand in das System eingegeben hat. Dem steht der Einsatz von ML-Verfahren gegenüber. In [Sar08, S. 278] findet man eine Gegenüberstellung der Anforderungen, die an den

Designer eines handkodierten und eines ML-basierten Systems gestellt werden. Jeder, der ein handkodiertes IE-System entwirft, muss ein Domänen-Experte und Programmierer sein sowie gleichzeitig über tiefes Verständnis der linguistischen Strukturen verfügen, um robuste Extraktionsregeln entwickeln zu können. Ein Designer eines ML-basierten Systems benötigt zwar ebenfalls Domänenwissen, dieses braucht er jedoch nicht zur Herleitung von Regeln, sondern zur Annotierung der Lerndaten. Ferner benötigt er ein gutes Verständnis von ML, um sich zwischen mehreren Modellen und Features (s. a. Abschnitt 1.2) entscheiden zu können.

Es wird außerdem das Problem der „Error-Propagation“ angeführt (vgl. [TAC06]): der Gedanke, dass sich jeder Fehler während der Verarbeitung immer weiter fortpflanzt. In dieser Hinsicht schafft ML Abhilfe, indem die Anforderungen an den Designer dahingehend abändert, dass sie besser bewältigbar erscheinen. Er muss jedoch immer noch über Expertenwissen verfügen, um die Trainingsdaten richtig zu annotieren, und über Verständnis der ML-Verfahren, um zwischen verschiedenen Modellierungsalternativen zu wählen. Zusammenfassend kann man also sagen, dass der Designer auf jeden Fall domänenspezifisches Wissen braucht. Im Falle eines handkodierten Systems muss er dieses Wissen verallgemeinern, um Extraktionsregeln herzuleiten, während in einem ML-basierten System der Verallgemeinerungsschritt von einem spezialisierten Algorithmus vollzogen wird.

Ein ML-basiertes System verwirft jedoch auf keinen Fall die im vorigen Abschnitt dargestellte Architektur. Vielmehr verwendet es Informationen, die von anderen Modulen ermittelt wurden, generalisiert sie, um daraus neue Hypothesen abzuleiten.

## 2.3 Alternative Lösungsmöglichkeit

Es darf nicht der Eindruck entstehen, dass die Entwicklung eines mehr oder weniger leistungsfähigen IE-Systems unumgänglich ist, um das Problem mit der Erfassung von Lebensläufen zu lösen. In der Praxis findet man eine einfachere, sehr pragmatische Lösung. Sie kommt von der Einsicht, dass das eigentliche Verarbeitungsproblem erst entsteht, wenn unstrukturierte Daten maschinell verarbeitet werden müssen. Viele Anbieter zwingen Bewerber dazu, ihre Daten in ein Web-Formular einzugeben, so dass sie dann automatisch strukturiert und maschinenlesbar abgespeichert werden. Auf diese Weise wird die Entstehung von großen Beständen an unstrukturierten Daten verhindert. Beispiele für solche Lösungen kann man etwa bei der Online-Plattform für IT-Projekte GULP[1] oder dem analogen Service bei Resoom Projects[2], aber auch auf Seiten zahlreicher anderer IT-Unternehmen finden. PASS selbst betreibt eine solche Plattform, profi4project[3]. Vor allem bei Resoom Projects trifft man auf ein sehr fortgeschrittenes AJAX-basiertes Formular, das neben Kontaktdaten bspw. auch die Bearbeitung von Skills zulässt.[4]Dieses Vorgehen verhindert die Entstehung unstrukturierter Daten und die damit verbundenen Verarbeitungsschwierigkeiten. Einige Unternehmen lassen entsprechend nur Bewerbungen über Web-Formulare zu, auch wenn dies zu restriktiv ist. Außerdem geben sie dem Be-

[1] `http://www.gulp.de`
[2] `http://projects.resoom.de`
[3] `http://www.profi4project.com`

werber oft die Möglichkeit, weitere Unterlagen hochzuladen, die auch erfasst werden müssen. Schlussendlich ist dies eine weniger benutzerfreundliche Alternative, den Bewerber zur Eingabe aller relevanten Daten zu zwingen, vor allem wenn sie schon einmal in einem anderen Dokument vorliegen.

[4]Unter „Skills“ kann man sich alle möglichen Fähigkeiten vorstellen, die von sehr allgemeinen wie „Betriebswirtschaftliche Kenntnisse“ bis zu sehr speziellen wie „COBOL-Programmierung auf Großrechenanlagen“ reichen können. Im Zusammenhang mit IT-Projekten stehen natürlich IT-bezogene Skills im Vordergrund und spielen dort eine besondere Rolle, da sie die Zuordnung von Mitarbeitern/Beratern zu Projekten erleichtern, s.a. 5.1.

# 3 Aufbau des Systems

Im Rahmen dieser Arbeit wurde ein IE-System im Sinne von Abschnitt 2.1 erstellt. In diesem Kapitel werden verwendete Komponenten – fertige und selbst erstellte – sowie ihre Konfigurationsmöglichkeiten beschrieben.

## 3.1 Verwendete fertige Software-Komponenten

Es wäre unmöglich gewesen, ein funktionsfähiges IE-System fertigzustellen, ohne auf fertige Komponenten zurückzugreifen. Für die Umsetzung wurden primär GATE und MinorThird verwendet, die im Folgenden beschrieben werden. Diese setzen jedoch auf weiteren Bibliotheken auf, die hier nicht alle beschrieben werden können.

### 3.1.1 GATE

GATE versteht sich als eine allgemeine Architektur für Language Engineering (s. [CMBT02]; daher auch der Name GATE, der gerade für General Architecture for Language Engineering steht). Diese Architektur ist komponentenbasiert, wobei die Komponenten einfache JavaBeans sind. Es werden drei Typen von Komponenten unterschieden:

- Language Ressources – Lexika, Korpora, Ontologien etc.;
- Processing Ressources – Einheiten wie Parser, Tokeniser u.a.;
- Visual Ressources – Komponenten zum Visualisieren und Editieren von Daten.

Processing Ressources können zu „Pipelines“ zusammengesetzt werden. Dabei wird ein Dokument oder ein ganzer Korpus in einer vorgegebenen Reihenfolge von den Processing Ressources abgearbeitet.

Eine wichtige Rolle spielen Annotationen, da sie die einzige Möglichkeit sind, bereichsspezifische Informationen den Dokumenten hinzuzufügen (bspw. Tokens bei der lexikalischen Analyse). Jede Annotation hat einen Namen und besteht aus einer Start- und Endposition (ausgedrückt in Zeichen ab Dokumentanfang) sowie optionalen Attributen (Eigenschaften)[1], die als Schlüssel-Wert-Paare gespeichert werden. Annotationen sind in sogenannte „Annotation Sets“ organisiert, die eine logische Aufteilung von Annotationen ermöglichen.

[1] In GATE heißen sie eigentlich „Features“. In diesem Kapitel werden sie jedoch „Eigenschaften“ bzw. „Attribute“ genannt, um Verwechslung mit Features von Lerninstanzen (s. Abschnitt 1.2) im Rahmen des Klassifizierungsproblems auszuschließen.

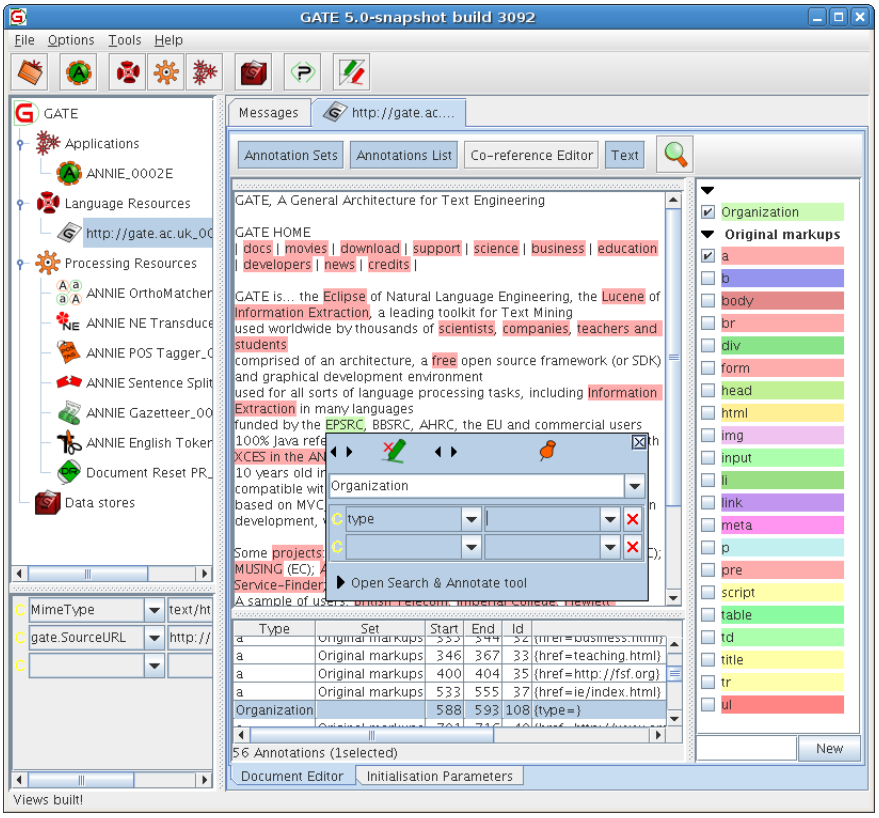


Abbildung 3.1: Das Hauptfenster von GATE. Quelle: GATE-Benutzerhandbuch

GATE hat auch eine bequeme Benutzerschnittstelle – die Autoren führen eine Analogie mit einer integrierten Entwicklungsumgebung auf –, die die Erstellung und Bearbeitung von Komponenten erleichtert, z.B. mittels eines komfortablen Annotations-Editors (s. Abb. 3.1).

Der Ursprung von GATE liegt in der IE-Forschung und -Entwicklung, deswegen wird es auch mit einem eigenen IE-System, ANNIE, ausgeliefert. Es besteht aus solchen nützlichen Komponenten wie Tokeniser, Gazetteer, POS-Tagger und anderen, die das Lösen von IE-Aufgaben erleichtern. ANNIE verfügt jedoch über keine ML-Kapazitäten. In der neuesten Version von GATE gibt es eine ML-API. Diese ist insofern komplexer als die hier vorgestellte Umsetzung, als sie neben der eigentlichen Informationsextraktion auch noch Dokumentenklassifikation und Relationsextraktion unterstützt. Die erweiterten Möglichkeiten werden mit einer höheren Komplexität zu Lasten der Bedienungskomforts bezahlt. Ansonsten ist das Vorgehen bei Informationsextraktion nicht anders als hier beschrieben.

Braucht der Benutzer zusätzliche Funktionen oder Komponenten, so kann er sie mittels selbst erstellter Plug-ins nachrüsten. Die Plug-in-Entwicklung gestaltet sich recht einfach, ebenso ihre Installation. Es reicht, alle JAR-Dateien (eigene Klassendateien sowie benötigte Bibliotheken) in ein Plug-in-Unterverzeichnis zu kopieren und über die GATE-Benutzeroberfläche zu laden.

Um die Eigenschaften der geladenen Komponenten (Name, Typ, Erzeugungs- oder Laufzeitparameter etc.) zu beschreiben, kann man ein XML-basiertes Format verwenden oder – seit der Version 5 von GATE – entsprechende Java-Annotations in den Quellcode integrieren.

Die Vorteile von GATE liegen auf der Hand: Dank seinem modularen Aufbau lassen sich NLP-Systeme schnell und einfach entwerfen. Die API ist einfach und gut dokumentiert, so dass das Schreiben von eigenen Plug-ins unproblematisch ist.

Einige der Processing Ressources (PR), die zum Lieferumgang von ANNIE gehören, wurden im entwickelten IE-System verwendet. An dieser Stelle werden sie kurz vorgestellt.

**Document Reset** Diese PR löscht alle Annotationen aus dem Dokument. Mithilfe der Laufzeiteinstellung `setsToKeep` können Annotation Sets angegeben werden, die beizubehalten sind. Die PR ist in Situationen nützlich, wenn z.B. Annotationen, die während des letzten Durchlaufs erzeugt wurden, nicht mehr benötigt werden.

**Tokeniser** Der Tokeniser führt die lexikalische Analyse eines Dokuments durch. Tokens werden als Annotationen mit dem Namen `Token` gespeichert. Diese Annotationen verfügen über weitere Attribute wie textueller Inhalt des Tokens, orthographische Eigenschaften (Groß-/Kleinschreibung) etc. Der Tokeniser ist flexibel und wird mit zwei Regelsätzen ausgeliefert, weitere können vom Benutzer erstellt werden.

**Gazetteer** Der Gazetteer implementiert ein Lexikon. Wird im Laufe der Verarbeitung ein Wort aus dem Lexikon im Dokument gefunden, so wird die Fundstelle entsprechend annotiert. Es können so z.B. Listen mit häufig vorkommenden Vornamen erstellt werden, um danach als Hinweis auf den Vornamen des Bewerbers zu suchen.

**JAPE Transducer** GATE bietet mit JAPE (Java Annotation Patterns Engine) eine fortgeschrittene Sprache zur Verarbeitung von Annotationen, die auf regulären Ausdrücken basiert. Die Sprache ist sehr mächtig und ermöglicht die Erkennung von komplizierten Mustern in Annotationen, ihre Verarbeitung, Erstellung neuer Annotationen etc. JAPE-Regeln werden im Rahmen dieser Arbeit bei der Erkennung von Adressen eingesetzt (s. Abschnitt 4.2.2). Weitere Hinweise hierzu können [CMBT02, S. 129ff.] entnommen werden.

### 3.1.2 MinorThird

MinorThird (s. [Coh04]) ist eine andere Java-Bibliothek, die zur Verarbeitung von natürlicher Sprache benutzt werden kann. Auch mit MinorThird kann der Text gespeichert und annotiert werden; die Unterstützung von ML-Methoden ist im Gegensatz zu GATE in die Bibliothek integriert. Auch wenn sich MinorThird als Alternative zu anderen ähnlichen Produkten positioniert

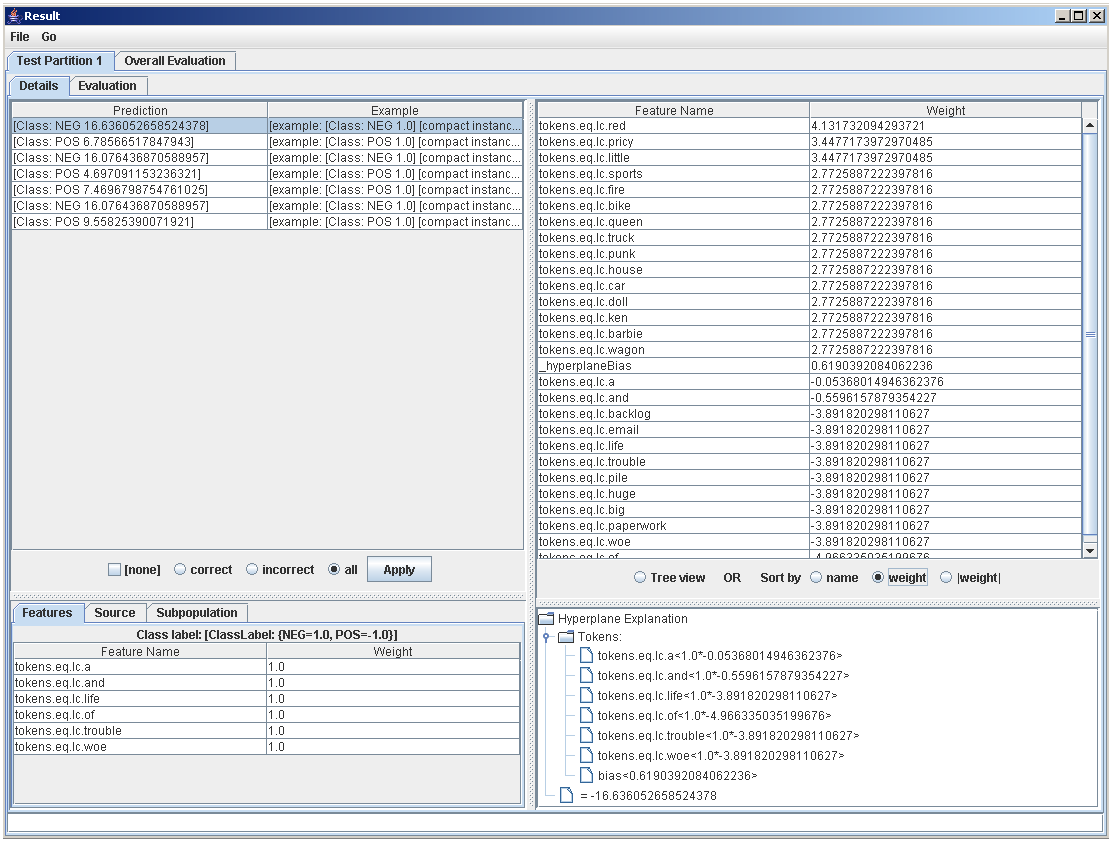


Abbildung 3.2: Auswertung eines Hyperebenenklassifizierers in MinorThird. Quelle: `http://minorthird.sourceforge.net/tutorials/TestClassifier%20Tutorial.htm`

(so auch zu GATE)[2], sind die beiden Lösungen jedoch sehr verschieden. Der Teil der Bibliothek, der die Textbearbeitung ermöglicht, ist in MinorThird eher rudimentär implementiert. Mit den mitgelieferten grafischen Werkzeugen lässt sich die Annotierung von Dokumenten nur mit Mühe bewerkstelligen. Dass die GUI versucht, den ganzen Korpus in einem Fenster anzuzeigen, zeigt zudem, dass sie für kleinere Korpora und Dokumente konzipiert wurde. Dagegen sind Algorithmen zum maschinellen Lernen fester Bestandteil von MinorThird. Die Auswahl an Algorithmen reicht von Naive Bayes über Entscheidungsbäume bis hin zu Support Vector Machines. MinorThird kann in zwei Modi verwendet werden: entweder als reines Klassifizierungswerkzeug (hierzu gibt es ein abgespecktes Distributions-Archiv) oder als IE-Werkzeug, das im Hintergrund auf die Klassifizierungsfunktion zurückgreift.

[2] „Minorthird differs from existing NLP and learning toolkits in a number of ways: Unlike many NLP packages (eg GATE, Alembic) it combines tools for annotating and visualizing text with state-of-the art learning methods [...]“; vgl. `http://minorthird.sourceforge.net/`, Abschnitt „What's Different About MinorThird?“

Im Klassifizierungsmodus ist die Anwendung von MinorThird ganz einfach. Es wird ein Korpus vorbereitet. Will man einen Klassifizierer trainieren, so sind es markierte Beispiele, sonst unmarkierte Instanzen. In beiden Fällen enthalten sie reellwertige Features. Um einen Klassifizierer zu lernen, wird der Korpus an ein entsprechendes Trainer-Objekt (`ClassifierLearner`) übergeben, der am Ende einen Klassifizierer (`Classifier`) zurückgibt. Beim Annotieren ungesehener Instanzen wird umgekehrt der vorhandene Klassifizierer verwendet, um Instanzen Klassen zuzuordnen. Ferner bietet MinorThird Unterstützung bei der Durchführung von Cross-Validation-Experimenten und Datenvisualisierung.

Wegen der offensichtlichen Schwächen von MinorThird im Bereich der Textverarbeitung sind die beiden Produkte eher komplementär zueinander: GATE bietet ein allgemeines Rahmenwerk, um Sprachverarbeitungsanwendungen zu entwickeln; MinorThird bringt ML-Werkzeuge mit, die GATE nicht bietet.

## 3.2 Eigene Processing Ressources

GATE stellt keinerlei Funktionalität zur Verfügung, um MinorThird aus einer Processing Ressource nutzen zu können[3]. Dank seiner leichten Erweiterbarkeit ist es allerdings kein Problem, die entsprechenden Plug-ins selbst zu programmieren. In diesem Abschnitt werden zwei solche PR vorgestellt, die programmiert wurden, um fehlende Funktionalitäten zur Verfügung zu stellen.

### 3.2.1 Feature Extractor

Einen Feature Extractor, also eine Komponente, die Eigenschaften eines Dokuments in Features seiner Tokens überführt, gibt es auch in MinorThird. Anfangs war auch seine Nutzung vorgesehen. Es hat sich jedoch eine Designschwäche von MinorThird herausgestellt, die die Kommunikation zwischen GATE und MinorThird an dieser Stelle unnötig erschwert hat. Beide Bibliotheken haben unterschiedliche Formate zur Beschreibung von annotierten Dokumenten: MinorThird benutzt ein textbasiertes Format mit Stand-Off-Annotations, GATE speichert alles in XML-Dateien ab. Die Konvertierung zwischen den Formaten ist nicht trivial, weil GATE Annotationspositionen in **Zeichen** relativ zum Dokumentbeginn angibt, während MinorThird zuerst eine lexikalische Analyse durchführt und die Positionen auf die **Tokens** bezieht. Nicht nur ist es in diesem Fall nicht möglich, Annotationen anzugeben, die Teile eines Worts umfassen, der Tokeniser ist in MinorThird festkodiert und liefert andere Ergebnisse als die beiden Tokeniser-Konfigurationen in GATE.

Da die Funktionalität des MinorThird Feature-Extractors nicht genutzt werden konnte, wurde eine über XML-Dateien leicht konfigurierbare PR geschrieben, um diese Funktionalität in GATE

[3] Es gibt ein Plug-in, um Mixup, eine JAPE-ähnliche Sprache von MinorThird, zu nutzen. Es ist aber kein Zugriff auf die ML-Funktionen möglich.

nachzubilden. Als Inspiration für das XML-Format diente die ML-API (s. Abschnitt 3.1.1), der Code wurde jedoch unabhängig geschrieben. Am Ende ist eine flexiblere und elegantere Lösung herausgekommen, die kein Neukompilieren nach jeder Änderung erfordert.

Die Feature-Extraktion hat folgende Möglichkeiten:

- Übernahme von anderen Annotationen als Features,
- Übernahme von Attributwerten anderer Annotationen als Features,
- Feature-Erzeugung mit BeanShell-Code[4].

Ferner werden folgende Typen von Features, die nach dem Wertebereich unterschieden werden, unterstützt. In jedem Fall müssen Feature-Werte auf reelle Zahlen abbildbar sein, damit auch lineare Klassifizierer wie SVM damit arbeiten können.

**Reellwertige Features** Diese Features bedürfen keiner speziellen Kodierung. Allenfalls können sie skaliert werden (s. a. Abschnitt 5.1.1).

**Boolesche Features** Features, deren Wert wahr ist, bekommen den Wert 1, sonst 0. Wegen der Sparse-Darstellung werden nur Features, deren Wert wahr ist, explizit gespeichert.

**Textuelle Features** Hierbei muss der Text als Teil des Featurenamen auftreten. Beim Feature `lc` und dem Wert `name` lautet der Featurename `lc.name`. Der Feature-Wert ist stets 1. Dies impliziert eine Sparse-Repräsentation des Featurevektors. Immer wenn das Paar `(lc.name, 1)` nicht auftaucht, bedeutet es, dass das Paar `(lc.name, 0)` vorhanden ist.

Die Sparse-Repräsentation ist bei textuellen Features besonders wichtig, denn ein textuelles Feature generiert für jeden möglichen Wert ein neues boolesches Feature. Textulle Features werden i.d.R. verwendet, um die Tokens auf Features abzubilden. Wenn es z.B. in einem Korpus 1.000 unterschiedliche Wörter und 100.000 Tokens gibt, dann muss man ohne Sparse-Darstellung im Featurevektor jeder Instanz 1.000 Werte speichern, obwohl nur ein Bruchteil davon ungleich Null ist.

Da bei der Umwandlung eines Tokens in eine Lerninstanz jegliche Kontextinformationen verloren gehen, muss die Umgebung eines Tokens über die Features einbezogen werden. Hierzu gibt es eine Reihe von sogenannten Windowing-Verfahren (s. [LM09, S. 3]), die in den Feature-Satz eines Tokens auch die Features seiner Umgebung, eines Fensters mit einer definierten Größe (in Tokens), angeben. Es gibt mehrere Verfahren, die die relative Position der Features angeben. Die Entfernung kann über ein Gewichtungsschema angegeben werden, d.h. Features in der nächsten Umgebung werden höher gewichtet. Manchmal wird auch auf eine relative Angabe völlig verzichtet, was einer Gleichgewichtung entspricht. Das Verfahren, das hier verwendet wurde, hängt die relative Position an den Featurenamen in Klammern an. Beispiel: Der Name `lc.name(-5)` bedeutet, dass der Textinhalt des Tokens, der fünf Tokens hinter dem aktuellen Token liegt, in

[4] BeanShell ist eine Script-Sprache, die eine stark Java-ähnliche Syntax besitzt. Gültiger Java-1.1-Code ist gleichzeitig gültiger BeanShell-Code. S. a. `http://www.beanshell.org`

Kleinbuchstaben „name“ lautet. Das XML-Format des Feature-Extractors lässt es zu, die Fenstergröße für jedes Token separat anzugeben. [LM09] gibt Größen zwischen $\pm 5$ und $\pm 10$ als üblich an.

In Listing 3.1 sieht man ein Beispiel einer fertigen Konfigurationsdatei. Alle Elemente sind Kinder des Wurzelelements **features,** das selbst keine Attribute enthält.

```
1  <?xml version="1.0"?>
2  <features>
3    <!-- instance legt fest, aus welchen Annotationen
4    Instanzen erzeugt werden. -->
5    <instance set="GATE">Token</instance>
6
7    <!-- String-Features -->
8    <feature name="lc" string="true">
9      <range from="-5" to="5" />
10     <!-- BeanShell-Code -->
11     <extract>Attributes.get("string").toString().toLowerCase()</extract>
12   </feature>
13
14
15   <!-- Wörterbuchbasierte Features -->
16   <feature name="lookup">
17     <extract-from set="GATE" name="majorType">Lookup</extract-from>
18     <range from="-5" to="5" />
19   </feature>
20
21   <!-- Markup-abhängige Features -->
22   <feature name="isInHeader">
23     <copy set="Original markups">h1</copy>
24     <range from="-5" to="5" />
25   </feature>
26
27 </features>
```

Listing 3.1: Beispiel einer XML-Konfigurationsdatei für Features

Weiter folgt die vollständige Beschreibung der Unterelemente von **features**. Alle Zeilenangaben beziehen sich auf das Listing 3.1.

**instance** Legt fest, aus welchen Annotationen Instanzen erzeugt werden. Das Attribut `set` gibt dabei den Namen des Annotation Set (AS) an. Der Inhalt ist der Annotationsname.

In Z. 5 wird mitgeteilt, dass Instanzen aus dem Annotationstyp „Token“ aus dem AS „GATE“ erzeugt werden.

**feature** Beschreibt ein Feature. Das Attribut `name` legt den Namen dieses Features fest. Unterschiedliche Features müssen unterschiedliche Namen haben. Es kann folgende Unterelemente haben:

**range** Leeres Element, das mit den Parametern `from` und `to` die Grenzen des beim Windowing-Verfahren verwendeten Fensters angibt. Ist das Element nicht vorhanden, wird für dieses Feature die Fenstergröße 0 angenommen (die Umgebung wird also gar nicht berücksichtigt).

**extract** Enthält BeanShell-Code. Das Ergebnis seiner Auswertung ist der Featurewert. Es werden automatisch Variablen `Attributes` und `Document` erzeugt. Erstere erlaubt den Zugriff auf alle Eigenschaften der aktuellen Instanz-Annotation, letztere auf das gesamte aktuelle Dokument (als Instanz von `gate.Document`).

**copy** Erzeugt ein boolesches Feature, dessen Wert wahr ist, wenn es an der Stelle eine Annotation des angegebenen Typs gibt. Das Attribut `set` gibt den AS an, der Inhalt ist der Annotationstyp. Ein Wert wird nur dann erzeugt, wenn die Instanzannotation in der Extraktionsannotation vollständig enthalten ist.

**extract-from** Übernimmt den Wert eines Annotationsattributs. Die Eigenschaft `set` gibt den AS an, `name` den Namen der Eigenschaft. Der Inhalt ist der Name des Annotationstyps. Im Beispiel mit dem Feature „lookup“ (s. Z. 17) muss also eine Annotation aus dem AS „GATE“ vom Typ „Lookup“ vorhanden sein und die Instanz-Annotation „Token“ aus demselben AS vollständig umschließen, damit der Wert nicht leer ist.

Die Elemente `extract, extract-from` und `copy` schließen sich gegenseitig aus, d.h. es kann nur eine der drei Extraktionsmethoden verwendet werden.

In einer Pipeline muss der Feature-Extraktor vor der Trainer-PR (s.u.) und nach allen anderen Processing Ressources kommen, denn er bildet eine Brücke zwischen der GATE-API und der Klassifikations-API von MinorThird. Um die Feature-Werte an die nachfolgende PR, den Trainer, weiterzugeben werden sie als Eigenschaften der Instanz-Annotationen gespeichert. Damit sie von der Trainer-PR als Features erkannt werden können, werden ihre Namen mit einem einstellbaren Präfix und Suffix versehen.

**Beispiel** Nachdem der Leser mit einer Flut an Einstellungen konfrontiert wurde, wird ein kurzes vollständiges Beispiel angeführt, um die Ausführungen zu verdeutlichen. Die Processing Ressource „Tokeniser“ führt eine lexikalische Analyse durch. Dabei werden entsprechende Token-Annotationen erzeugt und im AS „GATE“ abgelegt. Dem Feature-Extractor wird dies mit der folgenden Zeile in der Konfigurationsdatei mitgeteilt (Z. 5):

```
<instance set="GATE">Token</instance>
```

Im Beispiel ist die Fenstergröße bei allen Features auf $\pm 5$ gesetzt, d.h. neben dem Token selbst werden 5 Tokens vor und 5 Tokens nach dem aktuellen Token betrachtet. Dies erkennt man an der Anweisung (Z. 9, 18, 24):

```
<range from="-5" to="5" />
```

Während der lexikalischen Analyse wurden in den Token-Annotationen die Textinhalte der Tokens als Eigenschaften mit dem Namen „string“ gespeichert. Da jedoch nicht zwischen der Groß- und Kleinschreibung unterschieden werden soll, werden sie mithilfe des BeanShell-Codes (s. Z. 12) in Kleinbuchstaben konvertiert.

Beim Import eines XML-/SGML-Dokuments erzeugt GATE Annotationen für alle Tags und speichert sie im AS „Original markups“. Um zu testen, ob der Token innerhalb einer Überschrift 1. Orndung vorkommt wird die Anweisung in Z. 23 verwendet. Den Wert 1 bekommen bei diesem Feature nur Tokens, die sich vollständig innerhalb einer Annotation des Typs „h1“ aus dem AS „Original markups“ befinden.

Der Gazetteer (s. 3.1.1) erzeugt Annotationen des Typs „lookup“, deren Attribut „majorType“ den Typ des gefundenen Eintrags (z.B. „Name“, „Vorname“ o.Ä). Dieser Wert wird mit `extract-from` (Z. 17) extrahiert. Einen nicht leeren String bekommen allerdings nur Tokens, die sich vollständig innerhalb einer Annotation des Typs „lookup“ befinden.

Gegeben sei das folgende XML-Dokument:

```
<brief>
  <h1>Brief</h1>
  Sehr geehrter Herr Schmidt
</brief>
```

Ferner gebe es eine Gazetteer-Liste mit Nachnamen, wo u.a. der Name „Schmidt“ vorkommt und den `majorType` „surname“ hat. Für den Token „geehrter“ ergibt sich dann der folgende Featurevektor (Werte, die gleich Null sind, werden wie oben erläutert nicht dargestellt; alle anderen Werte sind 1).

$$\begin{bmatrix} \text{lc.brief}(-2) \\ \text{lc.sehr}(-1) \\ \text{lc.geehrter}(0) \\ \text{lc.herr}(1) \\ \text{lc.schmidt}(2) \\ \text{lookup.surname}(2) \\ \text{isInHeader}(-2) \end{bmatrix}$$

| Parameter | Bedeutung | Voreinstellung |
|---|---|---|
| classASName | Legt fest, in welchem AS sich die Klassen befinden. | Class |
| featurePrefix | Präfix für Feature-Eigenschaften | $ |
| featureSuffix | Suffix für Feature-Eigenschaften | (leer) |
| instanceASName | Name des AS, der Instanzannotationen enthält. Muss dem Inhalt des Elements `instance` in der Feature-Konfigurationsdatei entsprechen. | GATE |
| instanceAnnName | Typ der Instanzannotationen. Muss dem Inhalt des Elements `instance` in der Feature-Konfigurationsdatei entsprechen. | Token |
| classFeatureName | Name des Attributs, das dem Klassennamen enthält. | $CLASS |
| allowedClasses | Kommagetrennte Liste von Klassennamen, die für die Extraktion Berücksichtigung finden. Sie entsprechen den Annotationstypen aus dem AS classASName. | — |
| crossValidatorSeed | Im Cross-Validation-Modus: Einstellung des Seeds des Zufallszahlengenerators. Bei der Einstellung „now“ wird die aktuelle Uhrzeit verwendet. | now |

Tabelle 3.1: Initialisierungs-Einstellungen der Trainer-PR

### 3.2.2 Trainer

Die Processing Ressource Trainer verwendet die Klassifikations-API von MinorThird und ist die Komponente, die ML-Techniken nutzt. Er kann in einem der drei Modi arbeiten:

1. TRAIN. Trainiert einen neuen Klassifizierer auf einem annotierten Korpus.
2. CLASSIFY. Annotiert einen Korpus mithilfe eines vorhandenen Klassifizierers.
3. CROSS_VALIDATE. Misst die Güte der Systemeinstellungen für einen gegebenen Korpus mithilfe des Cross-Validation-Verfahrens (s. Abschnitt 4.1).

Je nach Modus wird eine Reihe an Einstellungen benötigt. Die Einstellungen, die bei der Initialisierung gesetzt werden, sind in der Tabelle 3.1 zusammengefasst. Damit sie nicht bei jeder Erzeugung neu gesetzt werden müssen, werden sie in eine properties-Datei ausgelagert. Bei der Erzeugung erfragt GATE lediglich die URL, die zu dieser Datei führt.

Damit die Kommunikation zwischen dem Feature Extractor und dem Trainer reibungslos funktioniert, muss darauf geachtet werden, dass diese Einstellungen korrekt sind. Insbesondere müssen die Eigenschaften `instanceASName` und `instanceAnnName` mit den Einstellungen

| Parameter | Bedeutung | Voreinstellung |
|---|---|---|
| classifierInit | Im Trainingsmodus: BeanShell-Code zur Initialisierung eines Lerners. Muss eine Instanz der Klasse `edu.cmu.minorthird.classify.ClassifierLearner` liefern. In anderen Modi hat der Parameter keine Bedeutung. | — |
| outputAS | Im Klassifizierungsmodus: AS, in dem neue Annotationen erzeugt werden sollen. In anderen Modi hat der Parameter keine Bedeutung. | Output |
| outputFile | Ein-/Ausgabedatei für den gelernten Klassifizierer. Hat im Cross-Validation-Modus keine Bedeutung. | — |
| usageMode | Benutzungsmodus (s. Text). Mögliche Werte: TRAIN, CLASSIFY, CROSS_VALIDATE. | TRAIN |

Tabelle 3.2: Laufzeit-Einstellungen der Trainer-PR

in der Feature-Konfigurationsdatei übereinstimmen. Der Feature Extractor schreibt die Features in die Instanzannotationen als Attribute mit speziellen Namen. Damit die Trainer-PR diese von anderen Attributen unterscheiden kann, wird über die Eigenschaften `featurePrefix` und `featureSuffix` angegeben, welche Zeichenfolgen an den Anfang und ans Ende des Namens angehängt werden. In der Normalkonfiguration beginnen alle Features mit dem Dollarzeichen „$“. Diese Einstellungen brauchen im Regelfall nicht geändert zu werden. Das gilt auch für die Eigenschaft `classFeatureName`. Klassennamen können als spezielle Features angesehen werden. Es werden nacheinander alle Annotationen im AS betrachtet, dessen Name in `classASName` angegeben ist. Jede Instanzannotation, die sich innerhalb einer von diesen Klassenannotationen befindet, wird über das Attribut `classFeatureName` dieser Klasse zugeordnet. Deswegen ist es auch wichtig, dass sich Klassenannotationen nicht überlappen, denn es kann in diesem Fall nicht vorausgesagt werden, welcher von mehreren Klassen eine Instanzannotation zugeordnet wird.

Die Einstellungen, die zur Laufzeit gesetzt werden müssen, sind in der Tabelle 3.2 zusammengefasst. Manche Parameter werden nur in bestimmten Modi benötigt, in anderen haben sie wiederum keine Bedeutung. Der wichtigste Parameter ist `usageMode`, der den Benutzungsmodus festlegt. Der Parameter `classifierInit` erlaubt eine sehr flexible Auswahl des Lernalgorithmus, da sie nur verlangt, dass das Ergebnis der Auswertung einen `ClassifierLearner` zurückgibt. Dieser muss nicht im MinorThird-Paket enthalten sein, sondern kann von einem anderen Urheber stammen. Die einzige Voraussetzung ist, dass der Klassenname dem Classloader bekannt ist.

# 4 Design-Entscheidungen

Aus dem vorhergehenden Kapitel folgt, dass eine Reihe von Design-Entscheidungen getroffen werden muss, um ein IE-System zu entwerfen. Im Folgenden werden die getroffenen Entscheidungen bzgl. der Ausgestaltung des IE-Systems vorgestellt und erläutert.

## 4.1 Evaluierung der Erkennungsgüte

Die Evaluierung eines IE-Systems hängt oft mit zwei unterschiedlichen Maßen zusammen: Precision und Recall. Im Folgenden gehen wir von tokenbasierten Maßen aus (man könnte sie alternativ auf gesamte Slots beziehen). Die Precision gibt den Anteil der richtig klassifizierten Tokens an; je weniger Fehler der Klassifizierer gemacht hat, umso höher ist die Precision. Der Recall misst den Anteil der richtig klassifizierten Tokens gemessen an der Gesamtzahl der zu extrahierenden Tokens; je mehr richtige Informationen extrahiert wurden, umso höher wird der Recall. Seien $N$ die Anzahl der Tokens in der Referenzannotation, $M$ die Anzahl der erkannten Tokens und $C$ die Anzahl der *korrekt* erkannt Tokens, dann lassen sich Precision ($P$) und Recall ($R$) folgendermaßen definieren (vgl. [MKSW99]):

$$P = \frac{C}{M}; \quad R = \frac{C}{N}.$$

Beim Entwurf eines IE-Systems wird man in der Regel mit folgenden zwei Zielen konfrontiert:

A: Precision maximieren

B: Recall maximieren

In den meisten Fällen sind die beiden Ziele konkurrierend: Erhöht man die Precision, so vermindert dies den Recall und umgekehrt. Nach [Lau05, S. 99] kann das Problem der Zielkonkurrenz zum Beispiel so gelöst werden, dass ein dominierendes Hauptziel gewählt wird, und alle weiteren Ziele als Satisfizierungsziele berücksichtigt werden. In diesem konkreten Fall habe ich beschlossen, die Maximierung der Precision als Hauptziel zu definieren. Diese Entscheidung ist folgendermaßen motiviert: Der Benutzer erwartet nicht (oder sollte zumindest nicht erwarten), dass das System alle benötigten Felder für ihn ausfüllt, daher ist ein leerer Slot nicht schlimm, denn dadurch wird er animiert, diesen Slot von Hand auszufüllen. Umgekehrt ist ein falsch ausgefüllter Slot schlecht, weil der Benutzer den Fehler übersehen kann, so dass falsche Informationen in die Datenbank eingepflegt werden. Daher lautet die Zielsetzung, die Precision des

Klassifizierers zu maximieren. Ein sinnvolles Mindestniveau für den Recall sollte nicht weniger als 40% betragen. Dieses Niveau geben wir hier vor. Der Recall kann auch als „Tie-Breaker“ dienen, falls zwei Konfigurationen ungefähr die gleiche Precision haben.

Natürlich wäre es wünschenswert, *ein* Performanzmaß zu haben, um die Extraktion zu bewerten. Es wurden mehrere solche Maße entworfen. Eines davon ist das F-Measure. Sei $P$ die Precision und $R$ der Recall, ferner sei ein Parameter $\alpha$ gegeben mit $\alpha \in [0;1]$, dann definiert sich das F-Measure wie folgt:

$$F = \frac{PR}{(1-\alpha)P + \alpha R}.$$

Ein üblicher Wert für $\alpha$ ist 0,5. [MKSW99] führt bei dieser Vorgehensweise als Problem an, dass bestimmte Fehlertypen gegenüber anderen bevorzugt werden, ohne dass es einen bestimmten Grund hierfür gibt oder es durch den Parameter $\alpha$ ersichtlich ist. Stattdessen wird der Anteil von falsch klassifizierten Tokens an der Gesamtzahl der Tokens („Slot Error Rate“) als ein adäquateres Fehlermaß vorgeschlagen. Als Vorteil wird genannt, dass alle Fehlertypen gleich berücksichtigt werden. Es wird auch vorgeschlagen, eine andere Gewichtung vorzunehmen, die mehr den eigenen Zielsetzungen entspricht. Ich habe auf ein aggregiertes Performanzmaß verzichtet, um mehr Übersicht über die Systemleistung zu behalten.

Um die Qualität einer Systemkonfiguration zu überprüfen, müssen Experimente auf vorhandenen Daten durchgeführt werden. Experimente direkt auf dem Trainingssatz sind nicht sinnvoll, da der Klassifizierer, der ja auf diesem Datensatz trainiert wurde, die Klassifikation stets fehlerfrei durchführen wird. Dies würde jedoch keine Aussage über die allgemeine Qualität der Konfiguration liefern. Wenn es ausreichend Testdaten gibt, dann ist eine Trennung in einen Test- und einen Trainingssatz möglich. Dies war hier nicht der Fall, da alle Testdokumente langwierig von Hand annotiert werden mussten. Aus diesem Grund habe ich mich für Cross-Validation entschieden.

Bei Cross-Validation wird der Datensatz in $n$ zufällig möglichst gleich große Teile (*Folds*) aufgeteilt. Es werden $n$ Iterationen durchgeführt, bei jeder Iteration wird ein Fold zum Testen verwendet, während der Algorithmus auf $n-1$ verbleibenden Folds trainiert wird. Cross-Validation eignet sich besonders in Fällen, wo nur wenige Testdaten vorliegen (vgl. [HCL03]). Dadurch, dass $n$ Experimente durchgeführt werden, ist es möglich, Durschnittswerte zu bilden.

Folgende Parameter wurden berücksichtigt:

- Durchschnittliche Precision ($P$)
- Durchschnittlicher Recall ($R$)
- Standardabweichung der Precision ($\sigma_P$)
- Standardabweichung des Recall ($\sigma_R$)

Die Standardabweichungen werden berechnet, um abzuschätzen, wie sicher ein Wert ist. Eine große Standardabweichung bedeutet, dass es große Unterschiede zwischen den einzelnen Folds

gibt. Dies vereinfacht auch die Schätzung, ob höherer Parameterwert einer anderen Konfiguration tatsächlich eine Verbesserung liefert. Bei einem Parameter, der eine Standardabweichung von 0,05 hat, kann z.B. eine Veränderung von 0,025 kaum signifikant sein. Weitere Ausführungen hierzu finden sich im Kapitel 5.

## 4.2 Features

Der zentrale Aspekt des Designs eines klassifikationsbasierten Systems ist die Festlegung der Instanz-Features, denn nur über Features erhält der Klassifizierer Informationen über das ursprüngliche Dokument. Wird eine relevante Eigenschaft nicht als Feature erfasst, so ist sie für den Klassifizierer nicht sichtbar und wird daher nicht berücksichtigt. Deswegen ist es so wichtig, die Features festzulegen. [Sar08, S. 298f.] gibt drei verbreitete Feature-Typen an:

**Wortbasierte Features** Manchmal ist das Wort selbst ein wichtiger Hinweis für die Klassifizierung. Im Kontext von Lebensläufen könnte bspw. das Wort „Adresse“ ein starker Hinweis darauf sein, dass auf den Token die Adresse des Individuums folgt.

**Orthographische Features** Hierbei handelt es sich um Features, die bspw. angeben, ob das Wort mit einem Großbuchstaben beginnt, bestimmte Sonderzeichen enthält etc. Zum Beispiel ist die Präsenz des Symbols „@“ ein Hinweis darauf, dass es sich möglicherweise um eine E-Mail-Adresse handelt.

**Wörterbuchbasierte Features** Manchmal ist es lohnend, das Wort mit einem Wörterbuch abzugleichen. Damit lassen sich z.B. verbreitete Vornamen wie „Thomas“ oder „Philipp“ finden.

Der Rest des Abschnitts beschäftigt sich mit der Beschreibung der Features, die für Experimente verwendet wurden. Es wurden zwei getrennte Featuresätze für Namen und Adressen formuliert, die ebenfalls getrennt beschrieben werden. Zuerst werden a-priori-Überlegungen angestellt, welche Features in Frage kommen könnten. Dann werden sie in einer Tabelle zusammengefasst. Die Tabelle ist wie folgt aufgebaut:

**Name** Ein Bezeichner für das Feature

**Wertebereich** Zeichenkette/Wahrheitswert/Reelle Zahl (vgl. Abschnitt 3.2.1)

**Typ** Wortbasiert/Orthographisch/Wörterbuchbasiert (s.o.)

**Erklärung** Kurze Erklärung, wie der Feature-Wert zustande kommt. Eine ausführlichere Erklärung wird ggf. im Text angegeben.

Man beachte außerdem, dass sich das wortbasierte Feature „lc“ bei allen Feldern wiederholt.

| Name | Wertebereich | Typ | Erklärung |
|---|---|---|---|
| lc | Zeichenkette | Wortbasiert | Der Textinhalt des Tokens in Kleinbuchstaben |
| commonFirstname | Wahrheitswert | Wörterbuchbasiert | Einer von 2.636 in Deutschland üblichen Vornamen, die mithilfe des in [Mic07] beschriebenen Programms `gender` extrahiert wurden. |
| commonLastname | Wahrheitswert | Wörterbuchbasiert | Einer der 1.000 Nachnamen, die in der PASS-CRM-Datenbank am häufigsten vorkommen. |
| isUpperInitial | Wahrheitswert | Orthographisch | Wahr, falls der Token mit einem Großbuchstaben beginnt, sonst falsch. |
| keyword | Wahrheitswert | Wörterbuchbasiert | Wahr, falls der Token „Lebenslauf“, „Persönliche Daten“ etc. ist[2] |

Tabelle 4.1: Features für Vor- und Nachnamen

### 4.2.1 Features für die Namenserkennung

Wie erkennt ein Mensch, dass es sich bei einem Token um einen Vor- bzw. Nachnamen handelt? Dies passiert zum Beispiel, wenn der Token als ein verbreiteter Name bekannt ist, etwa „John“ – hier weiß ein Mensch, dass es wohl der Vorname des Bewerbers ist. Ebenso wichtig ist das Wissen, dass Vor- und Nachname oft nebeneinander vorkommen. Erkennt der Leser die Zeichenfolge „Wu“ nicht als einen (verbreiteten chinesischen) Nachnamen, so könnte er darauf immer noch schließen, wenn er der Zeichenfolge „John Wu“ begegnet. Tabellarische Lebensläufe sind so aufgebaut, dass durch ein Signalwort wie „Vorname“, „Name“ oder „Persönliche Daten“ u.a. das Vorkommen des Vor- bzw. Nachnamens gekennzeichnet ist[1]. Ein weiteres schwaches Indiz ist die Tatsache, dass Namen groß geschrieben werden. Aus diesen Überlegungen ergeben sich Features, die in der Tabelle 4.1 zusammengefasst sind.

[1] Singalwörter wie „Persönliche Daten“ oder „Lebenslauf“ kennzeichnen den Namen des Bewerbers, weil er i.d.R. am Anfang des Lebenslaufs, fast unmittelbar nach dem Titel kommt.

[2] Die vollständige verwendete Liste ist: Beraterprofil, CV, Lebenslauf, Nachname, Name, Profil, Qualifikationsprofil, Vorname.

| Name | Wertebereich | Typ | Erklärung |
|---|---|---|---|
| lc | Zeichenkette | Wortbasiert | s. Tabelle 4.1. |
| looksLikeStreet | Wahrheitswert | — | Wahr, falls der Token zu einem Straßennamen gehören könnte (s. Text) |
| looksLikeHouseNumber | Wahrheitswert | — | Wahr, falls der Token zu einer Hausnummer gehören könnte. |
| looksLikeZip | Wahrheitswert | Orthographisch | Wahr, falls der Token eine fünfstellige Zahl ist und damit eine deutsche Postleitzahl sein könnte. |
| isNumber | Wahrheitswert | Orthographisch | Wahr, falls der Token eine Zahl ist |

Tabelle 4.2: Features für Adressen

### 4.2.2 Features für Adressen

In Deutschland enden viele Straßennamen mit Zeichenfolgen wie „Straße“ (z.B. „Konrad-Adenauer-Straße“), „Weg“ („Röderbergweg“), „Gasse“ („Klappergasse“) etc. Es gibt aber reichlich Ausnahmen, bspw. „Am Tiergarten“, „Zeil“ usw. sind Straßennamen; „Heimweg“ dagegen nicht. Dennoch ist eine solche Endung ein gutes Indiz für einen Straßennamen. Ferner sind Hausnummern i.d.R. Zahlen, obwohl es auch da Ausnahmen gibt (z.B. „221b“). Deutsche Postleitzahlen bestehen nur aus Ziffern und haben eine feste Länge von fünf Zeichen. Features, die in Frage kommen, sind in der Tabelle 4.2 zusammengefasst.

Das Besondere an diesen Features ist die Tatsache, dass `looksLikeStreet` und `looksLikeHouseNumber` mithilfe von komplexen JAPE-Regeln (vgl. Abschnitt 3.1.1) erstellt werden. Diese Regeln können folgendermaßen formuliert werden:

1. Setze bei allen Tokens, die auf „straße“, „gasse“, „gässchen“, „weg“, „platz“ oder „allee“ enden, das Feature `looksLikeStreet` auf 1.
2. Falls mehrere aufeinanderfolgende Tokens mit einem Bindestrich enden und das Feature `looksLikeStreet` des darauffolgenden Tokens 1 ist, setze `looksLikeStreet` dieser Tokens ebenfalls auf 1. Dies hilft, solche Namen zu erkennen wie „Konrad-Adenauer-Straße“.
3. Falls ein Token eine Zahl ist und der vorhergehende Token wie ein Straßenname aussieht (s.o.), dann setze das Feature `looksLikeHouseNumber` auf 1. Falls der nächste Token ein Buchstabe ist, setze bei diesem `looksLikeHouseNumber` ebenfalls auf 1. Dies

erkennt Hausnummern wie „221b“.

Im Kapitel 5 werden wir sehen, ob diese Regeln zu einer besseren Erkennung beitragen.

## 4.3 Andere Design-Entscheidungen

Beim Entwurf eines IE-Systems müssen neben der Festlegung der Features weitere Entscheidungen getroffen werden. Sie sind nicht weniger wichtig als die Festlegung der Features, die im vorigen Abschnitt ausführlich diskutiert wurde. Eine Untersuchung der weiteren Parameter hätte jedoch den Rahmen dieser Arbeit gesprengt, zumal die vorhandenen Standardlösungen als hinreichend gut angesehen wurden. Einige weitere Parameter sowie die gewählten Einstellungen werden hier kurz beschrieben.

**Lexikalische Analyse** Der GATE-Tokeniser lässt sich flexibel konfigurieren. So gehören zum Lieferumgang zwei Versionen der Regelsätze: eine, die Bindestriche als Teile der Wörter betrachtet (Voreinstellung), und eine, die sie als separate Tokens ansieht. Im Regelfall ist es völlig ausreichend, den vorgegebenen Regelsatz zusammen mit den JAPE-Postprocessing-Regeln zu verwenden. In bestimmten Fällen kann es vorkommen, dass keiner der Regelsätze für die konkrete Aufgabe geeignet ist. In diesem Fall muss der Tokeniser mit neuen Regeln ausgestattet werden.

**Boundary Classification** Bei Annotationen, die mehrere Tokens umfassen, gibt es verschiedene Möglichkeiten, die Annotationsgrenzen zu kodieren. [LM09] verwendet das *Begin/End*-Schema zur Kodierung von langen Textabschnitten. Hierbei werden der erste und der letzte Token als jeweils Anfang und Ende eines längeren Textsegments markiert, die Tokens zwischen den beiden werden als negative Beispiele betrachtet. Die Auswahl eines passenden Schemas ist eine wichtige Entscheidung. Da die zu extrahierenden Felder in diesem Fall nur wenige Tokens umfassten, wurde das einfache *Inside/Outside*-Schema verwendet, d.h. jeder Token innerhalb der Annotation wurde als ein positives Beispiel markiert. Es gibt auch noch weitere Schemata wie *Begin/Continue/End*, *Begin/Continue/Outside* (ebenda).

**Lernalgorithmus** Jeder Algorithmus, der sich für das Klassifizierungsproblem eignet, könnte verwendet werden, z.B. Naive Bayes oder Entscheidungsbäume. Wie [TAC06] jedoch angibt: „it seems that hyperplane separators present some features that make them specially suitable for NLP tasks, for instance, their ability to deal with a large number of features“. [LM09] gibt weitere Vorteile von SVM als einem Hyperebenenklassifizierer an. Aus diesem Grunde wurde SVM als der primäre Algorithmus verwendet.

# 5 Experimente

## 5.1 Vorbemerkungen

Das Beraterprofil-Korpus besteht aus mehreren Tausend Profilen. Die Abbildung 5.1 zeigt das typische Aussehen eines tabellarischen Lebenslaufs. Im Grunde genommen hatte ich Zugriff auf alle Beraterprofile, die in den letzten Jahren bei PASS eingegangen waren. Allerdings waren diese Profile nicht annotiert. Diese Arbeit musste also von Hand erledigt werden. Zum Glück waren die meisten Profildaten bereits in die Datenbank eingepflegt worden, so dass man einen Teil der relevanten Daten per Textsuche finden konnte. Nichtsdestotrotz war die Datenaufbereitung ein mühsamer Prozess, mussten doch alle Trainingsdokumente durchgesehen und Annotationen korrigiert werden. Aus diesem Grunde wurden für die Namens- und Adressenerkennung Korpora von jeweils 50 und 45 Dokumenten verwendet. Wie die folgenden Abschnitte zeigen, war diese Zahl völlig ausreichend.

Bei der Extraktion von Namen wurden mehrere Typen von Lebensläufen identifiziert. Es wurde nur mit einem Typ gearbeitet. Bei einer vollständigen Lösung sollten weitere Typen einbezogen werden, damit für jeden ein separater Klassifizierer gelernt werden könnte. Diese Klassifizierer sollten dann mit einem geeigneten Verfahren miteinander verbunden werden.

Insgesamt wurden folgende Typen von Slots extrahiert:

- Name:
  - Vorname,
  - Nachname;
- Adresse:
  - Straße (inkl. Hausnummer),
  - Postleitzahl,
  - Ort.

Damit ließe sich die komplette Postadresse des Lebenslauf-Verfassers feststellen, um ihn z.B. eindeutig zu identifizieren. Die Slot-Aufteilung entspricht der Struktur des Datenbankschemas, das zur Speicherung der Daten verwendet wird. Wenn die Daten mehrmals in einem Dokument vorkamen, dann wurden sie auch immer annotiert. Das kann bspw. bei Kopf- und Fußzeilen der Fall sein. Um Overfitting und anderen Anomalien vorzubeugen, wurden bei solchen Do-

## Lebenslauf

**Persönliche Daten**

| | |
|---|---|
| **Name:** | Aleksandrs Galickis |
| **Geburtsdatum:** | 12.02.1986 |
| **Geburtsort:** | Musterstadt |
| **Staatsangehörigkeit:** | deutsch |
| **Wohnort:** | Musterstraße 1a<br>01234 Musterstadt |
| **Telefon:** | (0123) 12 34 56 78 |

**Bisheriger Bildungsweg**

| | |
|---|---|
| September 2008 – Januar 2009 | Auslandssemester an der Université Montesquieu Bordeaux IV, Bordeaux, Frankreich |
| Oktober 2006 | Beginn des Studiums der Wirtschaftsinformatik an der TU Darmstadt |
| Juni 2006 | Abitur am Heinrich-von-Gagern-Gymnasium, Musterstadt |

**Berufserfahrung**

| | |
|---|---|
| seit Oktober 2007 | Werkstudent bei PASS IT-Consulting, Aschaffenburg |
| Oktober 2007 – Januar 2008 | Übungstutor für die Veranstaltung „Grundlagen der Informatik I“ an der TU Darmstadt |
| August 2007 – September 2007 | Praktikum bei PASS IT-Consulting, Aschaffenburg |
| April 2009 – Juli 2009 | Übungstutor für die Veranstaltung „Einführung in die Wirtschaftsinformatik II (Programmierung in Java)“ an der TU Darmstadt |

Abbildung 5.1: Ein typischer tabellarischer Lebenslauf

kumenten nur die ersten ein paar Seiten (höchstens drei) beibehalten, der Rest wurde gelöscht. Eine andere, etwas kompliziertere Möglichkeit wäre gewesen, die relative Position zum Dokumentanfang als ein Feature zu erfassen, um dem Klassifizierer mitzuteilen, dass Tokens am Dokumentanfang vorzuziehen sind.

### 5.1.1 Einstellung der SVM-Parameter

Die Autoren der SVM-Bibliothek `libsvm` , die von MinorThird verwendet wird, haben eine Anleitung geschrieben, die SVM-Neulingen in Kochbuch-Manier erklärt, wie sie den SVM-Lerner konfigurieren, um gute Ergebnisse zu erzielen (s. [HCL03]). Diese Vorgehensweise kann benutzt werden, um das Ergebnis weiter zu verbessern (s.u.). Die grundlegenden Schritte sehen dabei wie folgt aus:

- Features skalieren, so dass der Wertebereich aller Features [-1, 1] bzw [0, 1] ist. Dieser Schritt ist hier nicht erforderlich, da alle verwendeten Features ohnehin den Wertebereich $\{0;1\}$ haben.
- Die Kernelfunktion auswählen. Die Autoren empfehlen den RBF-Kernel, der in dem meisten Situationen gute Dienste tut. Bei einer großen Featureanzahl – wie in unserem Fall – kann der lineare Kernel benutzt werden.
- Die beste Parameterauswahl mit Cross-Validation (vgl. Abschnitt 4.1) suchen. Im Falle des RBF-Kernels gibt es zwei Parameter: $\gamma$ und $C$; bei einer linearen Kernelfunktion ist nur der Parameter $C$ vorhanden.
- System auf dem ganzen Trainingssatz mit den optimalen Parametern trainieren.

Als Parameter-Einstellungen werden stets Voreinstellungen der Bibliothek ([CL01]) mit linearem Kernel verwendet.

### 5.1.2 Durchgeführte Experimente

Für beide Informationstypen wird eine ähnliche Reihe an Experimenten durchgeführt. Hier werden sie kurz und knapp beschrieben. Die Anzahl der Cross-Validation-Folds ist bei allen unten angegebenen Experimenten 5. Die Systemperformanz wurde in einem iterativen Prozess verbessert: Zuerst wurde die Fenstergröße optimiert, dann die Featuremenge, schließlich die SVM-Parameter. Um die Auswirkung einer einzelnen Veränderung abzuschätzen hätte der „ceteres paribus“-Ansatz gewählt werden können, d.h. alle Einstellung bis auf die zu testende in den Anfangszustand zurücksetzen. Beiden Vorgehensweisen liegt die Annahme zugrunde, dass bei Änderung eines Parameters sich die anderen nicht ändern. Dies ist streng genommen nicht der Fall, jedoch erleichtert diese Annahme die Parameteroptimierung.

**Auswirkung der Trainingssatzgröße** Ein Teil des Korpus wird als Testdatensatz verwendet, um die Auswirkung der Größe des Trainingssatzes auf die Erkennungsrate zu veranschaulichen. Die Ergebnisse dieses Experiments sind weniger genau als die, die bei Cross-Validation erzielt werden, weil die Datensätze zu klein sind; ferner wird das Experiment pro Konfiguration nur einmal durchgeführt. Trotzdem kann eine gewisse Tendenz aufgezeigt werden.

**Auswirkung der Fenstergröße** Es werden mehrere Fenstergrößen und -anordnungen ausprobiert (vgl. Abschnitt 3.2.1). Dabei wird der gesamte Korpus herangezogen. Die Auswertung erfolgt mittels Cross-Validation. Eine adäquate Fenstergröße ist wichtig, denn ein zu kleines Fenster führt zu schlechter Erkennungsqualität, während ein zu großes Overfitting verursacht.

**SVM-Parameteroptimierung** Im Abschnitt 5.1.1 wurde die Anpassung der Parameter beim SVM-Lerner beschrieben. Bei der Verwendung eines linearen Kernels gibt es nur einen Parameter, die „Error penalty rate“ $C > 0$. Es wird vom Voreinstellungswert 1 ausgegangen und mehrere Werte ausprobiert. Nach einer Empfehlung von [HCL03] werden exponentiell wachsende Folgen verwendet (z.B. $C = 2^{-5}, 2^{-3}, \ldots, 2^{15}$).

**Auswirkung der einzelnen Features** Nacheinander wird jedes einzelne Feature im Featuresatz deaktiviert, auch ganze Gruppen können deaktiviert werden. Anschließend wird die allgemeine Performanz des Systems angegeben (ebenfalls mittels Cross-Validation). So lässt sich beurteilen, welche Features tatsächlich zur Verbesserung der Erkennungsqualität beitragen, denn die Features wurden ja a priori festgelegt. Zur Unterstützung werden außerdem die Feature-Gewichte des trainierten Klassifizierers herangezogen.

Bei den Gewichten bedarf es einer Erklärung, wie diese zustande kommen. Die Darstellung der Binarisierungsproblematik folgt [Für02]. Viele Klassifizierungsalgorithmen wie auch SVM können nur Zwei-Klassen-Probleme (positive vs. negative Beispiele) lösen. Um Mehrklassenprobleme lösen zu können, kann man sie entweder verallgemeinern (was nicht einfach ist) oder das Mehrklassenproblem in mehrere Zweiklassenprobleme zerlegen. Die einfachste Strategie, *One-Against-All,* besteht darin, ein $n$-Klassen-Problem in $n$ binäre Probleme zu zerlegen, wobei in jedem Problem eine Klasse alle positiven Lerninstanzen darstellt und der Rest alle negativen. Bei der *paarweisen* bzw. *Round-Robin-Binarisierung* werden $n(n-1)/2$ binäre Probleme erzeugt, wobei jeweils nur zwei Klassen berücksichtigt werden und der Rest ignoriert wird. Die von MinorThird benutzte SVM-Implementierung `libsvm` benutzt ausschließlich die Binarisierung mittels Round-Robin-Verfahren.[1] Obwohl mehr Klassifizierer erzeugt werden, werden im Folgenden nur Gewichte der Klassifizierer angegeben, die Slot-Klassen mit `NONE` (wie in Abschnitt 1.2 definiert) vergleichen. Dies hat zwei Gründe. Erstens sind Vergleiche wie `VORNAME` gegen `NACHNAME` nicht so einfach interpretierbar: Was bedeutet das bei einer Instanz, deren korrekte Klasse eigentlich `NONE` ist? Deswegen sind diese Klassifizierer weniger aussagekräftig. Zweitens sind diese Gewichte nur von Bedeutung, wenn der Klassifizierer systematisch Substitutionsfehler macht (also Slots falsch, aber nicht als `NONE` klassifiziert). Dies war bei den beiden

Klassifizierern nicht der Fall, deshalb wurde auf die Betrachtung der verbleibenden Featuregewichte verzichtet.

## 5.2 Namen

### Trainingssatzgröße

In der Abbildung 5.2 sind die Precision und Recall eines Klassifizierers dargestellt. Die Größe des Trainingssatzes wurde variiert, während der Testdatensatz konstant 10 Dokumente enthielt. Man sieht, dass mit der Zeit die Werte gegen $P \approx 0,8$ und $R \approx 0,4$ konvergieren. Hierbei *wächst* der Recall, während die Precision *fällt*[2]. Dies ist jedoch keine tatsächliche Verschlechterung. Vielmehr lassen sich die Verläufe folgendermaßen erklären. Der Recall verbessert sich mit der Korpusgröße, weil dank der besseren Generalisierung mehr und richtiger erkannt wird. Die Precision verschlechtert sich anfangs, weil immer mehr extrahiert wird, darunter auch falsche Informationen, wohingegen am Anfang kein Fehler gemacht wurde, weil der Klassifizierer nichts extrahieren konnte.

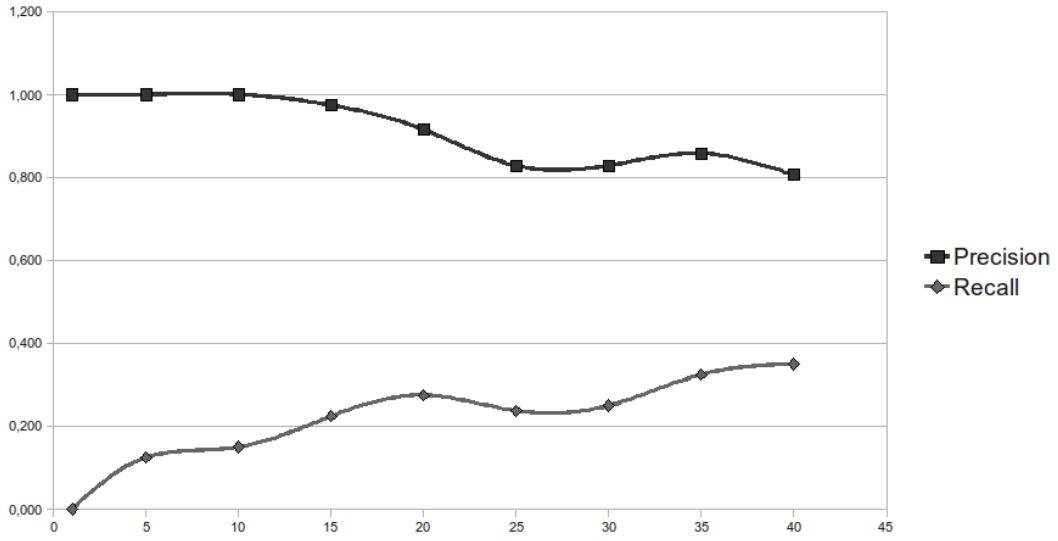


Abbildung 5.2: Auswirkung der Trainingssatzgröße auf die Erkennung von Namen

[1] Vgl. die FAQ-Seite `http://www.csie.ntu.edu.tw/~cjlin/libsvm/faq.html`, insbesondere die Antwort auf die Frage „What method does libsvm use for multi-class SVM?“ Hier wird „one against one“ genannt, was aber dasselbe wie Round-Robin ist.

[2] Bei der Größe 1 wurden keine Namen erkannt. An dieser Stelle wurde die Precision auf 1 gesetzt, damit die Kurve einen gleichmäßigen Verlauf annimmt. Es gibt Situationen, wo diese Sichtweise nicht sinnvoll und irreführend sein kann. Hier ist jedoch klar, was gemeint ist.

| von | nach | Vornamen | | Nachnamen | |
|---|---|---|---|---|---|
| | | Precision | Recall | Precision | Recall |
| -10 | 10 | 0,910 | 0,396 | 0,935 | 0,411 |
| -5 | 5 | 0,846 | 0,493 | 0,911 | 0,407 |
| -1 | 5 | 0,800 | 0,448 | 0,776 | 0,191 |
| -5 | 1 | 0,651 | 0,483 | 0,706 | 0,402 |
| -1 | 1 | 0,663 | 0,311 | 0,567 | 0,120 |
| -10 | 5 | 0,769 | 0,522 | 0,752 | 0,378 |

Tabelle 5.1: Auswirkung der Fenstergröße auf die Erkennung von Namen

### Fenstergröße

In der Tabelle 5.1 werden die Ergebnisse der Variation der Fenstergröße zusammengefasst. Hierbei wurden symmetrische wie asymmetrische Größen ausprobiert. Die Größe wurde dabei für alle Features gleichzeitig verändert. Aus der Tabelle lassen sich folgende Schlüsse ziehen:

- Größere Fenster führen zu einer Verbesserung der Precision, aber zu einer Verschlechterung des Recall.
- Asymmetrische Fenster: Bei Vornamen ist $[-1;5]$ eindeutig besser als $[-5;1]$, allerdings fällt im ersten Fall der Recall der Nachnamen auf unter 20% ab. Vermutlich liegt es daran, dass sich die Erkennung von Nachnamen viel mehr auf die Informationen über die Vorgänger-Tokens stützt (etwa bei den Signalwörtern oder der Tatsache, dass ein Vorname vom Nachnamen gefolgt wird). Ohne diese Informationen bricht die Qualität ein.
- Die Größe $[-10,10]$ scheint das beste Ergebnis zu haben. Hier gibt es jedoch noch ein paar Feinheiten zu beachten. Betrachten wir die Daten für Vornamen (bei Nachnamen waren Unterschiede eher gering). $\sigma_P$ war bei $\pm 5$ ungefähr 0,12, $\sigma_R \approx 0,06$. Nach der Ungleichung von Tschebyscheff liegen 75% der Werte im Intervall $[R-3\sigma_R; R+3\sigma_R]$ (vgl. [FKPT04, S. 329f.]). Mit anderen Worten waren die Werte je nach Fold recht unterschiedlich, so dass bei einer anderen Fold-Aufteilung ein ähnliches Ergebnis wie bei $\pm 10$ möglich gewesen wäre.

Trotz der obigen Ausführungen ist die Fenstergröße $\pm 10$ die beste Alternative. Diese wird also auch für weitere Experimente verwendet.

### Auswirkung der einzelnen Features

Die Auswirkungen der Deaktivierung jedes einzelnen Features sind in der Tabelle 5.2 dargestellt. Die Zeile „—“ wurde aus der Tabelle 5.1 übernommen und entspricht der Situation, dass alle Features aktiviert sind. Das wichtigste Feature ist offenbar `lc`, denn seine Eliminierung

| Abgeschaltetes Feature | Vornamen | | Nachnamen | |
|---|---|---|---|---|
| | Recall | Precision | Recall | Precision |
| — | 0,910 | 0,396 | 0,935 | 0,411 |
| lc | 0,754 | 0,330 | 0,200 | 0,025 |
| commonFirstname | 0,893 | 0,240 | 0,911 | 0,293 |
| commonLastname | 0,852 | 0,478 | 0,882 | 0,355 |
| keyword | 0,898 | 0,344 | 0,849 | 0,289 |
| isUpperInitial | 0,846 | 0,373 | 0,931 | 0,380 |
| *Wörterbuchbasierte* | 1,000 | 0,224 | 0,850 | 0,220 |

Tabelle 5.2: Auswirkung der einzelnen Features auf die Extraktion von Namen

hat den größten Effekt sowohl auf Precision als auch auf Recall. Insbesondere bei Nachnamen wird so eine zuverlässige Erkennung nicht möglich. Wichtig sind auch die wörterbuchbasierten Features `commonFirstname`, `commonLastname` und `keyword`. Das letztere scheint eine vergleichbare Auswirkung zu haben, obwohl die Wortliste weniger als 10 Einträge enthält (s. Fußnote 2 auf der Seite 30), die alle aus den Trainingsdokumenten manuell extrahiert wurden. Vermutlich wäre der Lernalgorithmus bei einem wesentlich größeren Korpus in der Lage gewesen, diese Terme alleine zu finden. Der Designer hat dem Algorithmus „auf die Sprünge geholfen“, indem er dank seinem Hintergrundwissen generalisiert hat, dass diese Signalwörter auch in anderen, ungesehen Lebensläufen recht häufig vorkommen werden.

Um die Auswirkung der wörterbuchbasierten Features noch eindrucksvoller darzustellen, wurden *alle* wörterbuchbasierten Features abgeschaltet. Während die Precision dabei immer noch hoch war, ist der Recall auf ca. 0,22 gefallen. Dagegen ist das Feature `isUpperInitial` weniger wichtig, denn seine Abschaltung hat keine nennenswerte Auswirkung auf das Ergebnis.

Es lohnt sich, einzelne Featuregewichte anzuschauen (s. Tabellen 5.3 und 5.4). Links stehen jeweils die Gewichte, die unter Verwendung der wörterbuchbasierten Features entstehen, rechts ohne, d.h. nur unter Verwendung von `lc` und `isUpperInitial`. Wenn kein Wörterbuch verfügbar ist, muss der Lernalgorithmus auf Signalwörter selber schließen, deswegen kommen im rechten Tabellenteil solche Wörter wie „Adresse“, „Anschrift“ u.a. vor. Links machen Wörterbucheinträge viel aus.

Wir stellt schnell fest, dass der Beitrag des Features `isUpperInitial` am geringsten ist, wovon sowohl die geringe Veränderung der Precision- und Recall-Werte bei Deaktivierung als auch sein geringes Gewicht bei den Featuregewichten zeugen. Im Weiteren kann es eliminiert werden.

| Mit Wörterbuch | | Ohne Wörterbuch | |
|---|---|---|---|
| Feature | Gewicht | Feature | Gewicht |
| commonFirstname(0) | 0,728 | lc(0).thomas | 0,473 |
| lc(4).1 | 0,426 | lc(-1).name | 0,446 |
| keyword(-3) | 0,377 | lc(-2).name | 0,428 |
| lc(-1).name | 0,334 | lc(0).stefan | 0,415 |
| lc(2).anschrift | 0,293 | lc(-1).herr | 0,398 |
| lc(-1).herr | 0,289 | lc(2).anschrift | 0,379 |
| lc(2).geburtsjahr | 0,278 | lc(0).michael | 0,378 |
| commonLastname(0) | 0,277 | lc(0).stephan | 0,335 |
| lc(0).thomas | 0,275 | lc(4).1 | 0,327 |
| keyword(-4) | 0,252 | isUpperInitial(0) | 0,317 |

Tabelle 5.3: Featuregewichte bei Vornamen mit und ohne wörterbuchbasierte Features

| Mit Wörterbuch | | Ohne Wörterbuch | |
|---|---|---|---|
| Feature | Gewicht | Feature | Gewicht |
| commonFirstname(-1) | 0,484 | lc(-2).name | 0,580 |
| lc(-2).name | 0,383 | lc(-2).herr | 0,402 |
| keyword(-2) | 0,342 | lc(1).anschrift | 0,379 |
| lc(1).anschrift | 0,315 | lc(-1).thomas | 0,365 |
| lc(-2).herr | 0,309 | lc(-3).name | 0,346 |
| keyword(-5) | 0,301 | lc(-1).stefan | 0,294 |
| lc(1)., | 0,300 | lc(1).adresse | 0,286 |
| commonFirstname(2) | 0,295 | lc(-9).berater | 0,276 |
| commonLastname(0) | 0,280 | lc(7).frankfurt | 0,274 |
| lc(-1).thomas | 0,277 | lc(-4).persönliche | 0,273 |

Tabelle 5.4: Featuregewichte bei Nachnamen

### SVM-Parameteroptimierung

Die Tabelle 5.5 fasst die Ergebnisse der Optimierung des SVM-Parameters *C* zusammen. Dabei wurden die Gesamtperformanz des Systems, also für Vor- und Nachnamen gemeinsam, ermittelt. Man sieht, dass in den positiven Zweierpotenzen (2, 4 etc.) ein leichter Anstieg des Recalls mit einer geringen Verkleinerung der Precision kompensiert wird. Bei der Zielsetzung „Precision-Maximierung“ mit Satisfizierungsziel „Recall mindestens 40%“ ist also ein Wert von ca. 0,5 zu wählen.

| ***C*** | 0,063 | 0,105 | 0,125 | 0,210 | 0,250 | 0,500 | 1,000 | 2,000 | 4,000 |
|---|---|---|---|---|---|---|---|---|---|
| **Precision** | 0,939 | 0,933 | 0,933 | 0,930 | 0,930 | 0,930 | 0,930 | 0,930 | 0,930 |
| **Recall** | 0,309 | 0,353 | 0,336 | 0,394 | 0,399 | 0,406 | 0,406 | 0,399 | 0,399 |

Tabelle 5.5: Auswirkung des Parameters *C* auf die Extraktion von Namen

## 5.3 Adressen

Die Adressextraktion geschah mittels eines neuen, von der Namensextraktion unabhängigen Klassifizierers. Die Erkennung war also völlig losgelöst von der Erkennung der Namen. Ferner wurde der Seed, der den Zufallsgenerator bei der Folderzeugung verwendet (s. Tabelle 3.1, S. 25, Einstellung `crossValidatorSeed`), konstant gehalten, so dass identische Folds erzeugt wurden, und die Ergebnisse untereinander vergleichbar waren.

### Trainingssatzgröße

Insgesamt wurden 45 Dokumente annotiert. Hiervon wurden fünf abgesondert und als Testdatensatz verwendet. Die Ergebnisse der Auswertung werden in der Abbildung 5.3 dargestellt. Man sieht hier einen ähnlichen Verlauf wie bei der Namenserkennung. Wie oben schon erwähnt reichen fünf Testdokumente nicht aus, um die Erkennungsgüte zu bewerten, deswegen spielt hier die Auswertung mittels Cross-Validation (s.u.) eine größere Rolle als eine Trennung in Trainigs- und Testsatz, wie sie hier vorgenommen wurde. Außerdem erkennt man, dass ein Trainingssatz von nur 15 Dokumenten in dem Fall völlig ausreichend war.

### Fenstergröße

Die Ergebnisse der Experimente mit den Fenstergrößen sind in der Tabelle 5.6 zusammengefasst. Hier wurden zwei unterschiedliche Konfigurationen getestet: einmal mit den JAPE-Regeln für Straßennamen und Postleitzahlen und einmal ohne. Man sieht, dass unter der Verwendung

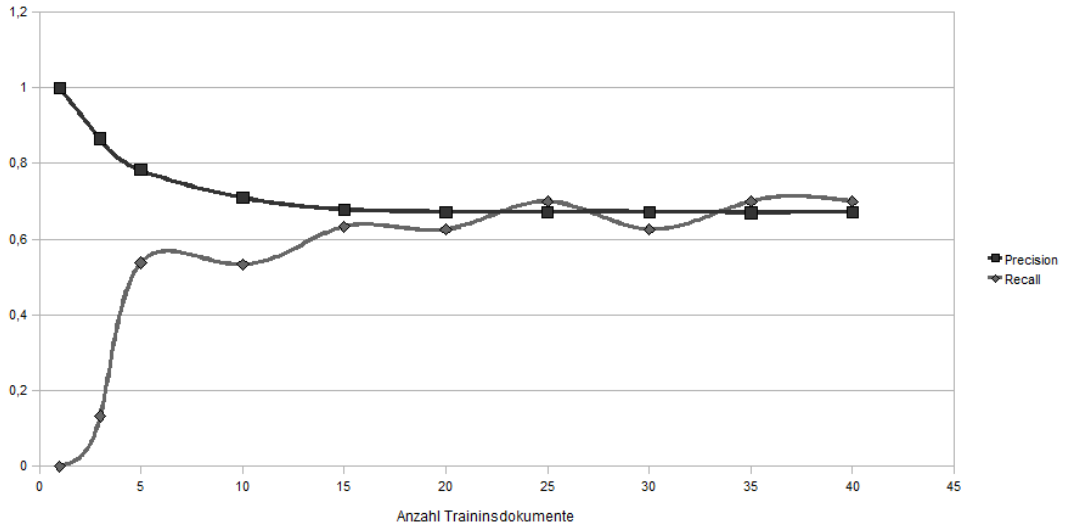


Abbildung 5.3: Precision und Recall der Adressenerkennung bei unterschiedlichen Trainingssatzgrößen

der JAPE-Regeln keine wesentliche Verbesserung der Erkennungsgüte durch Vergrößerung des Fensters erreicht werden kann. Für alle drei Fenstergrößen und Slottypen galt: $\sigma_P \geq 0,03$, so dass der Unterschied zwischen der Größe $\pm 5$ und $\pm 10$ vernachlässigt werden kann. Bei der Fenstergröße kann man also bei $\pm 5$ bleiben. Das Ergebnis ohne Verwendung der JAPE-Regeln ist dagegen sehr interessant: Der Recall fällt zwar drastisch auf teilweise unter 30%, aber die Precision bleibt bei einem ausreichend großen Fenster ebenso hoch wie im obigen Experiment. Daraus folgt, dass die Genauigkeit der Erkennung dank dem wortbasierten Feature `lc` ebenso hoch bleibt, aber weniger Adressen gefunden werden. Der Einsatz einer Konfiguration ohne JAPE-Regeln wäre also auch möglich, allerdings haben wir uns zum Ziel gesetzt, Recall über 40% zu halten (s. Abschnitt 4.1). Dies war hier nicht erfüllt, weswegen die JAPE-Regeln beibehalten wurden, zumal sie zu besseren Recall-Werten führen.

### Auswirkung einzelner Features

Bei dieser Experimentreihe wurden alle Features nacheinander deaktiviert. Tabelle 5.7 zeigt die Ergebnisse. Der Eintrag „—“ entspricht der Zeile $[-5;+5]$ in der Tabelle 5.6. Man erkennt, dass bei fast keinem der Features eine Deaktivierung zu einem drastischen Abfall von Precision oder Recall geführt hat. Nur bei `looksLikeZip` haben sich die Recall-Werte von Postleitzahlen und Städtenamen verschlechtert. Am kleinsten fällt die Veränderung beim Feature `isNumber` aus. Wie man jedoch an den Features mit den höchsten und niedrigsten Gewichten sehen kann (s. Tabellen 5.8, 5.9 sowie 5.10), landet dieses Feature unter den zehn wichtigsten sowohl bei den Städtenamen als auch bei den Postleitzahlen. Da `looksLikeStreet` und `looksLikeHouse`

| | von | nach | Straße | | PLZ | | Stadt | |
|---|---|---|---|---|---|---|---|---|
| | | | Precision | Recall | Precision | Recall | Precision | Recall |
| (a) | -1 | 1 | 0,954 | 0,766 | 0,888 | 0,933 | 0,805 | 0,817 |
| | -5 | 5 | 0,960 | 0,733 | 0,969 | 0,882 | 0,892 | 0,671 |
| | -10 | 10 | 0,959 | 0,718 | 0,982 | 0,828 | 0,919 | 0,682 |
| (b) | -1 | 1 | 0,735 | 0,179 | 1,000 | 0,359 | 0,874 | 0,397 |
| | -5 | 5 | 0,903 | 0,325 | 0,950 | 0,304 | 0,850 | 0,304 |
| | -10 | 10 | 0,947 | 0,313 | 0,950 | 0,273 | 0,950 | 0,271 |

Tabelle 5.6: Auswirkung der Fenstergröße auf die Extraktion von Adressen. (a) unter Verwendung von JAPE-Regeln; (b) ohne JAPE-Regeln.

| Abgeschaltetes Feature | Straße | | PLZ | | Stadt | |
|---|---|---|---|---|---|---|
| | Precision | Recall | Precision | Recall | Precision | Recall |
| — | 0,960 | 0,733 | 0,969 | 0,882 | 0,892 | 0,671 |
| lc | 0,864 | 0,662 | 0,937 | 0,908 | 0,867 | 0,799 |
| isNumber | 0,953 | 0,750 | 0,891 | 0,893 | 0,883 | 0,735 |
| looksLikeStreet | 0,956 | 0,712 | 0,888 | 0,925 | 0,883 | 0,735 |
| looksLikeHouseNumber | 0,930 | 0,697 | 0,862 | 0,904 | 0,870 | 0,746 |
| looksLikeZIP | 0,943 | 0,750 | 0,920 | 0,392 | 0,881 | 0,324 |
| *Straßenerkennung* | 0,882 | 0,500 | 0,840 | 0,969 | 0,905 | 0,810 |
| *Alle JAPE-Regeln* | 0,903 | 0,325 | 0,950 | 0,304 | 0,850 | 0,304 |

Tabelle 5.7: Auswirkung der einzelnen Features bei der Adressenerkennung

`Number` sich gegenseitig ergänzen und gemeinsam verwendet werden, wurden sie beide gleichzeitig deaktiviert (Zeile „Straßenerkennung“). Hier zeigt sich ein Abfall der beiden Werte, der allerdings wiederum nicht so stark ist, solange die JAPE-Erkennung von Postleitzahlen aktiv ist. Wird auch diese nicht durchgeführt, so zeigen sich Ergebnisse, die mit dem vorigen Abschnitt übereinstimmen (Zeile „Alle JAPE-Regeln“). Dies kann folgendermaßen erklärt werden. Alle drei Adress-Slots treten im Text nebeneinander auf (zuerst die Straße, dann – ohne Whitespaces zu beachten – die PLZ und schließlich die Stadt). Da jede JAPE-Regel an sich für gute Erkennung sorgt, wird sie vom Lernalgorithmus herangezogen, um die relative Position aller anderen Slots zu bestimmen. Das erkennt man gut an den Feature-Gewichten: Bei den drei Slot-Klassen sind alle drei `looksLike*`-Features zu finden – jeweils mit unterschiedlichen Indizes für die relative Position.

Man beachte ferner, dass es keine speziellen Features für die Erkennung der Stadt gibt, ob-

| Feature | Gewicht |
|---|---|
| looksLikeHouseNumber(0) | 0,9060 |
| looksLikeStreet(0) | 0,7210 |
| looksLikeHouseNumber(1) | 0,6974 |
| looksLikeZip(2) | 0,5605 |
| looksLikeStreet(-1) | 0,5564 |
| lc(1).weg | 0,4858 |
| looksLikeZip(3) | 0,4697 |
| lc(4).hamburg | 0,4078 |
| lc(-3).lebenslauf | 0,3522 |
| lc(5).köln | 0,3376 |
| lc(-2).hartmann | 0,3340 |

Tabelle 5.8: Features mit den höchsten Gewichten für die Straßenangabe

wohl sie oftmals richtig erkannt wird. Hierin liegt die Stärke der Kopplung eines regel- und eines ML-basierten Ansatzes. Der Algorithmus *lernt*, dass der Name der Stadt sich in der Nähe der Postleitzahl befindet. Beim Ausprobieren hat sich darüberhinaus herausgestellt, dass er aus Beispielen gelernt hat, dass etwa „am“ wie in „Frankfurt *am* Main“ ebenfalls zum Namen gehört – ein Detail, das bei einer handkodierten Regel hätte vergessen werden können.

Die Featuregewichte können weiteren Aufschluss vermitteln. Bei allen drei Slots sieht man, dass JAPE-basierte Features überwiegen. Das ist nicht verwunderlich, denn die definierten Regeln beschreiben Adressen bereits hinreichend genau.

Die meisten JAPE-Features findet man bei der Straßenangabe, denn sie besteht aus mehreren Tokens, die potenziell alle von der Regel korrekt erkannt werden können. Außerdem wird das Schlüsselwort „Lebenslauf“ erkannt, das sich in unmittelbarer Nähe zur Adresse befindet, falls sie – wie in einem tabellarischen Lebenslauf üblich – ganz oben unter dem Titel befindet.

Wie vorhergesagt bestimmt sich der Namen der Stadt durch die relative Position zur Straße und Postleitzahl. Eine wichtige Rolle spielen auch die Schlüsselwörter „Ort“, das davor kommt, und „Tel“, das nach der Adresse die Telefonnummer ankündigt. All diese Regelmäßigkeiten wurden vom Lernalgorithmus erkannt.

Bei der Postleitzahl werden neben den offensichtlich ins Spiel kommenden Features wie `looksLikeZip` auch `isNumber` verwendet. An 10. Stelle findet sich der Token „D-“ (wie in „D-60313 Frankfurt“), analog hierzu finden sich weiter unten das Paar `(lc.ch-(-1), 0,1713)` für schweizerische Postleitzahlen. Alle Features werden benötigt. Der einzig mögliche Kandidat für die Streichung wäre das Feature `isNumber`, doch es ist, wie oben gezeigt wurde, für die Erkennung von Postleitzahlen erforderlich.

Aufgrund der ohnehin sehr guten Erkennung der Adressen wurde auf eine weitere Optimierung der SVM-Parameter verzichtet. Einen nennenswerte Verbesserung hätte sie nicht mehr herbeigeführt.

| Feature | Gewicht |
|---|---|
| looksLikeZip(-1) | 0,9443 |
| looksLikeHouseNumber(-3) | 0,3840 |
| looksLikeHouseNumber(-5) | 0,3566 |
| lc(2).deutschland | 0,3020 |
| lc(-5).hahnstrasse | 0,3005 |
| lc(-2).60528 | 0,3005 |
| lc(-4).25 | 0,3005 |
| isNumber(-1) | 0,2930 |
| lc(-3).ort | 0,2889 |
| lc(1).tel | 0,2769 |
| lc(4).tätigkeitsbereich | 0,2710 |

Tabelle 5.9: Features mit den höchsten Gewichten für den Namen der Stadt

| Feature | Gewicht |
|---|---|
| looksLikeZip(0) | 0,9793 |
| looksLikeHouseNumber(-2) | 0,4224 |
| isNumber(0) | 0,3644 |
| looksLikeStreet(-2) | 0,2771 |
| looksLikeHouseNumber(-4) | 0,2434 |
| isNumber(-1) | 0,2274 |
| lc(2).telefon | 0,2132 |
| looksLikeStreet(-3) | 0,2117 |
| looksLikeHouseNumber(-1) | 0,1999 |
| lc(-1).d- | 0,1770 |

Tabelle 5.10: Features mit den höchsten Gewichten für die PLZ

## 5.4 Fazit

Es wurden mehrere Experimente durchgeführt, um die Erkennung der Namen und Adressen zu beurteilen und zu optimieren. Die Ergebnisse zeigen, dass beides durchaus möglich ist. Die Erkennung von Namen ist dabei schwieriger als die von Adressen, denn bei den letzteren lassen sich musterbasierte Regeln angeben, mit denen sie hinreichend genau erkannt werden können. Dies wäre sogar ganz ohne ML möglich gewesen. Dies ist bei Namen nicht möglich, weil sie sowohl im Bezug auf ihre Position im Dokument als auch im Bezug auf in ihnen auftretende Muster weniger Regelmäßigkeiten zeigen. Insofern ist die Erkennung von Namen, die trotz recht niedrigen Recalls immerhin eine hohe Precision aufweist, ein wahrscheinlich größerer Erfolg als die Erkennung von Adressen, die höhere Precision- und Recallwerte hat. Wegen der Ungenauigkeit bei der Erkennung kann das System natürlich nicht ohne menschliche Aufsicht verwendet werden. Ein Einsatz z.B. als Hilfe bei der manuellen Datenerfassung ist jedoch durchaus denkbar. Die Kopplung von handkodierten Regeln mit ML-Ansätzen ist aber nicht weniger interessant, denn damit ließen sich handgeschriebene Regeln auf ihre Gültigkeit prüfen und sogar auf weitere, nicht explizit abgedeckte Fälle erweitern.

Zusammenfassend können folgende Empfehlungen für die Einstellung der einzelnen Parameter ausgesprochen werden.

Für Namen:

- Fenstergröße: $\pm 10$
- Features: wie in Abschnitt 4.2.1, jedoch ohne `isUpperInitial`
- SVM-Parameter: linearer Kernel mit dem Parameter $C = 0,5$

Für Adressen:

- Fenstergröße: $\pm 5$
- Features: exakt wie in Abschnitt 4.2.2 beschrieben
- SVM-Parameter: linearer Kernel mit Voreinstellungen

# 6 Schlusswort und Ausblick

Im 1. Kapitel wurden Ziele für diese Arbeit formuliert. Das Hauptziel, ein IE-System zur Erkennung von Namen und Adressen aus Lebensläufen zu entwerfen, zu optimieren und zu evaluieren, wurde erreicht. Die Ergebnisse wurden im Kapitel 5 vorgestellt. In diesem Kapitel werden weitere Fragestellungen diskutiert.

In der Einführung wurde von einem „Kochrezept“ für den Entwurf eines IE-Systems gesprochen. Folgender Vorschlag erscheint hierbei vernünftig:

1. a-priori-Features festlegen;
2. relevante Performanzmaße festlegen;
3. stabile Korpusgröße herausfinden;
4. Variation der Parameter:
   - Algorithmus und seine Parameter,
   - Fenster-Größe,
   - Boundary Classification,
   - Relevante Features etc.

Dabei kann der Schritt 4 automatisiert werden, sofern die Performanzmaße und mögliche Features definiert sind. Alle anderen Schritte *müssen* von einem menschlichen Experten durchgeführt werden, da Computerprogramme hierzu nicht in der Lage sind. Insbesondere ist hier die Festlegung der Performanzmaße zu erwähnen, zumal sie auch weniger offensichtlich erscheint als andere Schritte. Sie ist jedoch nicht weniger wichtig, weil nur durch klar festgelegte Gütemaße die Qualität der Erkennung beurteilt werden kann. Eine Arbeitserleichterung stellen vorgefertigte Komponenten dar, um z.B. Dokumente vorzubereiten, Features zu extrahieren und Experimente durchzuführen. Weiter denkbar wären „Feature-Bibliotheken“, die fertige Featurebeschreibungen enthalten. Klassifikationen von Features wie in Abschnitt 4.2 vorgestellt machen bereits einen Schritt in diese Richtung.

Mit den im Internet frei verfügbaren Werkzeugen können durchaus praxisrelevante IE-Systeme erstellt werden. Hierbei sind solche allgemeinen Rahmenwerke wie GATE von großer Hilfe. GATE ist keineswegs perfekt, jedoch ist es seit über zehn Jahren in Entwicklung und ist dadurch entsprechend stabil. Die API ist gut dokumentiert und intuitiv aufgebaut. Dank der

Möglichkeit, Pipelines zu speichern, ist es nicht schwer die in GATE entworfenen Systeme in eigene Anwendungen zu integrieren. Vom Einsatz von MinorThird ist dagegen aus folgenden Gründen abzuraten:

- die API ist sehr umständlich und nicht intuitiv,
- sie ist außerdem schlecht dokumentiert,
- der Code ist unübersichtlich und teilweise fehlerhaft.

Im Nachhinein betrachtet hat sich der Einsatz von MinorThird also nicht gelohnt. Seine unbequeme, fehlerhafte API und der unlesbare „Spaghetti-Code“ machen jeden Mehrwert, den die Bibliothek gegenüber reinen Klassifizierungswerkzeugen wie WEKA bietet, völlig zunichte. Andererseits wäre im Bezug auf die Erkenennungsgüte bei einem möglichen Einsatz von WEKA statt MinorThird auch nicht viel gewonnen, denn WEKA verwendet dieselbe SVM-Bibliothek wie MinorThird.

Eine detaillierte und ausführliche Betrachtung aller Aspekte des Entwurfs eines IE-Systems hätte den Rahmen einer Bachelorarbeit gesprengt, deswegen mussten einige Einschränkungen vorgenommen werden, um die Arbeit durchzuführen (vgl. auch Abschnitt 4.3):

- Es wurden keine weiteren Schemata von Boundary Classifications betrachtet. Ferner wurde nur ein Windowing-Verfahren betrachtet.
- Außer SVM wurden keine weiteren Algorithmen betrachtet.
- Die ganze Arbeit widmet sich nur einem Typ von IE-Systemen, nämlich ML-basierten Systemen, die einen Klassifizierer lernen. Es gibt jedoch zahlreiche weitere Möglichkeiten, wie es in [TAC06] gezeigt wird.
- Eine wichtige Information, die ein Beraterprofil enthält, ist die Angabe seiner Projekterfahrungen. Um Projekterfahrungen zu extrahieren, müsste eine Textsegmentierung vorgenommen werden, die ganz andere Herausforderungen mit sich bringt, da große Textbereiche klassifiziert werden müssen. Zur Verwendung des Klassifizierungsansatzes bei der Textsegmentierung sei auf [LM09] verwiesen. [KAP09] berichtet vom erfolgreichen Einsatz einer anderen, nicht ML-basierten Herangehensweise, um solche Informationen aus Lebensläufen zu extrahieren.
- Auf eine linguistische Analyse der Texte, wie sie von den Modulen aus Abschnitt 2.1 durchgeführt werden kann, wurde verzichtet, weil es den tabellarischen Lebensläufen an grammatischer Struktur fehlt. Bei natürlichsprachlichen Texten (wie z.B. dem Text dieser Arbeit) wäre eine solche Analyse unumgänglich.

Eine weitere Anmerkung ist zu den Methoden in Kapitel 5 zu machen. Auch solche Methoden der ökonometrischen Regressionsanalyse wie die multiple lineare Regression können als ML-Methoden im Sinne von [Mit97] angesehen werden. Im Rahmen der linearen Regression wird versucht, eine reellwertige Funktion als lineare Kombination ihrer Parameter (in der Terminolo-

gie dieser Arbeit „Features“) zu lernen. Im Gegensatz zum hier verwendeten Ansatz gibt es eine Reihe von Diagnoseverfahren und statistischen Tests, die z.B. die Signifikanz eines Parameters ermitteln lassen (vgl. z.B. [FKPT04] oder jedes andere Statistik- oder Ökonometrie-Lehrbuch). Bei den Experimenten war es bei geringen Unterschieden in den Messwerten schwer zu beurteilen, ob bestimmte Features relevant waren. Eine Möglichkeit, dies schnell und zuverlässig – ähnlich einem statistischen Signifikanztest – zu entscheiden, wäre wünschenswert.

In dieser Arbeit wurde eine Methode zur Erstellung eines Informationsextraktions-Systems skizziert und die Leistungsfähigkeit des Ergebnisses vorgestellt. Die Experimente überzeugen, dass der Klassifizierungsansatz vielversprechend ist. Interessant wäre eine weitere Erforschung derjeniger Parameter, die hier nicht extensiv betrachtet werden konnten. Die Einführung standardisierter Komponenten und Vorgehensweisen könnte die Entwicklung solcher IE-Systeme in der Zukunft beschleunigen und vereinfachen.

# Literaturverzeichnis


[Abn91] Abney, S.: *Parsing by chunks*. Principle-based parsing, 1991. http://www.vinartus.net/spa/90e.pdf.

[CEE+04] Carstensen, Kai Uwe, Christian Ebert, Cornelia Endriss, Susanne Jekat, Ralf Klabunde und Hagen Langer (Herausgeber): *Computerlinguistik und Sprachtechnologie*. Spektrum Akademischer Verlag, 2. Auflage, 2004.

[CL01] Chang, Chih Chung and Chih Jen Lin: *LIBSVM: a library for support vector machines*, 2001. Software available at http://www.csie.ntu.edu.tw/~cjlin/libsvm.

[CMBT02] Cunningham, H., D. Maynard, K. Bontcheva, and V. Tablan: *GATE: A framework and graphical development environment for robust NLP tools and applications*. In *Proceedings of the 40th Anniversary Meeting of the Association for Computational Linguistics*, 2002.

[Coh04] Cohen, William W.: *Minorthird: Methods for identifying names and ontological relations in text using heuristics for inducing regularities from data*, 2004. http://minorthird.sourceforge.net.

[FKPT04] Fahrmeir, Ludwig, Rita Künstler, Iris Pigeot und Gerhard Tutz: *Statistik. Der Weg zur Datenanalyse*. Springer Verlag, 2004.

[Für02] Fürnkranz, J.: *Round robin classification*. The Journal of Machine Learning Research, 2:721–747, 2002.

[HCL03] Hsu, C.W., C.C. Chang, and C.J. Lin: *A practical guide to support vector classification*, 2003.

[Hob93] Hobbs, Jerry R.: *The generic information extraction system*. In *MUC5 '93: Proceedings of the 5th conference on Message understanding*, pages 87–91, Morristown, NJ, USA, 1993. Association for Computational Linguistics, ISBN 1-55860-336-0.

[KAP09] Kluegl, P., M. Atzmueller, and F. Puppe: *Meta-Level Information Extraction (Resubmission)*. In *Proc. of the LWA-2009 (KDML Track)*, 2009.

[Lau05] Laux, H.: *Entscheidungstheorie*. Springer Verlag, 2005.

[LM09] Loza Mencía, Eneldo: *Segmentation of legal documents*. In *Proceedings of the Twelfth International Conference on Artificial Intelligence and Law*, 2009.

[Mic07] Michael, Jörg: *40 000 Namen – Anredebestimmung anhand des Vornamens*. c't, 07(17):182, 2007. http://www.heise.de/ct/ftp/07/17/182/.

[Mit97] Mitchell, Tom M.: *Machine Learning*. McGraw-Hill, 1997.

[MKSW99] Makhoul, John, Francis Kubala, Richard Schwartz, and Ralph Weischedel: *Performance measures for information extraction*. In *Proceedings of DARPA Broadcast News Workshop*, pages 249–252, 1999.

[Sar08] Sarawagi, Sunita: *Information extraction*. Foundations and Trends® in Databases, 1(3):261–377, 2008.

[TAC06] Turmo, Jordi, Alicia Ageno, and Neus Català: *Adaptive information extraction*. ACM Comput. Surv., 38(2):1–47, 2006, ISSN 0360-0300.