Methoden der lexikalischen Nachkorrektur OCR-erfasster Dokumente

Christian M. Strohmaier

München 2004

Methoden der lexikalischen Nachkorrektur OCR-erfasster Dokumente

Christian M. Strohmaier

Inaugural-Dissertation
zur Erlangung des des Doktorgrades
der Philosophie an der Ludwig–Maximilians–Universität
München

vorgelegt von
Christian M. Strohmaier
aus Burghausen an der Salzach

München, den 16. August 2004

Erstgutachter: Prof. Dr. Klaus U. Schulz
Zweitgutachter: Prof. Dr. Franz Guenthner
Tag der mündlichen Prüfung: 4. Februar 2005

Thesen

Das maschinelle Lesen, d. h. die Umwandlung gedruckter Dokumente via Pixelrepräsentation in eine Symbolfolgen, erfolgt mit heute verfügbaren, kommerziellen OCR-Engines für viele Dokumentklassen fast schon fehlerfrei. Trotzdem gilt für die meisten OCR-Anwendungen die Devise, je weniger Fehler, desto besser. Beispielsweise kann ein falsch erkannter Name innerhalb eines Geschäftsbriefes in einem automatisierten System zur Eingangsspostverteilung unnötige Kosten durch Fehlzuordnungen o. ä. verursachen. Eine lexikalische Nachkorrektur hilft, verbleibende Fehler von OCR-Engines aufzuspüren, zu korrigieren oder auch mit einer interaktiven Korrektur zu beseitigen. Neben einer Realisierung als nachgelagerte, externe Komponente, kann eine lexikalische Nachkorrektur auch direkt in eine OCR-Engine integriert werden.

Meinen Beitrag zur lexikalischen Nachkorrektur habe ich in zehn Thesen untergliedert:

These $\mathfrak{T}_1$: Für eine Nachkorrektur von OCR-gelesenen Fachtexten können Lexika, die aus thematisch verwandten Web-Dokumenten stammen, gewinnbringend eingesetzt werden.

These $\mathfrak{T}_2$: Das Vokabular eines Fachtexts wird von großen Standardlexika unzureichend abgedeckt. Durch Textextraktion aus thematisch verwandten Web-Dokumenten lassen sich Lexika mit einer höheren Abdeckungsrate gewinnen. Zudem spiegeln die Frequenzinformationen aus diesen Web-Dokumenten die des Fachtexts besser wider als Frequenzinformationen aus Standardkorpora.

These $\mathfrak{T}_3$: Automatisierte Anfragen an Suchmaschinen bieten einen geeigneten Zugang zu den einschlägigen Web-Dokumenten eines Fachgebiets.

These $\mathfrak{T}_4$: Eine feingliedrige Fehlerklassifikation erlaubt die Lokalisierung der beiden Hauptfehlerquellen der webgestützten Nachkorrektur:

- falsche Freunde, d. h. Fehler, die unentdeckt bleiben, da sie lexikalisch sind
- unglückliche Korrekturen hin zu Orthographie- oder Flexions-Varianten

These $\mathfrak{T}_5$: Falsche Freunde werden durch eine Kombination mehrerer OCR-Engines deutlich vermindert.

These $\mathfrak{T}_6$: Mit einfachen Heuristiken wird ein unglücklicher Variantenaustausch der Nachkorrekturkomponente vermieden.

These $\mathfrak{T}_7$: Mit einer Vereinheitlichung zu Scores lassen sich diverse OCR-Nachkorrekturhilfen wie etwa Wort-Abstandsmaße, Frequenz- und Kontextinformationen kombinieren und zur Kandidaten- sowie Grenzbestimmung einsetzen.

These $\mathfrak{T}_8$: OCR-Nachkorrektur ist ein multidimensionales Parameteroptimierungsproblem, wie z. B. Auswahl der Scores, deren Kombination und Gewichtung, Grenzbestimmung oder Lexikonauswahl. Eine graphische Oberfläche eignet sich für eine Untersuchung der Parameter und deren Adjustierung auf Trainingsdaten.

These $\mathfrak{T}_9$: Die Software zur Parameteroptimierung der Nachkorrektur der Resultate einer OCR-Engine kann für die Kombination mehrerer OCR-Engines wiederverwendet werden, indem die Einzelresultate der Engines wieder zu Scores vereinheitlicht werden.

These $\mathfrak{T}_{10}$: Eine Wort-zu-Wort-Alignierung, wie sie für die Groundtruth-Erstellung und die Kombination von OCR-Engines notwendig ist, kann durch eine Verallgemeinerung des Levenshtein-Abstands auf Wortebene effizient realisiert werden.

Kapitel 1

Einleitung

Das Ziel einer lexikalischen Nachkorrektur ist das Aufspüren und Ausbessern fehlerhafter **Wörter** mit Hilfe von **Lexika**. In der vorliegenden Arbeit werden bei der Nachkorrektur in erster Linie falsch erkannte Wörter von OCR-Engines betrachtet; die Aufgabenstellung ist aber i. Allg. übertragbar auf andere Fehlerquellen in Texten, z. B. orthographische Fehler, Tippfehler oder Fehler maschinell erstellter Protokolle gesprochener Sprache. Generell liegt die Erkennungsqualität aktueller, kommerzieller OCR-Engines heute auf einem hohen Niveau. Trotzdem enthalten OCR-Leseresultate immer noch einen nicht vernachlässigbaren Fehleranteil, selbst wenn die Dokumente vermeintlich optimale Eigenschaften für die OCR-Engine aufweisen. Das heißt sauber gedruckte, unverschmutzte, einspaltige Dokumente mit schwarzer Schrift auf weißem Hintergrund, ohne weitere Farben, ohne Bilder, gesetzt in einem gewöhnlichen Font wie Times New Roman in einer gewöhnlichen Größe ohne Ligaturen, etc. Sobald ein Stolperstein wie etwa Verschmutzung, kleine Schriftgröße, Fontwechsel oder kursiver Schriftschnitt im Dokument auftaucht, steigt oft die OCR-Fehleranzahl in signifikanter Weise an. In diesen Fällen hilft eine lexikalische Nachkorrektur, um ein qualitativ hochwertiges Endresultat zu erzielen. Aus pragmatischen Gründen ist in meiner Arbeit die Nachkorrektur durchwegs als externe Komponente dargestellt, die in einer Prozesskette der Dokumentreproduktion nach dem Lesevorgang der OCR-Engine folgt. Die vorgestellten Techniken der automatischen Nachkorrektur lassen sich aber auch direkt in eine OCR-Engine integrieren. Der Themenkomplex der lexikalischen Nachkorrektur wurde bereits in einer Reihe wissenschaftlicher Arbeiten untersucht [45, 88, 28, 13, 47]. Das Korrekturmodell folgt in etwa immer folgendem Prinzip: Alle Wörter werden in einem Lexikon (bzw. mehreren Lexika) nachgeschlagen. Wird ein Wort nicht gefunden, werden nahe liegende Verbesserungsvorschläge ermittelt. Eine weitere Software-Komponente bestimmt den Umgang mit den Verbesserungsvorschlägen, wobei automatische und interaktive Nachkorrekturmodelle unterschieden werden. In den meisten Fällen scheitert eine naive Vorgehensweise für beide Arten von Modellen. Alle Korrekturvorschläge ohne weitere Vorkehrungen automatisch umzusetzen endet in der Regel mit einer schlechteren Fehlerbilanz als beim Aus-

gangspunkt, da Abdeckungsdefizite der Lexika zum Austausch korrekt erkannter Wörter führen. Die interaktive Alternative, einen Benutzer mit allen Korrekturvorschlägen zu konfrontieren, führt in der Regel zu einem nicht zu bewältigenden Inspektionsaufwand. In meiner Arbeit wird daher die lexikalische Nachkorrektur als Auswahl- und Optimierungsproblem vieler Parameter betrachtet. Ich habe eine Software entwickelt, die hilft, folgende Fragen zu beantworten.

- Welche Lexika sollen zur Nachkorrektur herangezogen werden?
- Mit Hilfe welcher Wortabstandsmaße sind Verbesserungsvorschläge zu bestimmen?
- Wie viele Verbesserungsvorschläge sind sinnvoll?
- Welche Rolle spielt Frequenzinformation in der Vorschlagsliste?
- Wie kann kontextuelle Information in die Vorschlagsliste mit einfließen?
- Wie sicher ist sich die lexikalische Nachkorrektur bzgl. einzelner Vorschläge?
- Wie kann der Einsatz weiterer OCR-Engines die Qualität aufbessern?

Diese und noch einige andere Parameter können gemeinsam, systematisch und visuell gestützt untersucht werden. Es wurde eine offene Systemarchitektur entwickelt, die erlaubt, auch neue Parameter miteinzubeziehen.
Mit meiner Software untersuche ich v. a. die zentrale These $\mathfrak{T}_1$ meiner Arbeit, das Web für die Nachkorrektur nutzbar zu machen.

Die Arbeit ist folgendermaßen gegliedert. Die Thesen werden auf einem Korpus von Fachtexten verschiedener Themengebiete überprüft, das sich aus einem Groundtruth-Korpus und einem OCR-Korpus zusammensetzt. Das Groundtruth-Korpus, d. h. die Originaldokumente stammen ursprünglich aus dem Web, und deren Text liegt daher in elektronischer Form vor. Aus den HTML-Dokumenten wurde via Anzeige in einem Web-Browser, Ausdruck auf Papier, Scannen, OCR-Bearbeitung und Alignierung das parallele OCR-Korpus erzeugt. Eine nähere Beschreibung beider Korpusteile ist im nachfolgenden Kapitel enthalten. Im Kapitel 3 stelle ich die Architektur meiner Nachkorrektur-Software vor. Da eine Nachkorrektur im Document-Engineering nur einen Verarbeitungsschritt in einer Prozesskette darstellt, wird zuerst mit Blick von außen an Hand einer Schnittstellenspezifikation gezeigt, wie die Software zwischen OCR-Engine und weiteren Verarbeitungsschritten platziert werden kann. Anschließend wird mit Blick von innen der interne Aufbau in zwei Komponenten gezeigt. Die erste Komponente erzeugt zu dem OCR-gelesenen Text zusammen mit den Lexika Korrekturfiles, die alle erdenklichen Korrekturvorschläge enthalten. Eine unkontrollierte Ausführung aller Vorschläge würde allerdings mehr schaden als nutzen. Diese Beobachtung motiviert eine weitere Komponente zur Optimierung der in $\mathfrak{T}_8$ angeführten Parameter. Ein Benutzer kann mit Hilfe einer graphischen Oberfläche die Einflussparameter auf Trainingsmaterial adjustieren und für den

Produktionsbetrieb übernehmen. Kapitel 4 behandelt die für die Nachkorrektur eingesetzten statischen sowie dynamischen Lexika. Im Zentrum stehen dabei die Techniken zum dynamischen Aufbau von Lexika aus Web-Dokumenten. Die Basisidee dazu ist eine Extraktion einschlägiger Fachtermini aus den OCR-Texten mit Hilfe von Frequenzlisten, eine Anfrage dieser Fachbegriffe an Suchmaschinen, Download der Ergebnismenge und automatische Extraktion eines Lexikons aus diesen Dokumenten. Ausserdem werden die in $\mathfrak{T}_6$ angeführten Heuristiken beschrieben, die fehlende Varianten im Lexikon in der Nachkorrektur berücksichtigen. Die in $\mathfrak{T}_7$ proklamierte Vereinheitlichung diverser OCR-Nachkorrekturhilfen – das sind in erster Linie String-Abstandsmaße, Frequenz-, Kollokations- und Kookkurenzinformationen – wird im Kapitel 5 vorgeführt. Dazu werden die numerischen Zusatzinformationen zur Kandidatenkür auf das Intervall [0; 1] zu Scores normiert und anschließend linear kombiniert. Um die in $\mathfrak{T}_4$ genannten Schwächen einer automatischen Nachkorrektur besser aufzudecken zu können, werden die beobachteten Fehler in ein detailliertes Klassifikationsschema eingeordnet. Der Fehlerklassifikation habe ich ein eigenes Kapitel gewidmet. Am Ende des Kapitels 6 steht ein Vergleich mit anderen Klassifikationsschemata aus dem Bereich der OCR-Nachkorrektur. Im Kapitel 7 findet sich die Umsetzung von $\mathfrak{T}_9$. Es werden zwei Alternativen einer Wiederverwendung der in Kapitel 3 vorgestellten Software zur Kombination von OCR-Engines gegenübergestellt. Der erste Ansatz basiert auf einem Recycling der Ergebnisse der Nachkorrektur einzelner OCR-Engines als Scores; im zweiten Ansatz werden aus weiteren OCR-Leseresultaten Korrekturlexika gebildet. Zusätzlich wird am Ende von Kapitel 7 eine Erweiterung der Fehlerklassifikation aus Kapitel 6 für die Kombination mehrerer OCR-Engines präsentiert. Die Kapitel 8 und 9 umfassen Hilfstechniken, die an verschiedenen Stellen der Arbeit eingesetzt werden. Sowohl die Verknüpfung der Groundtruth mit dem OCR-Leseresultat, als auch die Kombination von OCR-Engines erfordern eine Alignierung. Dazu wird der in $\mathfrak{T}_{10}$ propagierte Algorithmus, der auf ein dynamisches Programmierschema aufbaut, im Vergleich zu einem kontextbasierten Ansatz vorgestellt. Nicht nur für den in $\mathfrak{T}_3$ angeführten Lexikonaufbau aus inhaltsverwandten Web-Seiten, sondern z. B. auch für eine Kandidatendesambiguierung mit Hilfe von Kookkurenzen bzw. Kollokationen im Web werden Suchmaschinen eingesetzt. Im Kapitel 9 zeige ich zwei Methoden, um automatisierte Anfragen an eine Suchmaschine zu stellen, mittels Webservice und mittels Wrapper. Die Arbeit schließt im Kapitel 10 mit einer Evaluation der Thesen an Hand mehrerer Experimente. Zuerst wird die Eignung dynamisch generierter Web-Lexika zur Nachkorrektur demonstriert, anschließend der Nutzen einer Optimierung der Korrekturgrenze und schließlich der Vorteil einer Kombination von OCR-Engines. Die Evaluation umfasst sowohl automatische als auch interaktive Ansätze.

Kapitel 2

Korpus

Das in meiner Arbeit verwendete Textkorpus setzt sich aus einem Groundtruth-Korpus und einem OCR-Korpus zusammen. Das Groundtruth-Korpus enthält den tatsächlichen, textuellen Inhalt der einzelnen Dokumente, das OCR-Korpus die zugehörigen Leseresultate von OCR-Engines.

2.1 Groundtruth-Korpus

2.1.1 Überblick

Der Begriff **Groundtruth** wird in der Dokumentenanalyse und den Geowissenschaften gleichermaßen verwendet, wobei sich die metaphorische Bedeutung an die geologischen Messungen an der Bodenfläche anlehnt. Die exakte Nahinspektion der Bodenfläche dient der Bewertung entfernter Messungen wie etwa Satellitenfernerkundungssystemen. Übertragen auf die Dokumentenanalyse ist die Bodenfläche das Dokument. Es werden möglichst viele Details eines Dokumentes genau (d. h. in der Regel manuell) erfasst, um die Leistungsfähigkeit von (automatisierten) Dokumentenanalysesystemen bewerten zu können. Erfasst werden z. B. der textuelle Inhalt, sog. *bounding boxes* von Zeichen, Wörtern, Zeilen und Spalten, Fonttypen, etc. Schon die Erstellung der Groundtruth ist keine reine, wissenschaftliche Fleißarbeit, sondern schult den Akademiker:

- Man macht sich mit dem Untersuchungsgegenstand vertraut und lernt die wichtigen Einflussfaktoren kennen. Auswirkung der Druckqualität, Scanner-Auflösung, Schriftgröße, Schriftschnitt, Sprache , Fachvokabular, Qualität verschiedener OCR-Engines, etc.

- Man überblickt die Anwendungsdomäne und kann daraus geeignete Klassifizierungen für spätere Beobachtungen entwickeln.

Die Anwendungsmöglichkeiten der Groundtruth lassen sich in zwei Stufen unterteilen:

- **Qualitativer Vergleich**. Ziel eines qualitativen Vergleichs einer Dokumentenanalyse ist, auf einen Blick zu erkennen, ob der Prozess in gewünschter Güte durchlaufen wurde. Dazu eignen sich besondere Visualisierungen, beispielsweise nach einer OCR-Texterkennung mit Groundtruth-Abgleich, nicht das Dokument selbst darstellen, sondern lediglich an der Position falsch erkannter Wörter einen roten Punkt setzen. Damit können problematische Regionen sofort lokalisiert werden. Ein weiteres Visualisierungsbeispiel ist der in [38] vorgestellte *accuracy scatter plot*.

- **Quantitativer Vergleich**. Ziel eines quantitativen Vergleichs einer Dokumentenanalyse ist, den Groundtruth-Abgleich in Form einer Maßzahl –noch besser ist eine Zergliederung in eine Reihe von Maßzahlen – zu präsentieren. Damit lassen sich verschiedene Verfahren direkt vergleichen. In meinem Fall sind das OCR versus OCR plus lexikalische Nachkorrektur. Aus einem ökonomischen Blickwinkel lässt sich mit Hilfe der Maßzahl(en) auch eine untere Grenze der gewünschten Güte vorgeben und anschließend eine Kostenminimierung durchführen.

Ein **Korpus** im Sinne moderner Linguistik bezeichnen McEnery und Wilson in [53] (Seite 21) eine Textsammlung, die folgende Eigenschaften erfüllt:

- Standard-Referenz
- Stichprobencharakter und Repräsentativität
- endliche Größe
- maschinenlesbare Form

Neben besserer Vergleichbarkeit wissenschaftlicher Arbeiten spricht auch die Kostenfrage als ganz pragmatischer Grund für den Rückgriff auf vorhandene Standard-Korpora. Die Erstellung von Groundtruth-Daten ist extrem zeitaufwändig, da nicht alle Schritte automatisierbar sind und daher sehr teuer. Beispielsweise die Erfassung des textuellen Inhalts eines Dokuments, das nur in gedruckter Form vorliegt, erfordert entweder Abtippen oder manuelle Nachkorrektur eines OCR-Laufs. Allerdings ist nur ein Teil, der in Publikationen erwähnten Korpora frei zugänglich. Ausserdem spiegeln die wenigen, vorhandenen Korpora die enorme Bandbreite an verschiedenen Dokumentklassen und Fragestellungen der Forschung wider, z. B. [80] ist eine Sammlung handschriftlicher, arabischer, historischer, medizinischer Dokumente und [70] ist eine Dokumentensammlung, die speziell zur Untersuchung von Segmentierung komplexer Layouts zusammengestellt wurde. Das am ehesten für die Prüfung meiner Thesen geeignete Korpus, wäre das TREC-5 Confusion Track gewesen ([37], [59]). Dort wäre die Groundtruth zusammen mit dem Ergebnis zweier OCR-Läufen schon vorhanden gewesen. Allerdings enthalten diese Texte zu einseitiges und zu wenig fachspezifisches Vokabular, da sie allesamt aus innerbehördlichen Berichten der US-Verwaltung stammen. Außerdem sind die (künstlich erzeugten) Fehlerraten von 5% und 20% für heute verfügbare OCR-Engines unrealistisch

hoch und das Korpus enthält nur englischsprachige Dokumente. Da kein geeignetes Groundtruth-Korpus gefunden wurde, ist die Entscheidung gefallen, ein eigenes zu erstellen. Eine Bereitstellung meines Korpus an die Forschergemeinde scheitert (vorerst) an juristischer Unsicherheit, wie vermutlich auch bei anderen unveröffentlichten Korpora. Bei der Verwendung fremder Dokumente müssen die Rechte anderer Personen an diesen Dokumenten beachtet werden. Allerdings gestaltet sich die Ableitung einer Grenze des Erlaubten aus diesem einleuchtenden Grundsatz schwierig: die Begriffe Copyright und Urheberrecht (besonders bzgl. elektronischer Dokumente im Web) werden derzeit unter Juristen kontrovers diskutiert, die Verwendung fremder Dokumente zur Generierung von Groundtruth-Daten für wissenschaftliche Arbeiten – v. a. eine Weiterveröffentlichung der Dokumente in diesem Rahmen – ist meines Wissens juristisches Neuland, und juristische Gutachten sind teuer.
Umgeht man diese juristische Problematik, indem man nur eigene Dokumente verwendet oder *ein* frei verfügbares Dokument (z. B. in [15] wird ausschließlich ein Roman verwendet), gerät man in die nächste Problematik: man erhält eine extreme Klumpenstichprobe, d. h. im allgemeinen gleiche Sprache, gleiches Vokabular, gleiches Layout, etc. Allerdings muss man sich vor Augen führen, dass es eigentlich unmöglich ist, eine repräsentative Stichprobe *aller* Dokumente zu ziehen. Das Spektrum ist zu groß, reicht von antiken Grabsteinen bis hin zu Bedienungsanleitungen technischer Geräte. Wichtig ist daher, sich eine begrenzte Domäne vorzugeben und daraus eine möglichst repräsentative Stichprobe zu ziehen. Damit lassen sich Einflussparameter bestimmen und Aussagen über die Domäne treffen. Im nachfolgenden Abschnitt ist dargestellt, wie ich eine Stichprobe meiner Domäne „Fachtexte“ ziehe.
Anders als in anderen Disziplinen der Korpuslinguistik besteht auf Grund der enormen Kosten für die Groundtruth-Erstellung keine Gefahr, das Korpus in seiner Größe nicht zu beschränken.
Um die Groundtruth in maschinenlesbarer Form zu erhalten, können Dokumente abgetippt werden oder OCR-Läufe manuell nachkorrigiert werden. Beides ist sehr fehleranfällig und teuer. Eine Alternative ist, nur Dokumente in das Korpus aufzunehmen, die bereits in elektronischer Form vorliegen und einen direkten Zugriff auf den Text erlauben. Texte ohne formales würden dieser Anforderung entsprechen, entsprechen aber bzgl. ihrer graphischen Repräsentation einer Klumpenstichprobe, d. h. druckt man solche Texte einfach aus, haben alle Texte das gleiche Druckbild. Es bleibt die Wahl, die Einflussparameter wie Font, Schriftgröße oder Hintergrund selbst zu variieren, oder reale Dokumente eines prozeduralen Formats zu verwenden.

Gewinnung von Groundtruth-Daten ist zu einem eigenen Forschungsbereich angewachsen. In [40] wird ein interaktiver Groundtruth-Editor vorgestellt und auf drei weitere Forschungsentwicklungen dieser Art verwiesen. Groundtruth-Erzeugung aus realen Dokumenten hat eine Reihe von Nachteilen: hohe Kosten, Copyright-Problematik und in [39] wird zudem berichtet, dass in manchen Bereichen die manuelle Erstellung von Groundtruth-Daten zu ungenau ist, z. B. bei der Vermessung von *bounding boxes*. Um diese Nachteile zu umgehen,

wird in diesem Forschungsbereich auch die Gewinnung von Groundtruth-Daten aus künstlich erzeugten Dokumenten untersucht (vgl. dazu [39], [33] und [51]); zum Teil werden mit Zufallsalgorithmen aus vorhandenen, realen Dokumenten neue, künstliche Dokumente erzeugt ([60] oder [87]). Aber der Einsatz synthetischer Dokumente hat auch eklatante Nachteile:

- Da man sich selbst die Hindernisse vorgibt, die man dann überspringen will, ist diese Vorgehensweise nur in Wissenschaftsgebieten sinnvoll, wo man schon die Einflussparameter sehr gut kennt und daher nicht mit Neuentdeckungen rechnet, sondern an Performance-Optimierungen, o. ä. arbeitet.

- Ausserdem können sog. Artefakte entstehen, ein Ergebnis einer Beobachtungsreihe, das nicht dem tatsächlichen Sachverhalt zuzuordnen ist, sondern vielmehr Produkt der eingesetzten Methode ist. Da man von Misserfolgen in der Wissenschaft eher aus Erzählungen, als aus Publikationen erfährt, bleibt folgendes Beispiel ohne Quellenangabe: eine Forschergruppe hat bei der Evaluation einer kommerziellen OCR-Engine die Erkennungsrate an Hand von künstlich erzeugten Zufallstrings beobachtet und damit katastrophal schlechte Werte ermittelt. Jedoch haben sich die Werte als unübertragbar auf reale Dokumente erwiesen. Wie sich erst später herausstellte, enthielt die zuerst als Blackbox betrachtete OCR-Engine einen simplen, auf englische Wörter abgestimmten Trigramm-Nachkorrekturmechanismus.

2.1.2 Erstellung

Es folgt eine Liste von pragmatischen Einschränkungen der Domäne, die eine Fokussierung auf die lexikalische Nachkorrektur ermöglichen:

- **Spracheinschränkung**. Da Lexika sprachabhängig sind, liegt eine Einschränkung nahe. Es wurden die Sprachen Deutsch und Englisch gewählt. Mit dieser Einschränkung ist jedoch nicht ausgeschlossen, dass ein deutscher oder englischer Fachtext auch nicht allgemeingebräuchliche Wörter oder Phrasen aus anderen Sprachen enthält. Mit Einschränkung der Sprache ergibt sich automatisch auch eine Einschränkung der Alphabete. Ich betrachte in dieser Arbeit nur lateinstämmige Alphabete.

- **Layout-Beschränkung**. Probleme, die bei einer Rekonstruktion eines komplex gestalteten Dokuments auftreten, wie etwa Segmentierung und Bestimmung der Lesereihenfolge, Text- und Bildtrennung, Tabellenerkennung, etc. erschweren auch die Groundtruth-Erstellung (vgl. dazu [32]). Daher wurden für das Korpus nur einspaltige Dokumente ohne besondere Layout-Raffinessen verwendet.

- **Symbolbeschränkung**. Am Beispiel von Firmen-Logos sieht man den fließenden Übergang zwischen Schrift und Symbol. Weitere Beispiele sind

Römisches Reich	Mittelalterliche Geschichte	Bauernkriege
Holocaust	Postmoderne Philosophie	Meteorologie
Mykologie, Botanik	Neurologie, Medizin	Technische Informatik
Versicherungswesen	Fische, Angeln	Jagd
Kochen	Oper	Speisepilze

Tabelle 2.1: Themengebiete des Korpus.

Musiknotationen, mathematische oder chemische Formeln, etc. Der fließende Übergang macht es auch schwierig zu entscheiden, welche Dokumente nicht in das Korpus aufgenommen werden sollen. Da die Korpuserstellung ohnehin semi-automatisch erfolgte, wurde per Augenschein entschieden, ob ein Dokument nicht erkennbare Symbole enthält.

- **Quellenbeschränkung**. Um eine manuelle Nachbearbeitung der Korpustexte zu vermeiden, habe ich Dokumente gewählt, die bereits in elektronischer Form verfügbar sind. HTML-Seiten aus dem Web – einer äußerst umfangreichen Dokumentensammlung – entsprechen in etwa den beiden Anforderungen Textzugriff[1] und reale Visualisierung[2]. Mit dieser Quellenbeschränkung schließt man auch weite Dokumentklassen wie etwa Handschriften oder historische Dokumente aus.

Um den enormen Arbeitsaufwand der Groundtruth-Erstellung im Zaum zu halten, wurde entschieden, vorerst ausschließlich den textuellen Inhalt und ein gescanntes TIFF-Bild elektronisch zu erfassen. Damit kann man derzeit automatisiert OCR-Fehler lokalisieren und das Verhalten einer lexikalischen Nachkorrektur beobachten, aber keine weiteren Rückschlüsse auf visuelle Ursachen ziehen, o.ä. Die verwendeten Dokumente werden nicht öffentlich zugänglich gemacht, um juristische Auseinandersetzungen zu vermeiden. Um eine möglichst repräsentative Stichprobe der Domäne „Fachtexte“ zu sammeln, wurde aus den drei Hauptthemengebieten Geschichte, Wissenschaft und Vermischtes eine Liste von 15 spezialisierten Themengebieten zusammengestellt.

Zu jedem dieser Themen wurden charakteristische Begriffe gewählt und mit Hilfe der Suchmaschine AllTheWeb englische und deutsche HTML-Seiten recherchiert. Aus der Rückgabemenge wurden unter Berücksichtigung der o. g. Beschränkungen des Layouts geeignete Seiten ausgewählt [3]. Der Web-Browser Opera wurde als Rendering-Engine der Web-Seiten benutzt. Der Browser bietet eine Option an, Dokumente in eine PostScript-Datei zu drucken. Über diesen Umweg wurden die HTML-Seiten zu Papier gebracht. Der Ausdruck erfolgte auf einem HP-Laserdrucker Modell HP LaserJet 4550 mit 300 DPI. Pro Themengebiet und Sprache wurden 20 Seiten gewählt, d.h. insgesamt 600 Seiten. Da

[1] mit dem UNIX-Tool html2text [79]

[2] da HTML z. T. auch deskriptive Elemente besitzt, sind einige lokal voreingestellte Fonts überrepräsentiert

[3] an dieser Stelle wurden auch Seiten ausgeschlossen, die offensichtlich nicht zum gewählten Themengebiet gehören

frisch gedruckte Dokumente nicht unbedingt von repräsentativer Qualität für denkbare OCR-Anwendungen sind, wurden die Ausdrucke noch 1x kopiert. Die TIFF-Dateien – der visuelle Teil der Groundtruth – wurden schließlich durch Einlesen auf einem FUJITSU Scanner Modell M3097DE bei 300 DPI mit G4-Kompression gewonnen. Da Multipage nicht von allen Grafik-Programmen problemlos angezeigt und bearbeitet werden kann, wurde pro Seite eine eigene TIFF-Datei erzeugt. Der textuelle Teil der Groundtruth wurde direkt aus den HTLM-Seiten mit dem UNIX-Tool html2text [79] extrahiert. Die resultierende Codierung ISO-Latin-1 wurde anschließend mit dem UNIX-Tool recode [64] in die Codierung UTF-16BE überführt. Um zu jeder Bilddatei direkt den zugehörigen Text in einer eigenen Datei verfügbar zu haben, wurde der Textexport (manuell) zerteilt.
Da das Groundtruth-Korpus parallel zu dieser Arbeit entstanden ist, beziehen sich einige Messungen u. ä. auf Teilkorpora, die zum Zeitpunkt der Niederschrift verfügbar waren.

2.2 OCR-Korpus

2.2.1 OCR-Engines

Für den deutschen Massenmarkt sind eine Reihe von OCR-Engines zum Preis von ca. 100 Euro verfügbar. Diese werden von Zeit zu Zeit von der PC-Fachzeitschrift c't gegeneinander getestet (letzter Test [19]). Für diese Arbeit standen zwei Engines aus der Spitzengruppe dieser Tests zur Verfügung, ABBYY Finereader (Version 5.0 pro) und ScanSoft OmniPage (Version 10.0). Von diesen Engines existieren auch Entwicklerversionen, die über eine Programmierschnittstelle ansprechbar sind. Auf den Einsatz dieser deutlich teureren Engines konnte aber verzichtet werden, da für eine akademische Black-Box-Untersuchung die Endbenutzer-Engines genügen. Daneben gibt es noch OCR-Engines, die in ein komplexes Dokumenten-Management-System (DMS) integriert sind. Ein Großteil dieser Engines sind Eigenentwicklungen, einige wenige sind auch spezialisierte Weiterentwicklungen der genannten Profiversionen der Massenmarkt-OCR-Engines [65]. PaperIn der CCS Compact Computer Systeme AG ist ein Beispiel für solch ein DMS. Da in meiner Projektgruppe eine Kooperation mit dieser Firma besteht, konnte ich in dieser Arbeit auch die OCR-Engine von PaperIn einsetzen. Diese Engine ist eine Weiterentwicklung der Endbenutzer-Engine TextBridge von ScanSoft. Neben den kommerziellen OCR-Engines gibt es noch etwa ein Dutzend frei erhältlicher Open-Source-Entwicklungen. Davon habe ich die beiden mit den jüngsten feststellbaren aktiven Entwicklungstätigkeiten getestet, Ocrad [20] und ocre [93].

2.2.2 Ausgabeformate der OCR-Engines

Gemäß den Erwartungen, die man an eine OCR-Engine stellt, bieten alle getesteten OCR-Programme die Option einer Textausgabe ohne formales Markup. Die

Ausgabe enthält die zum Texte gehörige Interpunktion, das sog. punktuationale Markup[4]. Ausserdem ist i. Allg. auch präsentationales Markup enthalten, wie bspw. eine Folge von Minuszeichen, die eine durchgezogene Linie im Originaldokument wiedergeben oder die Simulation eines zweispaltigen Texts mit Hilfe von Leerzeichen. Dieses präsentationale Markup ist für einen Menschen intuitiv verstehbar, jedoch nicht näher spezifiziert. Die kommerziellen Programme enthalten darüber hinaus weitere Optionen zum Ausgabeformat. Die beiden Endbenutzer-Engines bieten eine Reihe prozeduraler Ausgabeformate für die Weiterverarbeitung der Dokumente: Textverarbeitung (DOC, RTF), Tabellenkalkulation (XLS), Ausdruck auf Papier (PDF), Publikation im Web (HTML), etc. Aus diesen Formaten lassen sich verschiedene Features des gescannten Dokuments auslesen, wie etwa geometrische Positionen oder Fonttypen. Das ist allerdings ein unbequemer Umweg, da

- manche Informationen nicht direkt enthalten sind. Geometrische Positionen sind bspw. in HTML zum Teil indirekt mittels Tabellen oder deskriptiven Elementen wie `center` codiert.

- die Spezifikation einiger Formate nicht frei zugänglich ist. Der Aufwand einer Rekonstruktion der Formatspezifikation darf nicht unterschätzt werden. In [16] wird ein Mannjahr für das Reverse-Engineering eines proprietären, prozeduralen Formats genannt.

Einfacher lassen sich Text-Features aus dem prozeduralen Format XDOC auslesen, ein Industriestandard zur Annotation von OCR-Output [7]. PaperIn bietet dieses Format als Ausgabeoption (neben einem Format, das für die Aufgabe der Eingangspostsortierung maßgeschneidert wurde, sowie purem Text) an. XDOC enthält auch Konfidenzwerte auf Zeichen- und Wortebene. Diese Werte geben eine Einschätzung der OCR zur Treffsicherheit der einzelnen Leseresultate. Bis auf spezielle Tests mit diesen Konfidenzwerten wurde im Rahmen dieser Arbeit nur Text ohne formales Markup betrachtet.

2.2.3 Auswahl der OCR-Engines

Die beiden open-source OCR-Engines wurden schon nach den ersten Vortests ausgemustert, da die Resultate katastrophal schlecht waren. Auf Testseiten des Korpus wurde kein einziges Wort komplett richtig erkannt. Da von PaperIn eine leicht veralteten Version am Institut vorliegt und daher eine nicht voll konkurrenzfähige Erkennungsrate aufweist, wurde diese OCR-Engine nur für Tests zur Integration der Konfidenzwerten verwendet. Die anderen beiden OCR-Engines sind von der Erkennungsrate ungefähr gleich stark. Es wurde entschieden, mit diesen beiden Engines die Evaluation durchzuführen.

[4]Die Definitionen verschiedener Markup-Typen stammen aus [8].

2.2.4 Korpus-Erstellung

Für eine Feinadjustierung der OCR-Programme wurde ein Kompromiss zwischen Zeiteffizienz und Erkennungsqualität gewählt. Es wurde lediglich die Sprachauswahl auf Deutsch bzw. Englisch gesetzt, ansonsten wurden alle Voreinstellungen übernommen, ähnlich einem Blackbox-Test. Das ist ein realistisches Szenario einer ORC-Anwendung in der industriellen Praxis. Für jede TIFF-Datei des Korpus (d. h. Seite) wurde pro OCR-Engine eine Textdatei erzeugt. Mit dem UNIX-Tool recode [64] wurden die Codierungen zusammen mit den Groundtruth-Daten zu UTF-16BE vereinheitlicht.

Kapitel 3

Software

Im diesem Kapitel stelle ich meine Software zur lexikalischen Nachkorrektur OCR-gelesener Texte vor, die ein Feineinstellung diverser Einflussparameter erlaubt. Zuerst wird besprochen, wie sich die Software in eine Prozesskette zwischen OCR-Engine und weiteren Verarbeitungsschritten integrieren lässt. Dazu wird gezeigt, wie verschiedene OCR-Ausgaben zu vereinheitlichen sind, und es werden die Außenschnittstellen meiner Software exakt spezifiziert. Anschließend wird der zweigeteilte, interne Aufbau vorgestellt. Die erste Komponente erzeugt Korrekturfiles, die sämtliche Korrekturvorschläge bis zu einem vorgegebenen String-Abstand (Levenshtein-Abstand $\leqslant 2$) aus allen vorhandenen Lexika enthalten. Die Fehlerbilanz im Resultat fällt allerdings negativ aus, wenn man alle diese Vorschläge blind ausführt. Daher motiviert sich die zweite Komponente zur Ermittlung einer Vertrauensgrenze der Nachkorrektur, geeigneter Lexika, einer geeigneten Gewichtung von Nachkorrekturhilfen, etc. Mit Hilfe einer graphischen Oberfläche können die Einflussparameter auf Trainingsmaterial adjustieren und für den Produktionsbetrieb übernommen werden.

3.1 Komponierbarkeit

3.1.1 Ziel

Da die lexikalische Nachkorrektur im Document-Engineering nur *einen* Bearbeitungsschritt in einer Prozesskette darstellt, ist es erforderlich, dass keine Information, die in der OCR-Ausgabe vorhanden war, durch die Nachkorrektur verloren geht. Zur Verdeutlichung habe ich als Beispiel eines komplexeren Dokumentensystems HYPERFACS gewählt [55]. Ziel des Systems ist, eine gedruckte Dokumentensammlung in ein verlinktes Hypermedium zu überführen. Das Ablaufdiagramm habe ich direkt aus der Veröffentlichung übernommen.

Wollte man dem System meine lexikalische Nachkorrektur nachträglich einbauen, würde man diese zwischen Prozessschritt 2 und 3 platzieren. Da Prozessschritt 3 direkt auf KDOC, einem proprietären, prozeduralen Ausgabeformat von ScanWorX aufsetzt, ist es wichtig, diese Schnittstelle beizubehalten.

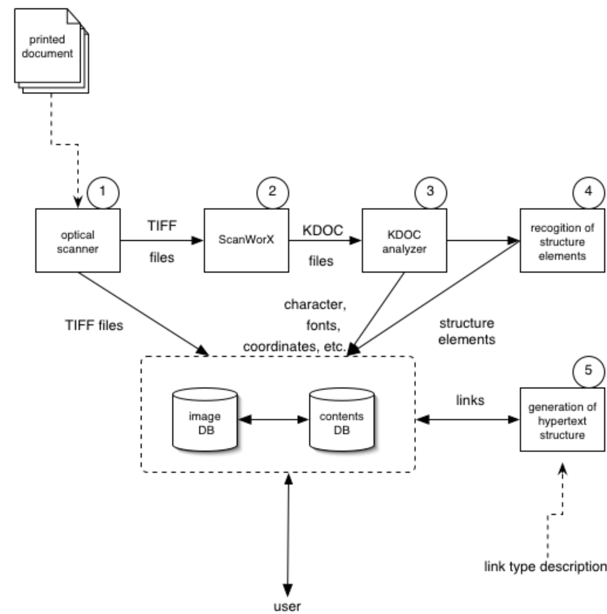


Abbildung 3.1: Ablaufdiagramm von HYPERFACS.

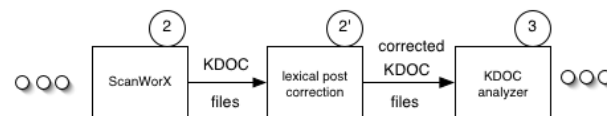


Abbildung 3.2: Einbindungsszenario für HYPERFACS.

Unter Beibehaltung der Schnittstelle zur OCR-Ausgabe lässt sich die lexikalische Nachkorrektur auch mit anderen Nachkorrekturen kombinieren, die einen orthogonalen Ansatz verfolgen, wie etwa das in [30] und [31] beschriebene Verfahren.

3.1.2 Realisierung

Diskussion zweier Ansätze

Zur Realisierung dieser Komponierbarkeit stehen zwei Ansätze zur Verfügung:

1. Das OCR-Ausgabe-File wird komplett (d. h. inklusive Markup) in eine Hauptspeicherdatenstruktur überführt, auf dieser Datenstruktur wird die lexikalische Nachkorrektur ausgeführt und zum Schluss wird sie wieder mit gleichgeartetem Markup exportiert.

2. Es werden lediglich die Wörter aus dem OCR-Ausgabe-File herausgefischt und der lexikalischen Nachkorrektur übergeben. Resultat dieser lexikalischen Nachkorrektur ist eine Menge von Korrekturanweisungen, die direkt auf dem OCR-Ausgabe-File ausgeführt werden.

Der erste Ansatz empfiehlt sich für ein umfassendes Nachkorrektursystem, da auf alle von der OCR gelieferten Informationen (z. B. erkannter Fonttyp oder -schnitt) direkt zugegriffen werden kann. Damit ist dieser Ansatz auch flexibler gegenüber Erweiterungen der Nachkorrekturstrategie. Diese Vorteile sind jedoch abzuwägen gegen größeren Hauptspeicherbedarf, größere Komplexität der Implementierung und damit verbundene geringere Robustheit. Da in dieser Arbeit eine Untersuchung der *lexikalischen* Nachkorrektur im Mittelpunkt steht, habe ich mich für den zweiten, schlankeren Ansatz entschieden. Der Kern dieses Ansatzes ist ein Adressierungsmechanismus, der es erlaubt, ein Wort aus der OCR-Rückgabe gegen einen Korrekturvorschlag auszutauschen. Dieser Ansatz funktioniert problemlos, solange im Text keine Rückbezüglichkeiten zwischen Inhalt und Markup bestehen. Bei punktuationalem Markup, d. h. Interpunktionszeichen, besteht keinerlei Gefahr. Bei präsentationalem Markup ist z. B. folgender Effekt denkbar: ein Wort wird durch eine Reihe von Minuszeichen unterstrichen; wird ein Wort durch ein längeres Wort ausgetauscht, müsste auch die Unterstreichung verlängert werden. Prozedurales Markup enthält sogar noch häufiger solche Abhängigkeiten zwischen Visualisierungsanweisungen und Inhalt. Deskriptives Markup ist in der Regel frei von Rückbezüglichkeiten. Integrierte Formate, die eine Kompression, Indexierung, Verschlüsselung, o. ä. enthalten, sind mit dem Adressierungsansatz nicht zu bewältigen.

Adressierung von Tokens mittels File-Positionen

Eine OCR-Ausgabe (egal mit welchem Markup) ist eine endliche Folge von Zeichen eines Alphabets. $\varepsilon_i \in \Sigma$, $0 \leq i < document_length$. Dadurch lässt sich jede Teilfolge $\varepsilon_a \varepsilon_{a+1} \ldots \varepsilon_b$ durch ein Zahlentupel (a, b) mit $0 \leq a \leq b \leq$

document_length eindeutig beschreiben. Im Kontext der Beschreibung von Textteilen heißt eine solche Teilfolge Region. Die Region $(0, document_length - 1)$ beschreibt z. B. das Dokument selbst und die Region $(0, 0)$ enthält nur das erste Zeichen ε_0 des Dokuments. Eine Regionenbeschreibung durch Startpunkt und Offset wäre etwas speicherplatzsparender gewesen, da alle Zahlenwerte als XML-Attribute in rein textueller Form abgespeichert werden, aber der gewählte Regionenaufbau lehnt sich direkt an die Syntax von `sgrep` an. In [34] und [35] wird diese Implementierung einer Regionenalgebra zur Anfrage an strukturierte Dokumente näher vorgestellt. Dieses Werkzeug eignet sich, um von der Kommandozeile direkt auf eine Region zugreifen zu können:

```
sgrep '[(0,12)]' file.txt
```

Da `sgrep` keine regulären Ausdrücke unterstützt, ist die Software jedoch für einen weitergehenden Einsatz in meiner Arbeit nicht geeignet.
Ein Token ist das technische Pendant zu einem Wort; das Token enthält die Zeichen, die zusammen ein Wort bilden. In der Regel korrespondiert *ein* Token direkt mit *einer* Region. Es gibt aber auch Beispiele, bei denen sich ein Token auf zwei oder mehr nicht aufeinanderfolgende Regionen erstreckt, da zwischen den inhaltstragenden Zeichen Markup-Teile eingeschoben sind. Je ein Beispiel mit formalen und punktuationalen Markup werden weiter unten konkretisiert). Die Adressierung von Tokens lässt sich also im einfachen Fall als Region implementieren und im allgemeineren Fall als Liste von Regionen. Im folgenden Kapitel wird noch eine weitere alternative Adressierungsmöglichkeit von Tokens vorgestellt. Um zu diesen Optionen eine gemeinsame Schnittstelle zu schaffen, habe ich ein Java-Interface definiert:

```
public interface Token {
String getContent();
void setContent(String content);
}
```

Die Referenzimplementierung dieser Arbeit verwendet eine einfache Java-Klasse `Region`, die das Interface `Token` implementiert. Bestehend aus einem String-Attribut für den textuellen Inhalt und zwei int-Attributen für die Start- und Endposition. Bei einer Implementierung des Interfaces mit Regionenlisten, erhält man den Inhalt durch Konkatenation der Inhalte der einzelnen Regionen. Die in der Arbeit vorgestellte Software ist in einer im Software-Engineering-Bereich typischen Weise zweigeteilt, in Interfaces und einer Referenzimplementierung. Der Architekturaufbau der Software wird durch eine Modularisierung der Aufgabenstellung in einzelne Interfaces und deren Beziehungen untereinander beschrieben. Die Interfaces selbst bestehen aus einer Deklaration von notwendigen Funktionen, im Bereich der Objektorientierung Methoden-Signaturen genannt. Die Referenzimplementierung ist ein Nachweis der Realisierbarkeit, der durch die Interfaces vorgestellten Architektur. Die als Hauptimplementierungssprache für diese Arbeit gewählte Sprache Java [77] unterstützt dieses zweigeteilte Konzept direkt im Sprachkern. Ein weit verbreitetes Beispiel für

Software-Entwicklungen, die in dieser Weise organisiert sind, sind JavaServlets und JavaServer Pages und deren Referenzimplementierung Tomcat [1].

Alternative Adressierung von Tokens mittels XPath

In [27] und [26] werden die Vorteile einer Verwendung von XML für alle Zwischenformate im OCR-Bereich herausgestellt, vornehmlich die Möglichkeit des Rückgriffs, auf die vielen im XML-Umfeld vorhandenen Werkzeuge. Auch für die Adressierung von Tokens wäre mit XPath ([83]) ein geeigneter Standard vorhanden. In [48] und [6] sind bereits DTDs für eine Verpackung von OCR-Output in SGML-Markup, jeweils mit Umsetzung innerhalb akademischer Prototypen, vorgestellt worden. Allerdings ist diese Art von OCR-Ausgabe in keiner der für diese Arbeit verwendeten OCR-Engines vorhanden.

Zeichensatz

Die Auswahl eines konkreten Zeichensatzes an Stelle von Σ ist eine Abwägung zwischen Speicherplatzbedarf und Grad der Internationalisierung. Für englischsprachige Dokumente würde ASCII, für deutschsprachige Dokumente ISO-8859-1 (besser bekannt als Latin-1) genügen. Da aber die Software, die im Rahmen dieser Arbeit entwickelt wurde auch für kyrillische Dokumente in der Arbeitsgruppe von Stoyan Mihov an der bulgarischen Akademie der Wissenschaft eingesetzt wird, wird der Unicode-Zeichensatz [78] verwendet. Unicode tritt mit dem Anspruch an, system-, programm- und sprachunabhängig jedem Zeichen eine eigene Nummer zuzuordnen. Am Anfang (d. h. 1991) waren dafür ein 16-bit Adressraum für 65536 Zeichen vorgesehen. Um aber auch exotische und historische Schriftsysteme in Unicode unterzubringen, musste der Adressraum auf 32-bit erweitert werden. Trotzdem wurde entschieden, für diese Arbeit nur den ursprünglichen 16-bit Adressraum zu verwenden, da

- der 32-bit Ansatz für diese Arbeit absolut überdimensioniert ist,
- die eingesetzte Implementierungssprache Java auch den 16-bit Adressraum verwendet,
- durch die relative Frische der Erweiterung, beim Einsatz diverser Software-Komponenten mit Problemen zu rechnen ist und
- die Festlegung eine spätere Expansion nicht vollkommen ausschließt.

Diese unteren 64K des Adressraums werden auch *basic multilingual plane* (BMP) genannt.
Zu dem Zeichensatz muss zusätzlich noch eine Codierung ausgewählt werden. Der Unicode-Standard nennt die drei Auswahlmöglichkeiten UTF-8 [92], UTF-16 [29] und UTF-32. Daneben existieren noch eine Reihe weiterer Codierungen mit speziellen Eigenschaften wie z. B. UTF-7, das jedes Unicode-Zeichen auf eine Folge von ASCII-Zeichen abbildet [22]. Die Auswahl ist durch eine Reihe von Zielkonflikten bestimmt:

- Software-Unterstützung
- Speicherplatzbedarf
- Verarbeitungsgeschwindigkeit und -komplexität

Da die Arbeiten in heterogener Betriebsystemumgebung entstehen (die eingesetzten OCR-Engines laufen unter Windows2000, Arbeitsplatzrechner am Institut sind mit Linux ausgestattet, auch MacOSX wird auf einem Laptop eingesetzt) ist eine breite Software-Unterstützung unabdingbar. Da das nur von prominenteren Standards zu erwarten ist, habe ich nur Texte in den Codierungen UTF-8, UTF-16 und UTF-32 zum Testen von weitverbreiteten Editoren und Textverarbeitungsprogrammen auf den genannten Plattformen herangezogen. Es hat sich schnell gezeigt, dass heute noch keine dieser Codierungen direkt, problemlos eingesetzt werden kann. Es ließe sich zwar für jedes Betriebsystem ein Bündel an geeigneten Programmen zusammenstellen und konfigurieren, jedoch verbietet das der inakzeptabel hohe Systemadministrationsaufwand. Daher habe ich entschieden, dem Arbeitsprozess einen Zwischenschritt einzufügen. Die Texte werden zwar in Unicode gespeichert, aber zum Lesen und Editieren werden sie vorher in einen lokal gebräuchlichen Zeichensatz konvertiert, den auch die individuell bevorzugt eingesetzten Editoren und Textverarbeitungsprogramme beherrschen; ISO-8859-1, um die deutschen Umlaute oder ISO-8859-5, um Zeichen des kyrillischen Alphabets korrekt bearbeiten zu können. Für die Umsetzung dieses Zwischenschritts braucht man ein plattformübergreifendes Konvertierungsprogramm für Zeichensätze. Im Java SDK [77] ist das wenig beachtete Werkzeug `native2ascii` enthalten. Damit kann man Texte zwischen einer Reihe von Zeichensätzen und einer proprietären Codierung des Unicode-Zeichensatzes hin- und herkonvertieren. Diese Java-Unicode-Codierung basiert auf der gleichen Idee wie UTF-7; alle Zeichen, die nicht in ASCII enthalten sind, werden durch eine spezielle Escape-Sequenz dargestellt. Da das Werkzeug die beiden Unicode-Codierungen UTF-8 und UTF-16 und eine Reihe von 8-bit ISO Zeichencodierungen unterstützt, kann man in zwei Schritten zwischen einem lokalen Zeichensatz und einer Unicode-Codierung wechseln. Die proprietäre Codierung selbst dient dabei nur als Zwischenformat. Da UTF-32 nicht unterstützt wird, wird im folgenden Teil nur noch zwischen UTF-8 und UTF-16 entschieden. Alternativ zu `native2ascii` kann man auch das GNU-Tool `recode` [64] einsetzen. Es werden deutlich mehr Codierungen unterstützt, allerdings ist die Software bei weitem nicht so plattformübergreifend verfügbar. Eine unbekannte Codierung eines vorliegenden Textfiles kann man mit dem UNIX-Kommando `file` erfragen[1].
Eigentlich sind UTF-8 und UTF-16 beide dynamische Codierungen, d. h. die Zeichen können unterschiedlich lang codiert sein, aber bei der Beschränkung auf das Unicode BMP sind in UTF-16 alle Zeichen mit zwei Bytes codiert, hingegen bei UTF-8 variiert die Codierung von einem Byte für Zeichen des ASCII-Alphabets bis hinzu 6 Bytes. Bei einem Test auf einem englischen und einem

[1]Portierungen des Kommandos für andere Plattformen sind verfügbar

deutschen Textkorpus mit jeweils mehr als 10^6 Zeichen ist die UTF-16 Version um den Faktor 1,905 bzw. 1,883 größer als die UTF-8 Version. Der niedrigere Faktor beim deutschen Korpus rührt von den Umlauten her, die in beiden Codierungen 2 Bytes beanspruchen. Verwendet man eine Standard-Kompression (gzip), fällt der Nachteil von UTF-16 gegenüber UTF-8 mit den Faktoren 1,167 bzw. 1,157 deutlich geringer aus. Eine Kompression sollte man für Archivierung und Transport größerer Textfiles auf jeden Fall verwenden, da auch bei UTF-8 das Datenvolumen auf etwa ein Drittel reduziert werden kann.
Der (geringe) Speicherplatzvorteil von UTF-8 gegenüber UTF-16 wird durch Nachteile in der Verarbeitung erkauft. Um ein einzelnes Zeichen zu laden, muss man erst ein Byte laden und dann prüfen, ob noch weitere Bytes nachzuladen sind. Dieser Algorithmus verbietet, gezielt auf ein Zeichen mit einer maschinennahen Operation zuzugreifen. Es ist auch nicht möglich, auf eine Region mit *random access* zuzugreifen, wenn sie wie in 3.1.2 beschrieben adressiert ist.
Da die einfachere Handhabung den Platzvorteil überragt, habe ich mich für UTF-16 entschieden, noch genauer für UTF-16BE. Durch die genauere Spezifikation der Byte-Reihenfolge *big endian* (BE), entfallen die beiden Bytes zur *byte order mark* (BOM) am Anfang jeder Datei. Beide vorgestellten Werkzeuge zur Konversion (`native2ascii` sowie `recode`) beherrschen diesen Substandard von UTF-16.

3.2 Spezifikation der Außenschnittstellen

In diesem Abschnitt werden sowohl die OCR-Engines, als auch die lexikalische Nachkorrektur als Blackboxes betrachtet. Die OCR-Engines geben Textdokumente aus, die mit Markup annotiert sind, dessen Spezifikation vorliegt. Für die Referenzimplementierung wird vereinfachend angenommen, dass nur punktuationales und präsentationales Markup in den Dokumenten enthalten ist. Aus den beiden Fragen

1. Welche minimale Eingabe benötigt die lexikalische Nachkorrektur?
2. Welche Ausgabe soll die lexikalische Nachkorrektur liefern?

wird die genaue Spezifikation der Ein- und Ausgabedateiformate erarbeitet, inkl. der Arbeitsschritte, die dazu durchlaufen werden müssen. Mit der Ausgliederung und Modularisierung der vor- und nachgelagerten Schritte wird der Kern der lexikalischen Nachkorrektur so schlank wie möglich gestaltet.
In meiner lexikalischen Nachkorrektur werden drei unterschiedliche Phasen durchlaufen, die für die Schnittstellenspezifikation zu beachten sind:

1. **Trainingsphase**. In der Trainingsphase wird versucht, alle Parametereinstellungen für einen Datensatz optimal einzustellen. Optimalitätskriterium ist in diesem Zusammenhang die Genauigkeit von OCR-Erkennung plus Nachkorrektur gegenüber der Groundtruth. Da bei der Vielzahl an Parametern ein komplettes Durchprobieren aller Einstellungen – bedingt

durch die kombinatorische Vielfalt – in der Komplexität entarten würde, wird nur ein Teil der Parameter brute-force optimiert. Für die anderen Parameter wird entweder die Optimierung mit heuristischen Verfahren approximiert oder die Adjustierung an einen Benutzer weiterdelegiert.

2. **Testphase**. In der Testphase werden die optimierten Parametereinstellungen der Trainingsphase übernommen und die damit erzielte Genauigkeit auf einem Testdatensatz gemessen. Ist das Ergebnis unbefriedigend, wird entweder die Trainingsphase wiederholt oder die Nachkorrektur nicht eingesetzt.

3. **Produktionsbetrieb**. Für eine isolierte wissenschaftliche Untersuchung zur lexikalischen Nachkorrektur würde ein Training von Parametereinstellungen mit anschließendem Test ausreichen. Für den Produktionsbetrieb, eine (kommerzielle) Umsetzung meines Verfahrens für die Praxis, sind diese beiden Phasen jedoch nur eine Vorbereitung. Auf einem Sample einer zu bearbeitenden Dokumentensammlung werden optimale Parametereinstellungen ermittelt, die dann für den Rest der Dokumentensammlung angewandt werden. Anders als in der Testphase finden keine umfassenden Qualitätsmessungen mehr statt, sondern nur noch Stichproben.

3.2.1 Eingabe für die lexikalische Nachkorrektur

Für eine konkrete Nachkorrekturaufgabe werden mit meiner Software die drei Arbeitsmodi in der Reihenfolge Trainings-, Test- und Produktionsbetrieb durchlaufen. Da aber der Produktionsbetrieb davon die einfachste Eingabedatenstruktur – eine Liste von Wörtern – benötigt, ziehe ich dessen Beschreibung in diesem Abschnitt vor. Innerhalb des Paragraphen zum Produktionsbetrieb wird eine Reihe von Detailproblemen erörtert, die sich bei der Zerlegung von OCR-Dokumenten in Wörter ergeben:

- Ausfiltern von Markup, wobei verschiedene Markup-Typen zu unterscheiden sind
- Charakteristika, die echte Wörter von anderen Tokens in Dokumenten unterscheiden (bspw. . enthaltene Symbole und Länge)

Aufbauend darauf werden im Anschluss die zusätzlichen Eingabeanforderungen für den Trainings- und Testbetrieb diskutiert.

Produktionsbetrieb

Als Eingabe benötigt die lexikalische Nachkorrektur im Produktionsbetrieb lediglich eine Liste von Wörtern. Mit Wörtern werden in diesem Zusammenhang Tokens bezeichnet, die man als Mensch in einem Lexikon nachschlagen würde. Auf den ersten Blick sieht diese Definition wie eine nebensächliche Eingangsbemerkung aus. Da es aber gilt, diese menschliche Intuition algorithmisch umzusetzen, und bei der Umsetzung einige Detailprobleme auftauchen, die die Grenzen

einer lexikalischen Nachkorrektur gut aufzeigen, ist dieser Abschnitt vielleicht ausführlicher als erwartet dargestellt. Zur Verdeutlichung eine Zerlegung des Problems in drei Hauptschritte:

1. **Markup Ausfiltern.** In einem Lexikon ist nur textueller Inhalt enthalten. Es muss also jede Art von Markup aus der OCR-Ausgabe ausgefiltert werden. Das Spektrum reicht von Font-Angaben eines prozeduralen Formats bis hin zu Interpunktion.

2. **Begrenzungen bestimmen.** Da in einem Lexikon typischerweise Einzelwörter verzeichnet sind, ist der textuelle Inhalt in einzelne Wörter zu zerlegen. Meine Referenzimplementierung ist auf Einzelworte beschränkt. Mit wachsender Rechen- und Speicherkapazität wird es in Zukunft auch mehr und mehr Phrasenlexika geben, so dass man Gruppen von Wörtern nachschlagen kann. Aber generell gilt es, die Begrenzungen der Einträge zu bestimmen.

3. **Im Lexikon nachschlagen?** In [2] werden in einer generellen Textnachbearbeitungsarchitektur drei Klassen von wortähnlichen Tokens unterschieden, für die unterschiedliche Bearbeitungsspezialisten notwendig sind: *alphabetical words*, *alphanumeric words* und *words on their own right*. Nur alphabetical words sollen im Lexikon nachgeschlagen werden. Alphanumeric words wie z. B. Dezimalzahlen würde man nicht in einem Lexikon speichern, da es unendlich viele davon gibt. Zur Bearbeitung sind grammatikbasierte Verfahren besser geeignet. Abhängig von der Anwendungsdomäne und den vorhandenen Lexika muss entschieden werden, ob ein alphanumeric word doch besser als Wort gewertet wird. Einige Beispiele aus diesem Graubereich sind Bankleitzahlen oder ISBN-Nummern. Words on their own right bilden die Restklasse der Ausnahmefälle. Auch hier existiert ein Graubereich an Subklassen, die anwendungsabhängig besser in einem Lexikon organisiert werden können, wie z. B. Email-Adressen oder Identifier in Programmausdrucken.

Für die schrittweise Zerlegung von Dokumenten werden hier folgende Begriffe eingeführt:

OCR-Ausgabe bezeichnet das gesamte File, also Text zusammen mit allem Markup

Text ist die OCR-Ausgabe ohne formales Markup, aber inkl. punktuationalen und präsentationalen Markup sowie Whitespaces

prätextuelles Token ist eine zusammengehörige Zeichenfolge ohne Whitespaces, die allerdings i. Allg. noch punktuationales und präsentationales Markup enthält (bspw. `(bspw.` oder `!WICHTIG!`).

textuelles Token ist eine prätextuelles Token ohne punktuationales und präsentationales Markup. Die Beispiele des vorherigen Punkts lauten als textuelle Tokens `bspw` und `WICHTIG`.

normales Token ist ein textuelles Token, das für einen Look-Up in einem Lexikon in Betracht kommt. Die Entscheidung wird an Hand des Aufbaus des textuellen Tokens gefällt. Als normal gelten z. B. Tokens wie `Ölkanne` oder `ISDN`, dagegen als nicht normal gelten z. B. die Tokens `34534314` oder `h:a:l:l:o`

Wort ist ein normales Token, das im Lexikon angefragt wird. Im Produktionsbetrieb sind alle normalen Tokens auch Wörter. Im Trainings- und Testbetrieb kommen spezielle Markierungen hinzu, die diese Unterscheidung notwendig machen.

Zu der schrittweisen Zerlegung werden auch zwei Java-Interfaces definiert:

```
public interface Tokenizer {
static final String FILE_ENCODING = "UTF-16BE";
void setSourceFile(File sourceFile);
List getText();
List getPreTextualTokens();
List getTextualTokens();
List getQueryTokens();
}
```

sowie

```
public interface TokenProperties {
boolean isNormal(Token t);
boolean isFiltered(Token t);
}
```

Mit der Methode `setSourceFile()` wird die Ausgabe der OCR-Engine gesetzt. Die eingesetzte Codierung wird im Interface als Konstante vermerkt. Mit `setSourceFile()` anstatt mit `setOCRFile()` wurde die Methode allgemeiner benannt, da bei der Tokenisierung der Groundtruth dieses Interface wiederverwendet wird. Außerdem ist denkbar, dass die Nachkorrektur auch für andere Fehlerklassen wie z. B. Tippfehler oder DFÜ-Fehler eingesetzt wird. Mit der Methode `getText()` wird alles formale Markup ausgefiltert und eine Liste von Tokens zurückgegeben, die reinen Text beinhalten. Da in der prototypischen Referenzimplementierung als Source-Files nur reine Text-Files verwendet werden, liefert diese Methode eine Liste mit nur einer Region zurück, die das gesamte File umfasst. Die Rückgabelisten aller Methoden enthalten Tokens, die in der Referenzimplementierung mit Hilfe einfacher Regionen adressiert werden. Dies ist aber nur eine Vereinfachung, da logisch zusammenhängende Textteile auch auf mehrere (nicht sequentielle) Regionen verteilt sein können. Dazu ein Beispiel mit einem HTML-Fragment:

```
<b>E</b>s war einmal ein König
```

Beim Aufruf von `getText()` ist eine Implementierung mit Regionenlisten einer einfachen Implementierung nicht wirklich überlegen:

```
[[(3, 3)], [(8, 29)]]
```

versus

```
[(3, 3), (8, 29)]
```

Jedoch bei der Methode `getPreTextualTokens()`. Sie liefert aus dem Text eine Liste aller prätextuellen Tokens. Also sollte inhaltlich die Liste so aussehen:

```
["Es", "war", "einmal", "ein", "König"]
```

Die korrespondierende Regionenschreibweise muss so aussehen:

```
[[(3, 3), (8, 8)], [(10, 12)], [(14, 19)], [(21, 23)], [(25, 29)]]
```

Will man solche Fälle mitbehandeln, ist eine Adressierung von Tokens mit Hilfe von Regionenlisten notwendig. In der Referenzimplementierung der Methode `getPreTextualTokens()` werden alle maximalen Tokens des Texts zurückgegeben, die kein Whitespace enthalten. Maximal bedeutet, dass eine Region in keiner anderen enthalten ist und auch mit keiner anderen Region überlappt. Die Entscheidung, welches Zeichen ein Whitespace ist, fällt die Methode `isWhitespace()` der Klasse `Character`. Dort werden die Unicode-Kategorien *line*, *paragraph* und *space separator* (allerdings ohne *no-break spaces*) sowie neun weitere Zeichen als Whitespaces definiert. Die Aufzählung zeigt, dass man bei der Spezifikation genau sein muss, um bei verschiedenen Implementierungen konsistent zu bleiben. Wollte man Worttrennungen mit im System behandeln, wäre auch ein Trennungsstrich gefolgt von einem Zeilenumbruch innerhalb eines prätextuellen Tokens erlaubt, also z. B. "`Trenn-\nstrich`". Wollte man Phrasenlexika einsetzen, wären auch *space separators* innerhalb eines prätextuellen Tokens erlaubt, also beispielsweise "`Phrasen dreschen`". Die Methode `getTextualTokens()` wandelt die Liste der prätextuellen Tokens in eine Liste der textuellen Tokens um. In der Referenzimplementierung wird die vereinfachende Annahme umgesetzt, dass punktuationales Markup lediglich an den Token-Rändern auftritt. Von beiden Seiten werden Zeichen der Klasse `punct` des POSIX Standards entfernt. Prätextuelle Tokens, die nur aus punktuationalen Markup bestehen, werden aus der Liste textueller Tokens entfernt. Auch beim Übergang von prätextuellen Tokens zu textuellen Tokens kann die Adressierung von Tokens mit Hilfe von Regionenlisten notwendig sein, wenn man auch punktuationales Markup innerhalb der Tokens ausfiltert. Greift man etwa das Beispiel mit dem Trennungsstrich nochmals auf, so wird ein File, das nur den String "`Beispiel mit Trennungs-\nstrich`" enthält, so in eine Liste textueller Tokens zerlegt:

```
[[(0, 7)], [(9, 11)], [(13, 17), (20, 25)]]
```

Die Methode `getQueryTokens()` selektiert aus der Liste textueller Tokens alle normalen Tokens; die Entscheidung fällt mit Hilfe der Methode `isNormal()`. Diese Methode setze ich sowohl für den Aufbau von Lexika, als auch als Entscheidungskriterium für Look-Ups ein. Wer schon einmal eine stark verunstaltete OCR-Ausgabe gesehen hat (z. B. `Ve+un5ta1tung` anstatt `Verunstaltung`), wird sich an dieser Stelle unweigerlich fragen, ob es nicht gerade die anormalen textuellen Tokens sind, die man korrigieren möchte. Die Erstellung des OCR-Korpus dieser Arbeit hat aber gezeigt, dass mit den verwendeten OCR-Engines bereits bei mittlerer Vorlagequalität (Ausdruck einmal kopiert) diese Verunstaltungen der Vergangenheit angehören; hingegen bei schlechter Vorlagequalität fallen die Resultate so schlecht aus, dass eine lexikalische Nachkorrektur fast chancenlos ist. Die Anzahl anormaler Tokens, die schon in der Groundtruth enthalten sind übersteigt die Anzahl der dazu von der OCR generierten anormalen Tokens bei weiten. Experimente haben deutlich gezeigt, dass der Versuch auch anormale Tokens in die Korrektur mit einzubeziehen mehr schadet als nutzt. Eine Verfeinerung wäre eine Aufgliederung in zwei Methoden. Eine Basismethode, die den Aufbau textueller Tokens vorgibt, die in ein Lexikon aufgenommen werden sollen und eine weitere Methode, die angibt wie weit ein OCR-Token davon maximal abweichen darf, z. B. mit einem vorgegebenen Levenshtein-Abstand.
Da die Methode `isNormal()` zur Vereinfachung auch für den Lexikonaufbau verwendet wird, ist sie nicht im Interface `Tokenizer` enthalten, und da ihre Implementierung bei Experimenten öfters variiert wird – anders als die Adressierung von Tokens – ist sie auch nicht im Interface `Token` verankert, sondern in einem eigenen Interface `TokenProperties`, zusammen mit einer Methode, die für den Trainings- und Testbetrieb benötigt wird. Die Wahl einer geeigneten Implementierung der Methode `isNormal()` ist vergleichbar mit der Adjustierung der beiden Antagonisten *precision* und *recall* in einem IR-System.
In der Referenzimplementierung wird mit dem einfachen, erweiterten regulären Ausdruck `[a-zA-ZäöüÄÖÜß-]{2,64}` die Normalität geprüft. Es folgt eine detaillierte Diskussion dieser Wahl.

Zeichensatz normaler Tokens Die restriktive Auswahl der Zeichen und ihre einfache Anordnung führen zu einer hohen Präzision. Der reguläre Ausdruck überdeckt nur wenige Strings, die Charakteristiken enthalten, so dass sie offensichtlich kein Wort bilden, wie z. B. mehrere aneinander gereihte Bindestriche. Im meinem Testkorpus (sieben verschiedenen Themengebiete der Groundtruth-Dokumentensammlung, Englisch/Deutsch mit mehr als 10^5 textuellen Tokens) sind auch keine derartigen Strings enthalten. Allerdings muss man vorsichtig sein, wenn man solche Charakteristiken non-normaler Tokens formuliert. Etwa eine Mischung von Groß- und Kleinschreibung tritt durchaus auch bei Tokens auf, die man in einem Speziallexikon aufnehmen möchte. Beispiele aus dem Testkorpus sind `McGurk` (Eigenname), `NaOH-extractable` und `CaMV` (Abkürzung für Cauliflower Mosaik Virus). Neben der hohen Präzision bietet der einfache, erweiterte reguläre Ausdruck:

- Eine – aus dem Blickwinkel des Software-Engineerings – robuste Umset-

zungsmöglichkeit. Die Robustheit liegt in der Kürze und der Formulierung in einem weit verbreiteten Standard, da im Software-Prototypen-Bau kurze Entwicklungszyklen, Reimplementierung in verschiedenen Programmiersprachen und parallele, isolierte Entwicklung üblich sind. Die Entwicklung der vorgestellten Referenzimplementierung lief parallel mit der Nutzung für wissenschaftliche Publikationen. In verschiedenen Reifestadien der Software (zu der auch betreute Studenten und Forschungspartner beigetragen haben) wurde die Normalitätsprüfung in awk, C, Java und Perl implementiert.

- Eine schnelle Laufzeit. Da die Methode aber im Rahmen der vorgestellten Arbeit nicht laufzeitkritisch eingesetzt wird, spielt dieser Vorteil kaum eine Rolle.

Diese Vorteile gehen aber alle auf Kosten des Recalls. In der Rückgabemenge normaler Tokens fehlen dadurch textuelle Tokens die Zeichen folgenden Typs enthalten:

- **Buchstaben anderer Alphabete**. Im Testkorpus habe ich eine Reihe französischer und ein paar wenige spanische Eigennamen und Fachbegriffe mit entsprechenden Sonderzeichen gefunden, bspw. `Hôpital Cantonal de Genève`, `Politècnica de Catalunya` oder `political naïveté`. Bei einer Weiterentwicklung der Software, ist es sinnvoll solche Tokens auch als normal zu werten. Durch die Wahl von UTF-16BE als Zeichencodierung ist dieser Schritt auch schon vorbereitet. Die Tatsache, dass es derzeit kaum ein professionelles Software-Tool gibt, das eine umfassende Internationalisierung problemlos bewältigt, zeigt aber, dass man den Aufwand für diesen Ausbau nicht unterschätzen sollte. Eine Erweiterung sollte daher inkrementell und am Bedarf orientiert erfolgen.

- **Zahlen**. Im Testkorpus finden sich auch textuelle Tokens, die Zahlen enthalten und für eine Organisation in einem Lexikon geeignet sind. Das sind Zahlworte (bspw. `2nd`), Ziffern vor Suffixen (bspw. `7fach`), Fachbegriffe (`3-methoxy-4-hydroxyphenylglycol` oder `Indol-3-carbinol`) und Akronyme (bspw. `CYP2B1-Induktion` oder `ICD-9`). Anders als beim vorherigen Punkt handelt es sich aber bei Mitaufnahme der Zahlen nicht um ein Skalierungsproblem, sondern ein komplexes Entscheidungsproblem, da es Kategorien von Tokens mit Zahlen gibt, die sicher nicht geeignet sind für die lexikalische Nachkorrektur, sondern besser durch gesonderte Automaten zu beschreiben sind. Ein Beispiel aus dem Korpus sind Seitenangaben von Literaturhinweisen. Es treten so unterschiedliche Ausprägungen wie `S.19`, `22--24` oder `11f` auf. In mancher Anwendung kann es sinnvoll sein, häufig zu erwartende Ausprägungen in einem Pseudolexikon zu organisieren. In dem angeführten Korpus finden sich bspw. mehr als 100 vierstellige Jahreszahlen (wobei keine der gefundenen Jahreszahlen in der Zukunft lieg), was etwa einem Promille entspricht.

- **Punktuation**. Im Testkorpus enthalten ca. 0.5% aller Tokens einen Bindestrich. Da dieses punktuationale Markup-Zeichen oft zur Kompositumbildung verwendet wird, habe ich es als einziges seiner Art auch zu den Bausteinen normaler Tokens gezählt. In den englischen Fachtexten (bspw. `land-holders` oder `multi-scale`) wird er sogar noch häufiger als in den deutschen (z. B. `Protein-Interaktionen` oder `literarisch-fiktionaler`) verwendet. Problematisch ist dabei die Ähnlichkeit des Bindestrichs mit dem Trennstrich, Spiegelstrich, Gedankenstrich, Minuszeichen, etc. In Codierungen wie Unicode existieren zwar unterschiedliche Zeichen für diese unterschiedlichen Bedeutungen, aber in den meisten beobachteten Fällen wird ein und dasselbe Zeichen (ASCII Zeichen an der Position 45) überladen. Diese Doppeldeutigkeiten treten bei vielen punktuationalen Markup-Zeichen auf und können zu Fehlern bei der Tokenisierung führen. Um das zu illustrieren, vertiefe ich ein Beispiel aus einer Magisterarbeit, in der zum Aufbau eines IR-Systems Texte tokenisiert werden [96]. Dort wird vorgeschlagen, einen (einzelnen) Schrägstrich als Whitespace zu verwenden, außer bei Abtrennung von Einzelbuchstaben und Beteiligung von Zahlen. Wende ich diese Spezialisierungsregel auf meinen Korpus an, steigere ich den Recall um ca. ein Promille durch Zerlegung von z. B. `utopia/dystopia` oder `künstlerisch/kulturelle`. Allerdings senke ich auch meine Präzision deutlich, da etwa die Hälfte der Tokens mit Schrägstrich zu Unrecht zerlegt werden (z. B. `Zwangsarbeiter/in` und `Mitarbeiter/innen`). Die Möglichkeit, die Einzelbestandteile als korrekte, normale Tokens zuzulassen, würde ein so aufgebautes Lexikon mit allerlei Prä/Suffixen o. ä. verschmutzen, da man in Texten durchaus solche Schreibweisen findet; Fundstücke aus dem Web, die auch mehrfach auftreten sind z. B. `be/entladen`, `Männchen/lein` oder sogar `Konditor/inn/en`. Im Korpus taucht der Schrägstrich außerdem noch in Web-Adressen auf, die ebenfalls nicht zerlegt werden sollten. Der Einwand, dass die korrekte Schreibweise von den o. g. `Prä/Suffixen` eigentlich `Prä- und Suffixen` wäre, und damit mein einfacher Normalitätstest auch zu einem falschen Resultat führen würde, ist kein Gegenargument zur gezeigten Lösung, sondern unterstreicht die Tatsache, dass im Ausbau der Referenzimplementierung für jedes punktuationale Zeichen eine umfassende Gebrauchsstudie notwendig ist.

- **Sonderzeichen**. Obwohl ich in meinem Testkorpus kein Dokument aufgenommen habe, das ein Zeichen außerhalb der bisher genannten Klassen enthält, führe ich eine Restklasse aller anderen Zeichenklassen ein, wie z. B. mathematische Symbole, Währungszeichen, Icons, Korrekturzeichen, Musiknotationen, etc. und schwer zu klassifizierende Einzelzeichen wie z. B. die Zeichen für Copyright, Temperaturgrade, S-Bahn, etc. Es ist offensichtlich, dass es auf Grund der unübersichtlichen Menge sinnvoll ist, Sonderzeichen nur bei anwedungsspezifischen Bedarf in die Definition normaler Tokens mit aufzunehmen.

Länge normaler Tokens Der Begriff `Hepatozyten-Suspensions-Modellsystem` ist mit 36 Zeichen das längste textuelle Token aus meinem Korpus. Einerseits sind durch die Komponierbarkeit von Nomen beliebig lange Tokens denkbar, andererseits liegt die Vermutung nahe, dass die Länge textueller Tokens gut durch eine Poisson-Verteilung approximiert werden kann. Das heißt, je länger ein Token ist, desto unwahrscheinlicher ist es, das Token in einem Lexikon zu finden, Lexika – zumindest ursprünglich – auch immer Dokumentsammlungen entspringen. In über $3 \cdot 10^9$ indexierten Web-Seiten bei Google ebenso wie bei AllTheWeb ist derzeit[2] kein Dokument auffindbar, das den Begriff enthält. Es ist auch zu beachten, dass zu lange Tokens den Lexikonaufbau unnötig aufblähen können. Daher wurde die Längenbeschränkung normaler (= 64) Tokens empirisch aber großzügig auf die vorhandene Systemtechnik abgestimmt. Vermutlich ließe sich der Wert bei gleich bleibender Qualität der Nachkorrektur (im Sinne einer Ingenieursarbeit) noch deutlich drosseln.
Auch die Untergrenze der Länge normaler Tokens ist diskussionswürdig. Textuelle Tokens, die nur aus einem einzigen Zeichen bestehen, als OCR-Fehler zu enttarnen, halte ich für fast aussichtslos, da sie zu häufig auftreten, als präsentationales Markup (Aufzählungszeichen), Variablen, Abkürzungen, Maßeinheiten, Wörter, etc. In meinem Testkorpus findet man z. B. von A bis Z alle Einzelzeichen als textuelle Tokens, die in diesem Falle zum großen Teil aus abgekürzten Vornamen stammen. Alle klassischen Frequenz- und Abstandsmaße würden bei der OCR-Fehlererkennung versagen, einzig eine fortgeschrittene Kontextanalyse könnte ein wenig helfen. Von den Tokens der Länge zwei (Bigramme) findet man hingegen bei weitem nicht alle Ausprägungen im Testkorpus. Um mir einen Überblick über die Verteilung der Bigramme zu verschaffen, habe ich meine Untersuchung auf die indexierten Webseiten einer Suchmaschine (AllTheWeb) ausgedehnt. Da Groß- und Kleinschreibung sowie Umlautkodierungen im Web und dessen Indexierungen i. Allg. (noch) nicht richtig funktionieren, habe ich mich auf die Zeichen a-z beschränkt und für jedes der $26^2 = 676$ Bigramme ermittelt, auf wie vielen Web-Seiten es auftaucht.

Die gleiche Untersuchung habe ich außerdem für alle $26^3 = 17576$ Trigramme durchgeführt.

Die Tatsache, dass im Index von AllTheWeb die Trigramm-Frequenz bei 10^3 und die Bigramm-Frequenz sogar bei 10^5 beginnt, ist ein Argument, auch diese Tokens nicht in die Reihe normaler Tokens mitaufzunehmen. Schaut man sich aber die beiden Verteilungsfunktionen genauer an, fällt auf, dass beide Male im obersten 10%-Quantil die Frequenz extrem ansteigt. Stichproben belegen die Vermutung, dass in diesem Bereich die natürlichsprachlichen Wörter und allgemein bekannte Abkürzungen versammelt sind, während sich im unteren Bereich mehrfach überladene Akronyme finden, die aber jeweils einer sehr engen Anwendungsdomäne zugeordnet werden können. Solch ein Trigramm wird scherzhaft als TLA (*three letter acronym*) bezeichnet. Da ich in meiner Arbeit u. a. die Zusammenstellung domainspezifischer Lexika untersuche, habe ich mich entschieden, auch Bi- und Trigramme in der lexikalischen Nachkorrektur

[2] obwohl das Korpus-Dokument zu einem früheren Zeitpunkt aus dem Web geholt wurde.

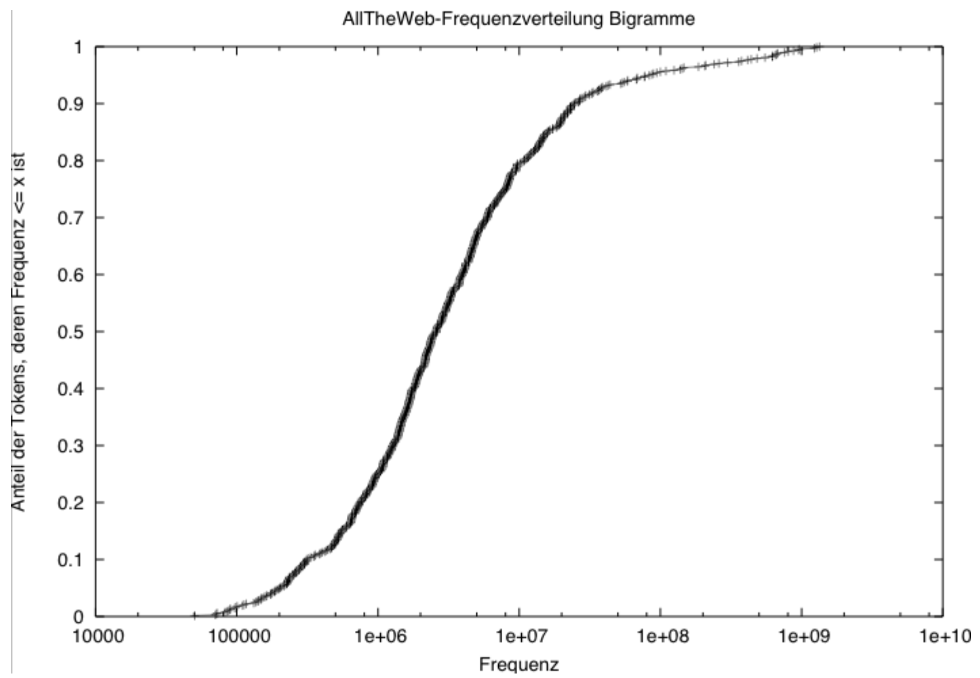


Abbildung 3.3: Frequenz-Verteilung der Bigramme in AllTheWeb.

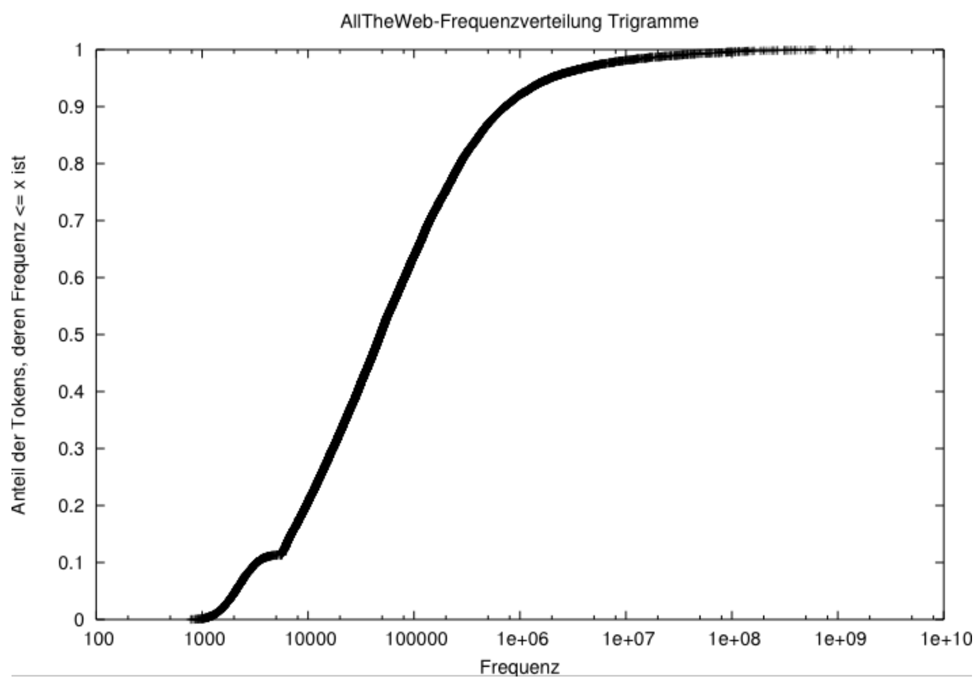


Abbildung 3.4: Frequenz-Verteilung der Trigramme in AllTheWeb.

zu untersuchen.

Trainings- und Testbetrieb

Im Trainings- und Testbetrieb genügt es nicht, eine Liste der Tokens der OCR-Ausgabe einzulesen, sondern man braucht auch die zugehörigen Tokens der Groundtruth, damit man permanent Überwachen kann, wie erfolgreich die Parametereinstellungen der Nachkorrektur sind.
Dazu habe ich ein Subinterface CoTokenizer eingeführt (siehe weiter unten). In der Methode `setGroundtruthFile()` werden die zugehörigen Tokens aligniert; außerdem ändern sich die Implementierungen der Methoden `getText()`, `getPreTextualTokens()`, `getTextualTokens()` und `getQueryTokens()`. Alle diese Methoden geben keine Liste von Tokens mehr zurück, sondern eine Liste von Doppellisten. Jeder Eintrag besteht aus einer linken und einer rechten Liste. Die linke Liste enthält Tokens der OCR und die rechte Liste enthält die zugehörigen Tokens der Groundtruth. Im Normalfall enthält die linke wie die rechte Liste genau ein Token. Doch Fehler der OCR-Engine können dazu führen, dass diese Zuordnung nicht klappt. Die OCR-Engine kann textuelle Tokens spalten, verschmelzen, verschlucken, oder auch hinzuerfinden. Die Algorithmen zum Aufbau dieser Doppelliste sind in ein eigenes Kapitel ausgelagert (siehe **??**), da sich die Aufgabe der Alignierung beim parallelen Einsatz mehrerer OCR-Engines erneut stellt.
In der Methode `getQueryTokens()` ergeben sich einige Änderungen und Anpassungen. Wie soll man in der Trainings- und Testphase z. B. mit verschmolzenen Wörtern verfahren? Fraglich ist, ob man nur die Tokens betrachten soll, von denen man schon vorher weiß, dass sie theoretisch korrigiert werden können, oder soll man die Philosophie verfolgen, auf allen Tokens zu trainieren und zu testen, so wie sie der lexikalischen Nachkorrektur in der Produktionsphase vorgelegt würden. Mit einer Analogie aus dem Alltag möchte ich meine Entscheidung für die erste Variante untermauern. Wer empirisch die Langzeitlagerung von Wein verbessern will, wird sicher Essigflaschen aus seiner Beobachtung ausschließen, da diese keinen Aufschluss über die Einflussparameter geben werden. Da von der OCR geerbte Fehler schon beim Tokenisieren ermittelt werden können und das Nachkorrekturmodul schlank gehalten werden soll, werden die Fehlerzahlen in Form von Integer-Werten vom Tokenisieren an die Nachkorrektur übergeben. Alle Arten von Alignierungsfehlern werden in der Methode `getQueryTokens()` von der Rückgabeliste ausgeschlossen. Außerdem werden nur textuelle Tokens in die Rückgabeliste übernommen, bei denen sowohl das OCR-Token als auch das zugehörige Groundtruth-Token normal sind. Im Falle, dass beide non-normal sind, ist die Entscheidung trivial. Falls eines der beiden Tokens normal und eines non-normal ist, liegt sicher ein Erkennungsfehler vor. Ist das Groundtruth-Token das non-normale, kann solch ein Fehler nie korrigiert werden, da in Lexika nur normale Tokens verzeichnet sind. Ist das OCR-Token das non-normale, wäre es inkonsequent, eine Ausnahme für die Anfragestrategie zu machen, die im Produktionsbetrieb nicht erkennbar wäre. Darüber hinaus gibt es weitere textuelle Tokens, die von der Trainingsphase ausgeschlossen werden sollen. Um eine

Fehladaption des Systems zu vermeiden wurde ein Mechanismus geschaffen, um Fehler aus folgenden Klassen kennzeichnen zu können:

- **Fehler im Ausgangsdokument**. Orthographische Fehler stehen beispielsweise außerhalb der Verantwortung der OCR-Engine und sind schon in der Groundtruth falsch enthalten. Schließt man solche Tokens nicht von der Testphase aus, erntet die lexikalische Nachkorrektur schlechte Ergebnisse für eine berechtigt durchgeführte Korrektur. Diese Klasse von Fehlern kann man nicht immer vernachlässigen. So kommen z. B. im deutschen Korpus zum Thema Fische auf zwei OCR-Fehler ein Rechtschreibfehler und in einem englischen Teilkorpus zum Thema Holocaust hat sich systematisch eine Umlautproblematik eingeschlichen: alle dort auftauchenden deutschen Eigennamen und Fachbegriffe werden durch die nichtumgelauteten Vokale ersetzt (`Goring` anstatt `Göring` oder `Gruppenfuhrer` anstatt `Gruppenführer`).

- **Wortspiele**. Der deutschen Korpus zum Thema Botanik enthält z. B. die Rhetorische Frage `Biologie - Leit- oder Leidwissenschaft zum Ende unseres Jahrhunderts?`. Das Wortspiel `Leidwissenschaft` taucht in je über $3 \cdot 10^9$ indexierten Web-Seiten bei Google und AllTheWeb kein weiteres mal auf, geschweige denn in einem am Institut verfügbaren, statischen Lexikon und ist daher für jede lexikalische Nachkorrektur ein kaum zu beseitigender Stolperstein.

- **Technische Störeffekte**. Bei der Korpuserstellung sind diverse Störeffekte beim Ausdruck einiger elektronischer Formate aufgetreten: z. B. unerwünschte Datums- und Dateiangaben wurden am Rand gedruckt oder der Schriftzug `Image` erschien an Stelle eines Bildes. Ohne Vorkehrungen kann das zu pathologischen Verhalten führen. Das Wort `Image` wird z. B. zum höchstfrequentesten Wort in einem deutschen Text, was wiederum zu einer falschen Adjustierung der Parameter in der Trainingsphase sowie nicht überzeugenden Statistiken in der Testphase führen kann. Alternativ zur Kennzeichnung und folglichen Nichtbeachtung dieser Tokens, kann man sie auch auf den Originaldokumenten entfernen. Da man Störeffekte meist erst in der Auswertungsphase realisiert, ist die (ebenfalls wissenschaftlich einwandfreie) Kennzeichnungsmethode sinnvoll, um nicht alle Experimente von Beginn an wiederholen zu müssen.

- **Fehlalignierungen**. Bei der Alignierung entstehen vereinzelt unzusammengehörige Wortpaare, die man erst in der Auswertungsphase entdecken kann und von weiteren Betrachtungen ausschließen möchte (siehe 21).

Die Kategorien lassen sich gut zusammenfassen, da sie alle die Eigenschaft haben, dass die Tokens

- nicht durch einen allgemeinen Algorithmus zu bestimmen sind, sondern eine explizite Aufzählung notwendig ist und

- ohne weitere Betrachtungen sowohl in der Test- als auch in der Trainingsphase am besten ausgefiltert werden.

Softwaretechnisch ist diese Filterung durch die Methode `isFiltered()` im Interface TokenProperties modelliert.
In der Testphase werden statistische Aussagen über die Leistungsfähigkeit des Nachkorrektursystems gemessen. Um mit anderen Systemen vergleichbar zu bleiben und um die Auswirkung des eigenen Eingriffs in Relation zum Gesamtprozess zu sehen, genügt es nicht, nur die Rückgabemenge der Methode `getQueryTokens()` zu betrachten. Je nach Definition des Gesamtprozesses, sind verschiedenste Kriterien noch miteinzubeziehen. Während in meiner Arbeit der Hauptaugenmerk auf Qualität des textuellen Inhalts liegt, fokussieren z. B. [91] auf die Qualität der Seitensegmentierung. Benchmarking von Systemen zur digitalen Dokumentenreproduktion ist ein eigener Forschungszweig. In den Jahren 1992 bis 1996 hat bspw. das Information Science Research Institute der Universität Nevada in Las Vegas jährlich einen umfassenden Testreport über die Genauigkeit diverser OCR-Engines erstellt [66].
Für diese Arbeit möchte ich mich für die Vergleichbarkeit auf den Bereich der Textnachkorrektur beschränken. In diesem Bereich ist ein beliebtes Verfahren, alle textuellen Tokens zu betrachten und Interpunktionsfehler nicht zu werten, da das genau den Ansprüchen von Information Retrieval Techniken genügt, einer wichtigen Anwendung von OCR. Diese sog. *word accuracy* entspricht in etwa meiner Definition *textueller Tokens*. Meine Definition von *Wort* ist enger als die im IR gebräuchliche Definition. Das folgende Beispiel soll die Motivation dafür verdeutlichen: Durch Kenntnis meiner Telefonnummer lässt sich meine Homepage mit Hilfe einer Web-Suchmaschine ($\hat{=}$ IR-System) auffinden. In ein Korrekturlexikon würde man die Nummer hingegen nicht aufnehmen. Daneben gibt es noch andere Genauigkeitsmaße für textuelle Nachkorrektur:

- *character accuracy*, der Anteil richtig erkannter Zeichen. Der Nachteil dieser Maßzahl ist, dass sie durch hohe Werte eine Genauigkeit suggeriert, die Menschen so nicht empfinden. Dazu ein Rechenbeispiel: Eine Buchseite habe 400 textuelle Tokens wovon 10 falsch geschrieben sind. Ein Mensch würde diese Seite sicher als stark fehlerbehaftet einordnen. Die word accuracy von 97,5% sieht schon sehr positiv aus; aber relativiert man die 10 Fehler zu 3232 Einzelzeichen sieht die character accuracy mit 99,7% fast perfekt aus (den Faktor beim Übergang von der Anzahl textueller Tokens zu allen Textzeichen habe ich mit Hilfe eines englischen Korpus mit mehr als 10^6 Zeichen ermittelt). Kommerzielle OCR-Engines werden gerne mit der character accuracy beworben.

- *phrase accuracy*, der Anteil korrekt erkannter Folgen textueller Tokens, die Folgenlänge ist dabei vorgegeben. Dieses Maß trifft weit besser die menschliche Intuition zum Fehlergrad, ist aber wenig verbreitet.

- *non-stop word accuracy*, eine Variante der word accuracy, die nicht bedeutungstragende Wörter wie Pronomen oder Artikel ausklammert. Dieses

Maß zielt speziell auf Information Retrieval Anwendungen und spielt eher eine untergeordnete Rolle.

(vgl. zu dieser Aufstellung [66])

Aus der Forderung, mit anderen OCR-Nachkorrektursystemen vergleichbar zu bleiben, ergeben sich eine Reihe von Fehlertypen, die meine lexikalische Nachkorrektur nicht korrigieren kann, aber die trotzdem statistisch erfasst werden. Die Fehlertypen im Einzelnen:

- Merge auf Wortebene. Man erkennt Verschmelzungen an folgender Charakteristik: die linke Liste eines Eintrags enthält ein Token und die rechte zwei oder auch mehr. Mit der Anzahl Verschmolzener Wörter könnte man eine noch feinere Untergliederung erstellen.
- Split auf Wortebene. Zersplitterungen weisen die umgekehrte Charakteristik von Verschmelzungen auf. Die linke Liste enthält zwei oder mehr Tokens. Auch diesen Fehlertyp kann man noch feiner zergliedern.
- Deletion auf Wortebene. Beim Eintrag einer Auslassung ist die linke Liste ist leer und die rechte Liste enthält ein Token.
- Insertion auf Wortebene. Einfügungen sind invers zu Auslassungen. Die linke Liste enthält ein Token und die rechte Liste ist leer.
- Andere Fehlalignierungen. Das sind bspw. Alignierungen, die sich nicht weiter auflösen lassen wie: `["aa", "bb", "cc"], ["aab", "bcc"]`
- OCR-Fehler, die ausgefilterte Tokens betreffen, wie bspw. Wortschöpfungen.
- OCR-Fehler, bei denen das zugehörige Groundtruth-Token nicht normal ist. Dabei ist egal, ob die OCR ein normales Token erkennt oder nicht.
- Normale Tokens, die von der OCR als non-normale erkannt werden.

Im Subinterface `CoTokenizer` ist eine Methode `errorCount()` vereinbart, die je nach übergebenen Fehlertyp die absolute Anzahl angibt. Relative Fehlerquoten lassen sich daraus mit Hilfe der Methode `getTextualTokenCount()` errechnen.

```
public interface CoTokenizer  extends Tokenizer {

static final int MERGE = 0;
static final int SPLIT = 1;
static final int DELETION = 2;
static final int INSERTION = 3;
static final int OTHER_ALIGNMENT_ERROR = 4;
static final int FILTERED_ERROR = 5;
```

```
static final int NON_NORMAL_ERROR = 6;
static final int OCR_TO_NON_NORMAL = 7;

int getErrorCount(int type);
int getTextualTokenCount();
void setGroundtruthFile(File groundtruthFile);

}
```

3.2.2 Ausgabe der lexikalischen Nachkorrektur

Datenstruktur

Für den Trainings- und Testbetrieb ist eine Ausgabe konkreter Korrekturanweisungen nachrangig, da im Vordergrund die Feineinstellung der Nachkorrekturparameter steht. Für diese Adjustierung wird innerhalb der Nachkorrekturkomponente die Güte der Korrekturvorschläge permanent für alle Parametereinstellungen neu berechnet.
Für den Produktionsbetrieb hingegen stehen Vorschläge zur Nachkorrektur im Zentrum. Im Sinne der geforderten Komponierbarkeit ist die gewünschte Ausgabe ein korrigiertes Textdokument gleichen Formats wie die OCR-Ausgabe. Es ist aber sinnvoll eine weitere Zwischenschicht abzutrennen. Der Kern der lexikalischen Nachkorrektur liefert für das Ausgangsdokument lediglich eine Liste von Korrekturvorschlägen mit Bewertungen. Diese können dann automatisch ausgeführt werden oder an eine interaktive Korrektur übergeben werden. Auch andere Anwendungen sind mit einer Korrekturvorschlagsliste denkbar, z. B. wird in [36] vorgeschlagen, für IR-Anwendungen auf OCR-Texten Korrekturen nicht auszuführen, sondern alle Korrekturvorschläge im Index mit zu berücksichtigen.

In der Nachbehandlung von OCR-Ergebnissen wird zwischen Erkennungs- und Korrekturverfahren unterschieden ([45]). Durch die verschiedenen Einsatzmöglichkeiten der Ausgabedatenstruktur wird die Entscheidung für eines der beiden Verfahren aus der Kernkomponente der Software herausgelöst. Damit die Datenstruktur sprach- und plattformübergreifend weiterverwendet werden kann, wurde eine XML-Schnittstelle geschaffen, die folgender DTD gehorcht:

```
<!ELEMENT CF (record+)>
<!ELEMENT record (cands)>
<!ATTLIST record
wOCR CDATA #REQUIRED
addr CDATA #REQUIRED
<!ELEMENT cands (cand*)>
<!ELEMENT cand EMPTY>
<!ATTLIST cand
vCand CDATA #REQUIRED
conf NMTOKEN #REQUIRED>
```

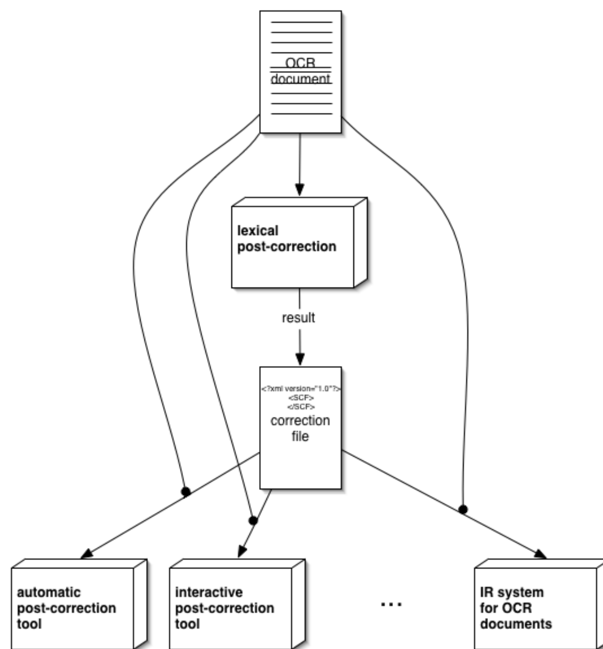


Abbildung 3.5: Zwischenschicht der lexikalischen Nachkorrektur.

Das Korrekturfile (Wurzelelement `CF`) besteht aus einer Liste. Für jedes normale OCR-Token (d. h. aus der Rückgabemenge der Methode `getQueryTokens()`) ist ein Element vom Typ `record` in der Liste enthalten. Das OCR-Token selbst ist als Attribut `wOCR` realisiert. Da meine Normalitätsdefinition relativ eng gefasst ist und keine in XML reservierten Zeichen zulässt, ist die Platzierung innerhalb eines Attributs ohne weitere Vorkehrungen möglich. Erweitert man jedoch die Definition dahingehend, dass z. B. auch **can't** als normal gewertet wird, muss an Stelle des Apostrophs mit Escape-Sequenzen gearbeitet werden, um dessen Sonderbedeutung in XML zu umgehen. Das Attribut `addr` enthält den in 3.1.2 diskutierten Adressierungsmechanismus. Innerhalb des Attributs wird die sgrep-Syntax für die Adressierung verwendet, also z. B. `addr="(0, 12)"`. Diese Informationen in *einem* Attribut zu organisieren hat den Vorteil, dass der Adressierungsmechanismus jederzeit ausgewechselt werden kann und die Nachkorrekturkomponente davon unberührt bleibt. Der Nachteil ist, dass für weiterverarbeitende Komponenten ein XML-Parser nicht alle Arbeit übernehmen kann, sondern die Syntax des Attributwerts gesondert geparst werden muss. Jedem `record` ist eine Korrekturkandidatenliste zugeordnet. Findet die Nachkorrektur zu einem OCR-Token keine Korrekturkandidaten, kann die Liste `cands` auch leer sein. Eine leere Liste bedeutet, dass in den verwendeten Lexika der Korrekturkomponente weder das OCR-Token, noch ein ähnliches Wort gefunden wurde. Andererseits ist das OCR-Token auch selbst immer in der Kandidatenliste eingereiht, sobald es in den eingesetzten Lexika vorhanden ist. Ein Korrekturkandidat `cand` enthält in Form von Attributen das vorgeschlagene Wort (`vCand`) und einen Konfidenzwert (`conf`). Der Konfidenzwert $conf(v^{Cand})$ definiert eine totale Ordnung auf allen Korrekturkandidaten, die widerspiegelt, wie sicher sich die Nachkorrekturkomponente bzgl. der Ausführung eines Korrekturvorschlags ist. Zu jeder nichtleeren Kandidatenliste lässt sich also ein Spitzenkandidat v^{cand}_{max} mit maximalem Konfidenzwert bestimmen. Der maximalen Konfidenzwert dieses Spitzenkandidat bestimmt den Konfidenzwert eines OCR-Tokens $conf(w^{ocr}) = conf(v^{cand}_{max})$. Ist die Kandidatenliste leer, wird $conf(w^{ocr}) = 1$ gesetzt. Durch $conf(w^{ocr})$ wird eine weitere totale Ordnung auf allen OCR-Tokens definiert. Für die Konfidenzwerte gilt:
$conf > 1$ falls die Ausführung des Korrekturvorschlags empfohlen wird
$conf = 1$ falls die Nachkorrekturkomponente unentschlossen ist, d. h. keine Meinung hat
$conf < 1$ falls von der Ausführung des Korrekturvorschlags abgeraten wird
Es ist auch denkbar, dass zu einem OCR-Token zwei Korrekturvorschläge angeboten werden, deren Konfidenzwert größer eins ist.

Anwendung

Mit dieser Datenstruktur lässt sich eine automatische Korrektur direkt umsetzen. Für alle OCR-Tokens mit $conf(w^{ocr}) > 1$ wird der maximale Vorschlag automatisch ausgeführt.
Aus der Datenstruktur lassen sich auch folgende drei Hauptvarianten einer in-

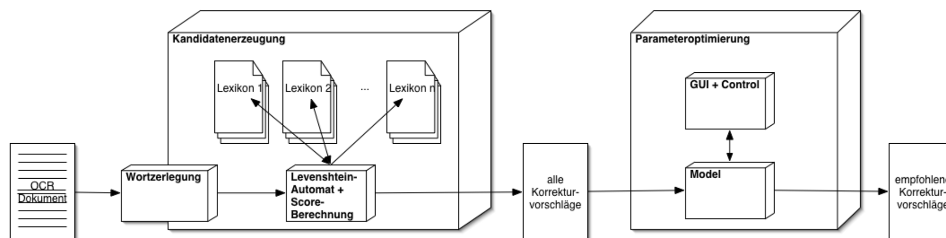


Abbildung 3.6: Software-Architektur.

teraktiven Korrektur realisieren:

1. **Inspektion aller unsicheren OCR-Tokens**. Erst wird eine untere Schranke des Vertrauens σ_{conf} festgelegt. Bis zu dieser Schranke wird eine automatische Korrektur ausgeführt. Anschließend werden einem Korrektor alle OCR-Tokens mit $conf(w^{ocr}) > \sigma_{conf}$ präsentiert. Zu diesen wird jeweils die Kandidatenliste zur Korrekturauswahl sowie eine manuelle Korrekturmöglichkeit angeboten. Werden beispielsweise alle Tokens mit Konfidenzwert $\leqslant 1$ inspiziert, wird die maximale Qualität der automatischen Korrektur ausgeschöpft und zusätzlich werden Fehler der Klassen *too cautious*, *no chance I* und *wrong candidate and threshold* überprüft.

2. **Interaktive Korrektur mit fester Korrekturmenge**. Im Unterschied zur ersten Variante wird die Vertrauensgrenze nicht direkt vorgegeben, sondern aus der Korrekturmenge zurückgerechnet. Zuerst wird eine feste Anzahl von n OCR-Tokens, ein Prozentsatz aller Tokens oder eine feste Korrekturzeit vereinbart. Die Liste aller OCR-Tokens wird nach $conf(w^{ocr})$ sortiert und dem Korrektor werden die obersten n OCR-Tokens mit Kandidatenliste und manueller Korrekturmöglichkeit präsentiert, der Rest wird automatisch korrigiert.

3. **Visualisierte, interaktive Korrektur**. Ein Mensch kann wichtige und kritische Dokumentteile wie z. B. Kundennamen oder klein gedruckte Bankverbindungen in einem Geschäftsbrief durch visualisierte, prozedurale Informationen wie Positionen oder Hervorhebungen besser überblicken. Daher ist es vorteilhaft, wenn ein Korrektor das Dokument möglichst detailgetreu sieht. Die Korrekturvorschläge (analog zu den beiden vorherigen Punkten) sollen als zusätzliche visuelle Information innerhalb des Dokuments auftauchen, z. B. durch farbige Unterlegung. Dazu muss in der Regel das OCR-Ausgabeformat dahingehend erweitert werden und ein eigenes Rendering-Tool dafür entwickelt werden.

3.3 Software-Architektur

3.3.1 Zweiteilung

Intern ist mein Nachkorrektursystem im Wesentlichen zweigeteilt in

- eine Komponente zur Erzeugung von Korrekturkandidatenlisten sowie
- eine visualisierte Komponente zur Parameteroptimierung.

Die beiden Komponenten werden in den nachfolgenden Abschnitten näher vorgestellt.
Mehrere Gründe motivieren die Zweiteilung der Architektur:

- **Performanz**. Die Anfrage an die Lexika zur Erzeugung der Kandidatenlisten ist der performance-kritische Flaschenhals des Systems und muss daher auf einem leistungsstarken Rechner laufen. Die Parameteroptimierung hingegen läuft auch auf meinem Notebook.

- **Unterschiedliche Implementierungssprachen**. Der Kern der Komponente zur Erzeugung der Kandidatenlisten ist eine Levenshtein-Automaten-Anfrage. Dazu ist eine Software im Projekt-Team vorhanden, die von Stoyan Mihov und Klaus U. Schulz entwickelt und implementiert wurde [72]. Um dem Leistungsflaschenhals zu begegnen, wurde dieser Kern in der systemnahen Sprache C belassen. Die visualisierte Komponente wurde u. a. auf Grund der Grafik-Bibliotheken in Java implementiert. Die Automatenanfragen sind rechenintensive, einmalige Vorberechnungen. Die graphische Parameteroptimierung ist hingegen ein Prozess, der schon mehrfach in Vorträgen vorgestellt wurde. Erst die Komponententrennung erlaubt diese Mobilität der GUI, da die plattformunabhängigkeit der Implementierungssprache Java erhalten bleibt.

- **Abtrennung der Lexika**. Lexika sind wertvolles, geistiges Eigentum, das man nur mit Vorsicht aus der Hand geben will. Die verwendeten statischen Lexika aus dem CISLex-Projekt sind z. B. das Ergebnis einer mehrjährigen Sammeltätigkeit am Institut. Durch die Zweiteilung konnte die graphische Software problemlos in der Lehre an Studenten weitergegeben werden. Gleiches gilt für mögliche zukünftige Schritte in Richtung einer Kommerzialisierung der Software. Eine denkbare Architektur wäre, serverseitig erzeugte Kandidatenlisten mittels Webservice auszuliefern.

3.3.2 Schnittstelle zwischen den beiden Komponenten

Die Schnittstelle zwischen den beiden Komponenten – die Übergabe aller Korrekturvorschläge – ist in XML realisiert. Zu jedem Lexikon wird ein SCF (single correction file) übergeben, das folgender DTD gehorcht:

```
<!ELEMENT  SCF (record+)>
<!ELEMENT  record (cands)>
<!ATTLIST  record
wOrig CDATA #REQUIRED
wOCR CDATA #REQUIRED
inLex (true|false) #REQUIRED
addr NMTOKEN #REQUIRED>
<!ELEMENT  cands (cand*)>
<!ELEMENT  cand EMPTY>
<!ATTLIST  cand
vCand CDATA #REQUIRED
score0 NMTOKEN #REQUIRED
score1 NMTOKEN #REQUIRED>
```

Diese SCF-Datenstruktur ist eine Liste von Kandidatenlisten. Neben der Kandidatenliste (`cands`) enthält jeder Eintrag (`record`) das OCR-Token (`wOCR`), das zugehörige Original-Token (`wOrig`), die Angabe, ob dieses lexikalisch (`inLex`) ist und den in 3.1.2 vorgestellten Adressierungsmechanismus (`addr`). Prinzipiell ist der Einsatz noch weiterer Scores neben `score0` und `score1` denkbar; die Beschränkung auf diese zwei Attribute entspricht lediglich meiner Umsetzung in der Referenzimplementierung.
Durch die Entscheidung, pro Lexikon ein eigenes SCF zu erzeugen, entstehen offensichtliche Redundanzen. Die Einzeldateien korrespondieren jedoch gut mit meiner intern verwendeten Datenstruktur und sind weitaus besser handhabbar bzgl. ihrer Größe als eine polyglotte Gesamtdatei. Ebenfalls um die Größe der einzelnen Dateien im Zaum zu halten, wurden alle Informationen in XML-Attributen und nicht auf textueller Ebene gespeichert, so dass End-Tags wegfallen. Die in der Evaluation eingesetzten SCFs sind trotz aller Sparanstrengungen noch zu groß, um sie mit der Standardbibliothek DOM zu erzeugen; der Hauptspeicherbedarf übersteigt deutlich die vorhandene Kapazität von 2GB.

3.4 Komponente zur Erzeugung von Korrekturkandidatenlisten

3.4.1 Aufgabe

Im Architekturbild ist die Wortzerlegung als Eingangstor dieser Komponente dargestellt. Sie ragt zum Teil aus der Komponente heraus, da für verschiedene OCR-Ausgabeformate verschiedene Filter für formales Markup implementiert werden müssen. Die Wortzerlegung wurde bereits bei der Beschreibung der Außenschnittstellen detailliert behandelt. Die Hauptaufgabe dieser Komponente ist, für jedes OCR-Token folgendes zusammenzustellen:

- **Korrekturkandidaten** eines Lexikons bzgl. eines Abstandsmaßes. Zur Bestimmung der benachbarten Tokens wird der klassische Levenshtein-

Abstand mit den Werten 0, 1 und 2 verwendet. Diese Abstandsberechnung bildet den ersten Grobfilter für die Bestimmung von Korrekturkandidaten. Mögliche Fehler, die durch diese Vorauswahl bedingt werden, fallen in die Klassen (3) und (4) meiner Fehlerklassifikation (siehe Kapitel 6). Der negative Effekt dieser Filterung fällt moderat aus, da im Korpus mehr als 95% der beobachteten Fehler innerhalb dieser Toleranzgrenze liegen.

- **Scores** zu jedem Korrekturkandidaten. In der Referenzimplementierung werden Scores zum Levenshtein-Abstand und zur Frequenz berechnet.

3.4.2 Lexikonanfrage

Lexikonorganisation

Ausgehend von einer Organisation als Wortliste werden die Lexika in einen minimalen deterministischen endlichen Automaten (DEA) überführt. Diese Datenstruktur kann man sich als Graphen vorstellen, dessen Pfade Lexikoneinträgen entsprechen. Daher sind Blätter des Graphen Endzustände des Automaten. Präfixe von Lexikoneinträgen, die selbst kompletten Einträgen entsprechen, wie z. B. `Bau` und `Baum`, bedingen weitere Endzustände innerhalb des Graphen. Ein minimaler DEA wird durch Zusammenfaltung gemeinsamer Suffixe gebildet. In [9] wird ein inkrementeller Algorithmus zur Berechnung eines minimalen DEA vorgestellt. In meiner Arbeit verwende ich direkt die zugehörige C-Implementierung. Vorteil dieser Datenstruktur ist die kompakte Darstellung selbst großer Lexika, so dass sie problemlos komplett im Hauptspeicher gehalten werden können Dadurch werden schnelle Suchzeiten im Millisekundenbereich ermöglicht. Für die in 4.2 genannten Lexika D^E mit $3.15 \cdot 10^5$ Einträgen und D^G mit $2.24 \cdot 10^6$ Einträgen ergeben sich Speichergrößen von 2,25 MB und 10,8 MB. Die Zeitkomplexität für den Aufbau eines minimalen DEA geben [9] mit $O(l \log n)$ an, wobei l die Anzahl aller Zeichen in der ursprünglichen Wortliste und n die Anzahl der Zustände des minimalen DEA bezeichnet. Auf meinem Arbeitsplatzrechner habe ich eine Aufbauzeit von 6 Sekunden für D^E und 82 Sekunden für D^G gemessen. Einmal aufgebaute DEA lassen sich persistent in ein File auslagern; die Ladezeit zurück in den Hauptspeicher beträgt ca. 200 Millisekunden.

Levenshtein-Automaten erzeugen Korrekturkandidatenlisten

Der naive Ansatz, alle benachbarten Korrekturkandidaten eines Lexikons bzgl. des Levenshtein-Abstands zu berechnen, indem man Eintrag für Eintrag im Lexikon nach dieser Bedingung prüft, scheitert an der linearen Komplexität in Kombination mit den verwendeten Lexikongrößen. Wie z. B. in [62] beschrieben wird an Stelle dessen die Suchrichtung umgedreht. Aus einem anzufragenden String werden alle Variationen im gewünschten Levenshtein-Abstand berechnet, in einer Korrekturkandidatenliste organisiert, im Lexikon angefragt und im Falle eines Mismatch wieder aus der Liste entfernt. In [72] wird eine äußerst effiziente Variante dieses Verfahrens beschrieben, die sich direkt auf die DEA-Organisation des

Lexikons stützt. Dazu wird aus dem Anfrage-String ein Levenshtein-Automat aller Strings zu einem vorgegebenen Abstand konstruiert und die Schnittmenge mit dem Lexikon-Automaten gebildet, indem beide Automaten parallel traversiert werden. Das Verfahren wurde von Klaus U. Schulz und Stoyan Mihov in C implementiert. In [72] wird die Leistung mit Hilfe eines bulgarischen Lexikons mit $8.69 \cdot 10^5$ Einträgen auf einem 500 MHz Pentium III getestet. Für alle im Lexikon enthaltenen Präfixe der Länge $3, 4, \ldots, 20$ wird jeweils die Korrekturkandidatenliste mit Levenshtein-Abstand $\leqslant 1$ berechnet. Annähernd unabhängig von der Länge des Eingabeworts bleibt die Suchzeit immer unter einer Millisekunde. In den Tests nimmt die Automatenkonstruktion deutlich mehr als die Hälfte der Anfragezeit ein. Da die Struktur der Levenshtein-Automaten nur von der Länge des Eingabeworts abhängt, kann mit generischen, vorberechneten sog. Template-Automaten die Konstruktionszeit noch weiter verkürzt werden. In meiner Arbeit verwende ich eine dahingehende Weiterentwicklung des Programms aus [72], inkl. der Möglichkeit, Anfragen mit Levenshtein-Abstand $\leqslant 2$ zu stellen. Die Autoren von [72] arbeiten derzeit an einer weiteren deutlichen Beschleunigung der Levenshtein-Automaten.

Einbindung der Levenshtein-Automaten

Ich habe folgendes Java-Interface definiert:

```
public interface CandidateGenerator {

static final String LEXICA_PATH = "./lexica/";

void setLexicon(String lexiconName);
String[] getCandidates();
}
```

Das Interface wird implementiert, indem die vorhandenen, in C verfassten Algorithmen zum Aufbau und Laden der Lexika sowie zur Erzeugung von Korrekturkandidatenlisten von den beiden Wrapper-Methoden `setLexicon()` und `getCandidates()` über das Java Native Interface (JNI) aufgerufen werden. Alternativ hätte man die gesamte Server-Komponente durchgehend in C programmieren können; die Einbindung in Java bringt jedoch eine Reihe von Vorteilen mit sich:

- **Einsatz von Klassen der Java-API**. Beispielsweise für den weiter unten vorgestellten Caching-Mechanismus konnte auf die Klasse `Hashtable` der API zurückgegriffen werden. Die Wahl begünstigt auch die Möglichkeit, die Server-Komponente in einer zukünftigen Ausbaustufe mit einer GUI auszustatten.

- **Wiederverwendung eigener Klassen**. Für die visualisierte Komponente zur Parameteroptimierung wurde bspw. eine wiederverwendbare Klasse

Teilkorpus	$t_{startup}$[s]	t_{total}[s]	$t/Token$[ms]	$\|Token\|$	$\|CandList\|$
Bauern	0,39	111,76	11,91	9387	187,48
Informatik	0,38	77,33	11	7033	145,74
Fische	0,38	86,36	10,37	8326	141,36
Holocaust	0,38	63,79	11,35	5619	160,04
Jagd	0,41	126,56	11,31	11188	169,72
Kochen	0,43	73,13	11,09	6595	162,78
Mykologie	0,38	54,04	10,02	5394	123,87
Neurologie	0,38	50,4	10,25	4919	124,19
Philosophie	0,38	126,32	11,51	10976	168,55
Rom	0,39	79,42	11,69	6794	166,07
Speisepilze	0,38	54,8	11,3	4849	153,07

Tabelle 3.1: Kandidatenbestimmung mit D^G.

entwickelt, die ein OCR-Token mitsamt seiner Kandidatenliste repräsentiert und eine Methode zur Ausgabe als XML-Element enthält.

- **Bessere Portabilität und Wartbarkeit**, da die Plattformabhängigkeiten der Software auf einen engen, algorithmischen Teil mit wenigen Systemaufrufen begrenzt wurde.

Die Methode `setLexicon()` verlangt als Eingabeparameter einen eindeutigen Lexikonnamen. Die Methode prüft zuerst, ob bereits eine kompilierte Version des Lexikons im Verzeichnis `LEXICA_PATH` vorhanden ist, d. h. ein File mit fertig aufgebautem DEA. Falls das Lexikon nur in Form einer Wortliste vorliegt, wird der DEA aufgebaut und für spätere Aufrufe in ein File gespeichert. Die beiden Dateitypen werden mit den Extensionen `wl` und `dea` unterschieden. Die Methode `getCandidates()` ruft den Levenshtein-Automaten so, dass alle Wörter des Lexikons mit Abstand $\leqslant 2$ durch ein Sonderzeichen abgetrennt, innerhalb eines Strings zurückgeliefert werden. Der C-String wird vor der Rückgabe in einen Java-String umcodiert. Die Zerlegung in ein Array erfolgt in der Java-Methode. Diese eigenverwaltete Datenstruktur bewährte sich besser als eine direkte Übergabe von String-Arrays. Die Performanz dieser Implementierung zeige ich mit einem Laufzeittest auf elf deutschen OCR-Teilkorpora mit dem deutschen Standard-Lexikon D^G und den jeweiligen Crawl-Lexika D^{GC}. Die Tests wurden auf einem vergleichbaren Rechner (Pentium III mit 450 MHz) wie die Originalexperimente von Mihov und Schulz durchgeführt. Weitere Tests auf einem Server-Rechner mit Dualprozessor-Board (2xPentium III mit 1,1 GHz) halbieren etwa die angegebenen Laufzeiten. Im ersten Laufzeittest mit D^G liegt das statische Lexikon schon als fertige DEA-Struktur vor und muss daher nur geladen werden.

Die gesamte Bearbeitungszeit t_{total} liegt zwischen ein und zwei Minuten, wobei die Ladezeit der DEA $t_{startup}$ vernachlässigbar ist. Die Unterschiede resultieren zum Großteil aus der unterschiedlichen Anzahl angefragter Tokens $|Token|$,

Teilkorpus	$t_{startup}$[s]	t_{total}[s]	$t/Token$[ms]	$\|Token\|$	$\|CandList\|$
Bauern	20,73	355,72	37,89	9387	837,45
Informatik	14,38	185,54	26,38	7033	578,89
Fische	22,32	213,76	25,67	8326	510,96
Holocaust	15,71	139,8	24,88	5619	532,22
Jagd	24,23	398,61	35,63	11188	763,81
Kochen	24,64	222,15	33,68	6595	706,38
Mykologie	13,23	130,55	24,2	5394	510,16
Neurologie	14,42	115,29	23,44	4919	478,51
Philosophie	22,8	393,21	35,82	10976	782,13
Rom	19,09	189,2	27,85	6794	563,27
Speisepilze	18,7	119,88	24,72	4849	475,37

Tabelle 3.2: Kandidatenbestimmung mit den Crawl-Lexika.

Teilkorpus	$t_{startup}$[s]	t_{total}[s]	$t/Token$[ms]	$\|Token\|$	$\|CandList\|$
Bauern	0,39	37,52	4	9387	187,48
Informatik	0,38	27,99	3,98	7033	145,74
Fische	0,39	32,39	3,89	8326	141,36
Holocaust	0,38	26,12	4,65	5619	160,04
Jagd	0,39	43,94	3,93	11188	169,72
Kochen	0,38	23,25	3,52	6595	162,78
Mykologie	0,4	24,67	4,57	5394	123,87
Neurologie	0,38	22,01	4,47	4919	124,19
Philosophie	0,38	41,32	3,76	10976	168,55
Rom	0,38	29,11	4,28	6794	166,07
Speisepilze	0,38	21,48	4,43	4849	153,07

Tabelle 3.3: Kandidatenbestimmung mit D^G mit Caching.

da die Bearbeitungszeit pro Token $t/Token$ nahezu konstant ist. Die durchschnittliche Länge der Kandidatenlisten $|CandList|$ liegt etwa bei 150.
Da die Crawl-Lexika nur für einen Teilkorpus eingesetzt werden, enthält beim zweiten Laufzeittest $t_{startup}$ auch die Kompilierzeit.

Die Längere Gesamtlaufzeit resultiert aber weniger aus der ca. 20 Sekunden langen Kompilierzeit, sondern eher aus den viermal so langen Kandidatenlisten. Die Häufigkeit der Levenshtein-Automaten-Anfragen kann mit einem Caching-Mechanismus gesenkt werden, da die Tokens in den Korpora nach dem Zipf'schen Gesetz [97] verteilt sind. Der Cache wurde in einfachster Weise mit einer Java-Hashtable implementiert. Schlüssel der Hashtable sind die Anfrage-Strings und Werte sind die Kandidatenlisten. Es wird immer zuerst nach einer Kandidatenliste in der Hashtable gesucht. Falls keine vorhanden ist, wird die Liste mit einer Automaten-Anfrage berechnet und in die Hashtable eingefügt.

Die Gesamtlaufzeiten reduzieren sich durch den Cache auf ca. ein Drittel.

Teilkorpus	$t_{startup}$[s]	t_{total}[s]	$t/Token$[ms]	$\|Token\|$	$\|CandList\|$
Bauern	20,97	90,64	9,66	9387	837,45
Informatik	14,75	55,02	7,82	7033	578,89
Fische	22,7	75,73	9,1	8326	510,96
Holocaust	16,07	53	9,43	5619	532,22
Jagd	24,54	103,68	9,27	11188	763,81
Kochen	25,05	73,13	11,09	6595	706,38
Mykologie	14,91	56,54	10,48	5394	510,16
Neurologie	16,56	52,91	10,76	4919	478,51
Philosophie	25,72	107,55	9,8	10976	782,13
Rom	21,63	72,13	10,62	6794	563,27
Speisepilze	23,91	56,64	11,68	4849	475,37

Tabelle 3.4: Kandidatenbestimmung mit den Crawl-Lexika mit Caching.

Auch für die Crawl-Lexika reduziert sich die Gesamtlaufzeit durch den Caching-Mechanismus auf etwa ein Drittel. Die Kompilierzeit der Lexika fällt kaum ins Gewicht.

3.4.3 Berechnung der Scores

In der Referenzimplementierung wurden die Familie der Levenshtein-Scores und der Frequenz-Score realisiert. In Kapitel 5 werden diese Scores zusammen mit genauen Formeln zur Berechnung vorgestellt. Im folgenden Abschnitt werden Implementierungsdetails diskutiert.

Levenshtein-Scores

Die Levenshtein-Scores werden direkt aus den Strings der Korrekturkandidaten und dem OCR-Token berechnet. Nur für gewichtete Levenshtein-Scores wird als weitere Information von außen eine Gewichtungsmatrix benötigt. Berechnung und Einbindung dieser Matrix sind ausführlich in [67] dargestellt.
Der Berechnungsaufwand der Levenshtein-Scores ist linear zur Anzahl der Kandidaten. Der Engpass ist dabei durch die Berechnung des Abstands bestimmt. Da bei der Levenshtein-Automaten-Anfrage der Abstand nicht explizit enthalten ist, muss er separat berechnet werden. In einer Messreihe mit den Levenshtein-Scores bzgl. der deutschen Crawl-Lexika auf einem Server-Rechner (2xPentium III mit 1,1 GHz) wird deutlich, dass die Berechnung erst in Kombination mit den extrem langen Kandidatenlisten zu Gesamtlaufzeiten bis zu einer Minute führen.

Die Berechnungszeit pro Kandidat liegt im Mikrosekundenbereich und etwa in gleicher Größenordnung wie die weiter unten angeführte Hash-Zugriffszeit für die Frequenz-Score-Berechnung. Durch Caching der Levenshtein-Distanzen kann keine Zeit gewonnen werden, da der Organisations-Overhead etwa gleichstark zu Buche schlägt. Ein Einsparpotential liegt dagegen beim Caching der

Teilkorpus	t_{total}[s]	$t/Token$[ms]	$t/Kandidat$[ms]
Bauern	56,22	5,995	0,007
Informatik	28,4	4,044	0,007
Fische	33,24	4	0,008
Holocaust	23,45	4,174	0,008
Jagd	64,33	5,752	0,008
Kochen	32,95	4,996	0,007
Mykologie	18,64	3,457	0,007
Neurologie	15,4	3,132	0,007
Philosophie	58,93	5,374	0,007
Rom	24,11	3,55	0,006
Speisepilze	14,98	3,095	0,007

Tabelle 3.5: Berechnungsaufwand der Levenshtein-Scores bzgl. der deutschen Crawl-Lexika.

gesamten Datenstruktur der Korrekturkandidatenliste, d. h. Automatenanfrage zusammen mit Score-Berechnung.

Frequenz-Score

Für die Berechnung des Frequenz-Scores wird zu jedem Lexikon eine Frequenzliste in eine Hashtable eingelesen. In den Frequenzlisten werden alle Einträge mit dem meistverbreiteten Frequenzwert (in der Regel 1) weggelassen und als Default-Wert bei erfolgloser Anfrage in der Hashtable gesetzt. Mit diesem einfachen Trick werden die Ladezeiten der Frequenzlisten auf ca. ein Drittel reduziert. Für die Frequenzlisten statischer Lexika liegt ein weiteres Reduktionspotential in einer Vorkompilierung, ähnlich zu den Lexika in DEA-Form. Es wurde erneut eine Messreihe bzgl. der deutschen Crawl-Lexika auf einem Server-Rechner (2xPentium III mit 1,1 GHz) vorgenommen.

3.5 Visualisierte Komponente zur Parameteroptimierung

Die Komponente zur Parameteroptimierung ist nach dem Design-Pattern Model-View-Control (MVC) aufgebaut.

3.5.1 Model

Das Modell der Komponente ist die Klasse `InvestigationList`, eine Subklasse von `ArrayList`. Die Einträge der Liste sind Objekte der Klasse `Record`, die einem OCR-Token entsprechen, zusammen mit Originalwort, Korrekturkandidatenliste, Adressierungsmechanismus und der Angabe, ob das Original in einem der Lexika enthalten ist. Dies entspricht dem XML-Element `record`, der

Teilkorpus	$t_{startup}$[s]	t_{total}[s]	$t/Token$[ms]	$t/Kandidat$[ms]
Bauern	7,6	18,35	1,955	0,002
Informatik	5,97	10,96	1,561	0,003
Fische	6,85	12,2	1,467	0,003
Holocaust	6,96	11,91	2,12	0,004
Jagd	8,37	20,76	1,855	0,002
Kochen	7,65	14,65	2,23	0,003
Mykologie	4	8,08	1,498	0,003
Neurologie	6,18	9,67	1,968	0,004
Philosophie	9,12	20,64	1,88	0,002
Rom	8,09	13,91	2,048	0,004
Speisepilze	7,37	11,3	2,333	0,005

Tabelle 3.6: Berechnungsaufwand der Frequenz-Scores bzgl. der deutschen Crawl-Lexika.

in 3.3.2 vorgestellten DTD, jedoch nicht nur für ein einzelnes Lexikon, sondern allgemeiner für eine Kombination von Lexika. Zu allen Elementen der DTD wurde eine korrespondierende Java-Klasse implementiert. Sortiert wird `InvestigationList` an Hand der kombinierten Scores jedes Records, indem `Record` das Interface `Comparable` implementiert. Die Klasse enthält neben der Liste auch eine Korrekturgrenze, die die Liste in einen aktiven und einen passiven Teil spaltet. Die Korrekturgrenze lässt sich mit einer Methode frei verschieben. Der Grenz-Score $score_{border}$ kann mit einer Methode berechnet werden. In Attributen von `InvestigationList` sind diverse Zähler organisiert. Gezählt werden die aktuelle Fehlerzahl zu jeder Klasse des Schemas aus 6.2 und die Anzahl normaler Original-Tokens, normaler OCR-Tokens sowie lexikalischer Original-Tokens. Neben der Auslesemöglichkeit der absoluten Zahlen sind Methoden zur Berechnung diverser Fehlerraten und der Coverage vorhanden. Die Fehlerzähler werden auch für die beiden Optimierungs-Methoden von `InvestigationList` verwendet. Eine erste Optimierung stellt unter gegebener Zusammensetzung der kombinierten Scores die Korrekturgrenze $score_{border}$ so ein, dass die Gesamtzahl der Fehler minimiert wird. Die zweite Methode findet eine optimale Gewichtung der einzelnen Scores, indem die erste Optimierungsmethode iteriert aufgerufen wird.
Der Konstruktor von `InvestigationList` verlangt eine Liste von SCF-Objekten. Die Klasse `SCF` liest die von der Komponente zur Kandidatenerzeugung übergebenen XML-Files ein. Eine neue Kombination von Lexika für die Nachkorrektur erfordert einen neuen Konstruktor-Aufruf von `InvestigationList`.

3.5.2 View und Control

View und Control fasse ich in einem Abschnitt zusammen, da die gesamte Steuerung des Modells über Listener erfolgt, die als innere Klassen implementiert wurden. An Stelle dessen teile ich in *Start* und *Interaktion* auf.

Start

Der Start der graphischen Nachkorrektur wird über eine XML-Konfigurationsdatei gesteuert, die folgender DTD gehorcht:

```
<!ELEMENT        config         (scfs,gui)>
<!ELEMENT        scfs           (scf)>
<!ATTLIST        scfs
  arity          NMTOKEN        #REQUIRED
  alpha          NMTOKEN        #REQUIRED
  scoreName0     CDATA          #REQUIRED
  scoreName1     CDATA          #REQUIRED>
<!ELEMENT        scf            EMPTY>
<!ATTLIST        scf
  name           CDATA          #REQUIRED
  germanStemmer  CDATA          #REQUIRED
  englishStemmer CDATA          #REQUIRED>
<!ELEMENT        gui            EMPTY>
<!ATTLIST        gui
  scoreVisible   CDATA          #REQUIRED
  listSaveVisible CDATA         #REQUIRED>
```

Die Liste `scfs` enthält alle zu ladenden Korrekturfiles der verschiedenen Lexika. Das Attribut `arity` begrenzt die Anzahl der Korrekturkandidaten zu einem OCR-Token. Diese Drosselung ist notwendig, da bei ungebremstem Import von Korrekturkandidaten der Hauptspeicher schnell überfüllt ist, ohne dass durch lange Kandidatenlisten ein echter Vorteil entsteht. Es wurden viele Kandidatenlisten mit mehr als 1000 Einträgen beobachtet, wobei der korrekte Vorschlag fast ausschließlich in der obersten Spitzengruppe rangiert. Zur Bestimmung der besten Kandidaten ist eine initiale Sortierung gemäß eines kombinierten Scores nötig, da der Levenshtein-Automat die Kandidaten lediglich alphabetisch geordnet ausgibt. Die Parameter der Sortierung werden über die Attribute `alpha`, `scoreName0` und `scoreName1` gesetzt.
Diese Selektion ist eine zweite Vorfilterung neben der Grobfilterung durch den Levenshtein-Automaten, der nur Kandidaten mit Abstand kleiner 2 liefert.
Die Platzierung dieser Vorfilterung in der interaktiven Komponente ist nur in der Prototypenphase sinnvoll. Dadurch muss bei wiederholten Experimenten mit der Vorfilterung nicht immer der Gesamtprozess durchlaufen werden, der pro Teilkorpus und Lexikon im Minutenbereich liegt. Für eine weitere Ausbaustufe der Software sollte ein empirisch ermittelter Wert in der Automaten-Komponente fest verankert werden, um Speicher- und Netzressourcen zu schonen.
Bei jedem Korrektur-File `scf` kann via Attribut angegeben werden, ob das in 4.4 vorgestellte Werkzeug zur Behandlung von Flexions- und Orthographievarianten verwendet werden soll. Die Diskussion über die richtige Platzierung verläuft analog zu der Diskussion über die Platzierung der Vorfilterung.

Die Graphische Oberfläche ist in mehreren Fenstern organisiert. Mit den Attributen des Elements `gui` lässt sich die Anzeige von Fenstern unterdrücken, die nicht in allen Anwendungsfällen benötigt werden.

Interaktion

Die Interaktions-GUI ist thematisch in vier Fenster unterteilt:

1. **Lexikonauswahl und Bestimmung des Korrekturgrenzpunkts**. Die Beschreibung findet man zusammen mit Screen-Shots in den Abschnitten 4.1.3 und 5.7.2.

2. **Score-Zusammensetzung**. Im Abschnitt 5.6.2 findet sich dazu eine Beschreibung sowie einen Screen-Shot.

3. **Fehlerstatistik**. Dieses Fenster zeigt diverse Statistiken wie die Coverage, den Anteil normaler Tokens sowie bei aktuell eingestellter Korrekturgrenze die Genauigkeit, die Fehleranzahl und deren Verteilung auf die verschiedenen Klassen des Schemas aus 6.2.

4. **Ausgabe**. Dieses Fenster bietet die Möglichkeit, unter aktuellen Einstellungen, einzelne Teile oder auch die gesamte Korrekturliste mit Konfidenzwerten in XML in ein File oder auf die Kommandozeile auszugeben. Damit lässt sich einerseits das Verhalten der Komponente zielgenau inspizieren und andererseits das Endresultat einer Nachkorrektur sichern.

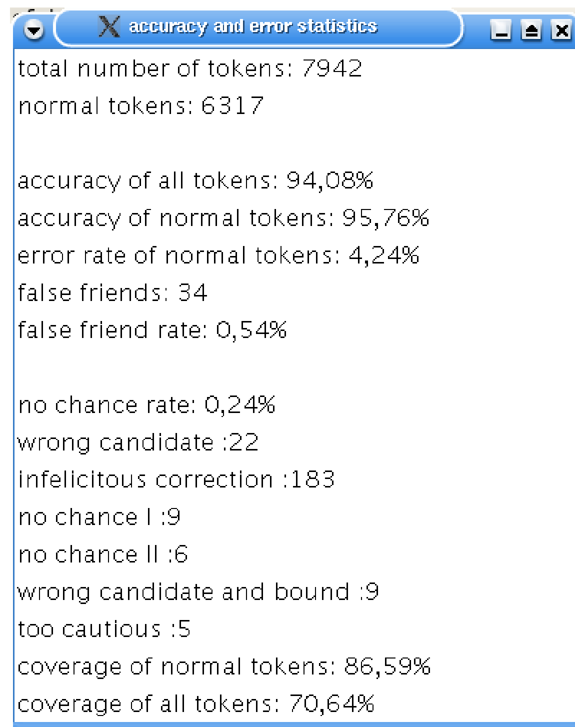


Abbildung 3.7: Fehlerstatistik.

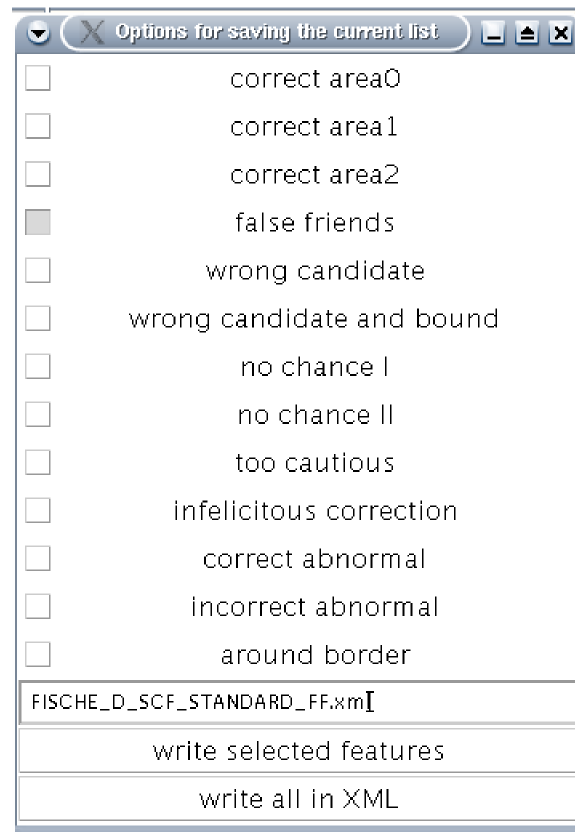


Abbildung 3.8: Ausgabe.

Kapitel 4

Lexika

Die Beschreibung der Komponente zur Erzeugung von Korrekturkandidatenlisten im vorangegangenen Kapitel schließt die technischen Details der Realisierung von Anfragen an ein Nachkorrekturlexikon ein, d. h. die Einbindung einer effizienten, extern entwickelten DEA-Datenstruktur zusammen mit einem Mechanismus zum approximativen String-Matching (siehe 3.4.2). In diesem Kapitel wird der Inhalt des Lexikons in den Mittelpunkt gerückt. Mit dem Begriff Lexikon bezeichne ich eine Wortliste, in der alle von einer OCR-Engine erkannten Wort-Tokens nachgeschlagen werden und aus der die Kandidatenlisten für eine Nachkorrektur gewonnen werden. In der einschlägigen Literatur zur OCR-Nachkorrektur werden die drei englischen Begriffe *word list*, *dictionary* und *lexicon* synonym verwendet (vgl. dazu den Überblicksartikel [45]). Zu Beginn wird in diesem Kapitel die geeignete Lexikongröße und -zusammensetzung diskutiert. Anschließend werden die eingesetzten Lexika besprochen. Neben den statischen Lexika werden vor allem Methoden zum dynamischen Aufbau von Lexika aus Web-Dokumenten vorgestellt. Ausserdem werden Techniken beschrieben, die fehlende Varianten im Lexikon berücksichtigen. Das Kapitel schließt mit der Einführung eines sog. perfekten Lexikons, einer theoretischen Obergrenze der lexikalischen Nachkorrektur.

4.1 Motivation zur Verwendung mehrerer Lexika

4.1.1 These: ein kleines Lexikon soll es sein

Ein Teil der Forschergemeinde vertritt die These, dass das Nachkorrekturlexikon möglichst klein sein soll, da dadurch die Wahrscheinlichkeit sinkt, dass ein falsch erkanntes Wort unentdeckt bleibt, da es ebenfalls im Lexikon enthalten ist. Je größer das Lexikon, desto größer die Anzahl falscher Freunde (*false friends*). Diesen offensichtlichen Zusammenhang habe ich mit Hilfe meines Korpus untermauert. Dazu habe ich die deutschen Groundtruth-Teilkorpora aus

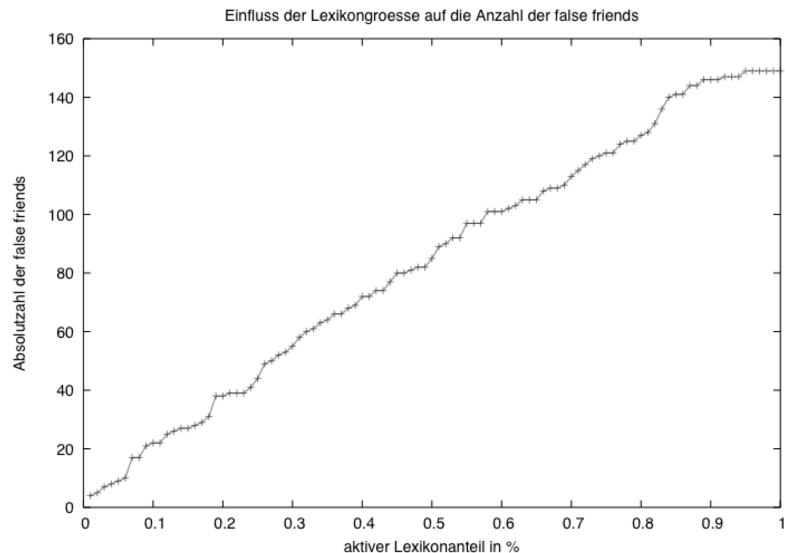


Abbildung 4.1: Einfluss der Lexikongröße auf die Anzahl der false friends.

sieben Themengebieten mit insgesamt 49 306 normalen Tokens herangezogen, inkl. mehrfach auftretender Tokens. Als Lexikon habe ich ein Standardlexikon des Deutschen mit 2 235 136 Wort-Tokens gewählt. Das Lexikon stammt aus dem CISLex-Projekt und steht institutsintern zur Verfügung. Zusammen mit einem OCR-Erkennungslauf habe ich die falschen Freunde bzgl. des Lexikons extrahiert, insgesamt 149 Stück. Um die Anzahl falscher Freunde mit der Lexikongröße in Relation setzen zu können, habe ich das Lexikon randomisiert auf 99%, 98%, 97%, etc. reduziert und dabei die Abnahme falscher Freunde beobachtet.

In dieser Untersuchung lässt sich der Kurvenverlauf durch eine Gerade mit Steigung 1 gut approximieren.
In [63] findet man eine andere Modellrechnung, die die Gefahr falscher Freunde aufzeigt. Innerhalb eines Lexikons mit $3,5 \cdot 10^5$ Wörtern sind im Schnitt ein halbes Prozent aller Strings mit Levenshtein-Abstand 1 zu einem Wort selbst wieder im Lexikon enthalten. Da andere Untersuchungen gezeigt haben, dass auch die meisten OCR-Fehler Abstand eins aufweisen, empfiehlt der Autor für die Nachkorrektur ein kleines Lexikon.

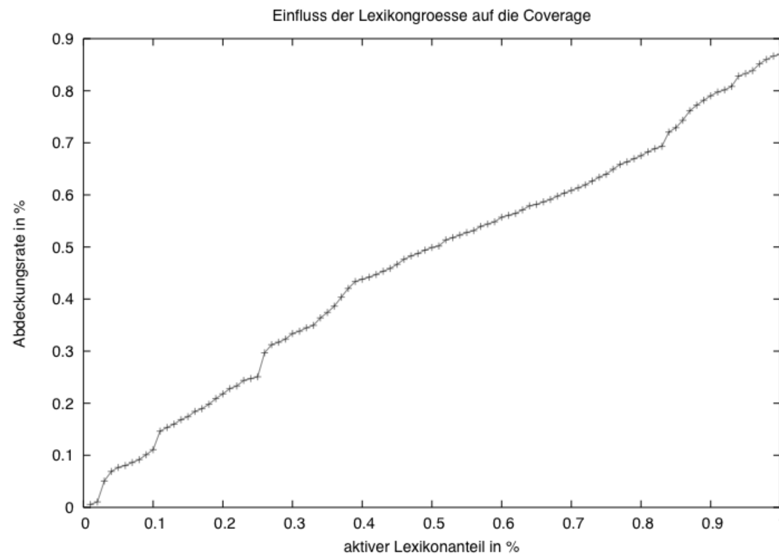


Abbildung 4.2: Einfluss der Lexikongröße auf die Coverage.

4.1.2 Antithese: ein großes Lexikon soll es sein

Ein anderer Teil der Forschergemeinde vertritt hingegen die These, dass das Lexikon möglichst groß sein soll, da dadurch die Wahrscheinlichkeit sinkt, dass ein Wort des Originaltexts nicht im Lexikon enthalten ist. Durch ein großes Lexikon wird vermieden, dass ein korrekt erkanntes Wort fälschlicherweise korrigiert wird, und es wird erreicht, dass zu möglichst jedem falsch erkannten Wort der richtige Korrekturkandidat in der Kandidatenliste enthalten ist. In meiner Fehlerklassifikation der Nachkorrektur deckt der rechte Ast meines Baumschemas die Fälle eines zu kleinen Lexikons ab.
Je größer das Lexikon, desto größer die Abdeckung (*coverage*). Um auch diese offensichtliche Aussage empirisch zu untermauern, habe ich mit dem o. g. Ausschnitt aus meinem Groundtruth-Korpus die Relation zwischen der Abdeckungsrate und der Lexikongröße unter einer schrittweisen, randomisierten Reduktion des Lexikons untersucht. Das Gesamtlexikon deckt 87,03% aller Groundtruth-Tokens ab.

Auch diese Kurve verläuft nahezu linear. Die wenigen Sprungstellen in der Abdeckung werden durch Hinzunahme bzw. Auslassung topfrequenter Wörter wie `die` verursacht. Der Einwand, dass bei einem Ausdünnen des Lexikons gemäß der Seltenheit der Wörter die Kurve anders verlaufen würde ist berech-

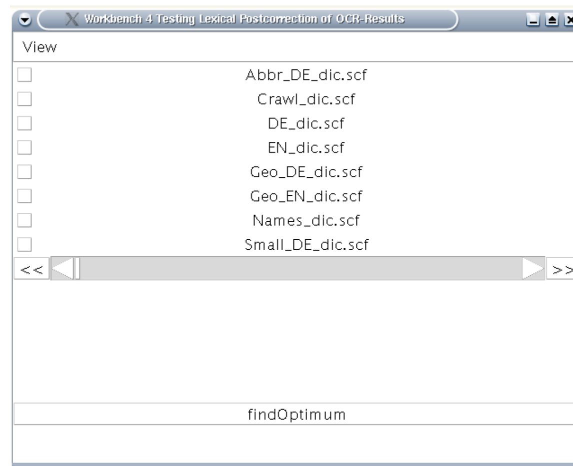


Abbildung 4.3: GUI zur Kombination von Einzellexika.

tigt; schon ein kleines Kernlexikon hat eine große Abdeckung, die bei zunehmender Lexikonvergrößerung immer langsamer ansteigt. In [12] wird gezeigt, dass eine Lexikonvergrößerung im Bereich von $5 \cdot 10^5$ auf $6 \cdot 10^5$ unter Beachtung der Frequenz, einen fünfzigfach höheren Nutzen als Schaden verursacht. Dieser Einwand beweist aber keinesfalls, dass ein großes Lexikon in der Nachkorrektur überlegen ist, sondern zielt in die Richtung der folgenden Synthese.

4.1.3 Synthese: ein maßgeschneidertes Lexikon soll es sein

Da sowohl ein kleines als auch ein großes Lexikon gleichermaßen Probleme mit sich bringen, sind sich die meisten Forscher einig, dass die Wahl nicht direkt über die Größe zu treffen ist. Die Auflösung des Konflikts besteht darin, ein möglichst maßgeschneidertes Lexikon für die Nachkorrektur zu verwenden, das möglichst genau das Vokabular des zu korrigierenden Dokuments enthält (siehe [11] und [45]). Dazu werden in dieser Arbeit zwei Lösungsansätze präsentiert:

- Das Korrekturlexikon als Kombination mehrerer Speziallexika.
- Maßanfertigung eines Lexikons aus thematisch ähnlichen Dokumenten.

In der Nachkorrektur-Software habe ich die Möglichkeit geschaffen, diverse Einzellexika zusammenzuschalten.

Der Benutzer kann interaktiv eine geeignete Auswahl an statischen und auch dynamisch erzeugten Lexika treffen, ohne dass durch die Berechnungen eine merkliche Zeitverzögerung entsteht.
Treten in zwei oder mehr Lexika gleichlautende Korrekturvorschläge auf, werden diese zu einem Kandidaten reduziert. Für die Zusammenfassung der Einzel-Scores wird der jeweils vorliegende Maximalwert herangezogen. Dieses Kombi-

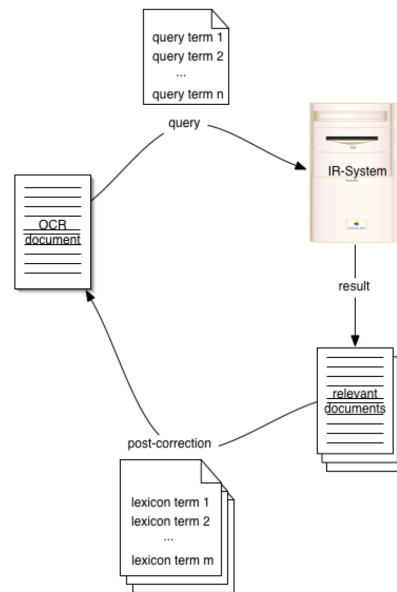


Abbildung 4.4: Nachkorrektur mit Hilfe eines IR-Systems.

nationsmodell für Lexika ließe sich folgendermaßen verfeinern: An Stelle der Booleschen Information „Lexikon an/aus“ wird eine Gewichtung der Lexika einführt. Jeder Einzel-Score wird zusätzlich mit dem Gewichtungsfaktor des Lexikons aus dem er stammt multipliziert. In der graphischen Umsetzung werden die Toggle-Buttons gegen Schieberegler ausgetauscht.

Statische Lexika, die für die Evaluation dieser Arbeit eingesetzt wurden, werden im folgenden Abschnitt vorgestellt. Die Erzeugung dynamischer Crawl-Lexika wird im Anschluss daran vorgestellt.
Fred Damerau zeigt in [10], wie er aus verschlagworteten, elektronischen Dokumenten domänenspezifische Lexika generiert. Da sich die vorgestellte Technik fortwährend auf neue derartige Dokumente anwenden lässt, ist es gerechtfertigt, von dynamischen Lexika zu sprechen. Indem ich mir Information-Retrieval-Systeme zu Nutze mache, umgehe ich die Notwendigkeit einer Verschlagwortung. IR-Systemen liegt die Annahme zu Grunde, dass gleiche Sachverhalte mit gleichem Vokabular ausgedrückt werden. Das wiederum ermöglicht es mir, IR-Systeme zu verwenden, um möglichst passgenaue Lexika für die Nachkorrektur zu gewinnen. Dazu wird aus dem zu korrigierenden Dokument eine Anfrage an ein IR-System generiert. Das IR-System liefert terminologisch ähnliche Dokumente zurück, deren Wörter extrahiert und zu einem Lexikon zusammengefasst werden.

D^E	D^G	D_p	D_g	D_a
315 300	2 235 136	372 628	147 415	1 185

Tabelle 4.1: Größe statischer Lexika.

Das Web bildet zusammen mit Suchmaschinen ein IR-System, das derzeit ca. $3,1 \cdot 10^9$ Dokumente umfasst[1] und durch seinen Aufbau permanent aktualisiert wird. Daher ist dieser Ansatz auch robust gegenüber Neuerungen in der Sprache, wie z. B. neu geformten Akronymen, neu in den Mittelpunkt rückenden Eigennamen oder neuen Schreibvarianten. Im Abschnitt 4.3 stelle ich detailliert vor, wie aus dem Web Lexika gewonnen werden können und ordne die Technik unter diversen Web-Crawl-Technologien ein.

4.2 Statische Lexika

Institutsintern stehen eine Reihe von Speziallexika aus dem CISLex-Projekt zur Verfügung, von denen folgende Lexika für die lexikalische Nachkorrektur eingesetzt wurden:

- D^E, ein Lexikon des Englischen.
- D^G, ein Lexikon des Deutschen (mit den Schreibweisen vor der Rechtschreibreform von 1996).
- D_p, ein Lexikon internationaler Eigennamen.
- D_g, ein Lexikon geographischer Bezeichnungen.
- D_a, ein Abkürzungslexikon.

Aus der ebenfalls am Institut verfügbaren, 2TB großen Web-Spiegelung aus dem Jahre 1999 (WebInTheBox) in Kombination mit D^E bzw. D^G und einem Spracherkenner wurden Lexika mit den 10^5 häufigsten Tokens des Deutschen $D^G_\downarrow$ bzw. des Englischen $D^E_\downarrow$ erzeugt.
Die Korpustexte enthalten z. T. einen nicht vernachlässigbaren Fremdsprachanteil. Zum Beispiel wurden in den deutschen Texten zahlreiche englische Literaturangaben und Fachbegriffe beobachtet oder umgekehrt enthalten die englischen Texte zum Themenkomplex Holocaust eine Reihe deutscher Begriffe. Daher wurde mit WebInTheBox, einem Spracherkenner sowie den Lexika D^E und D^G zusätzlich je ein Lexikon mit den 10^5 häufigsten englischen Tokens innerhalb deutscher Texte $D^{GE}_\downarrow$ und den 10^5 häufigsten deutschen Tokens innerhalb englischer Texte $D^{EG}_\downarrow$ gebildet.

Die auffallende Größe des deutschen Lexikons D^G rührt von der Möglichkeit der Kompositumbildung im Deutschen her.

[1]Sowohl AllTheWeb als auch Google indexieren derzeit etwa diese Menge an Web-Dokumenten.

4.3 Crawl-Lexika

4.3.1 Crawler-Klassifikation

Ein *Crawler* ist ein Programm, das mit einer vorgegebenen Steuerung automatisch die Hypertextstruktur des Webs traversiert und die dabei aufgelesenen Dokumente bearbeitet. Crawling ist keine Technik, die auf einen ersten Ur-Crawler zurückgeht, sondern wurde zu verschiedenen Zwecken an mehreren Stellen unabhängig voneinander entwickelt. Diese Historie spiegelt sich auch an den zahlreichen Namen wider, die in einschlägigen Publikationen synonym zum Begriff Crawler gebraucht werden: *Ant*, *Gatherer*, *Robot* (kurz *Bot*), *Spider*, *Walker*, *Wanderer* und *Worm*.
Einen Eindruck von der Vielfalt an Crawler-Programmen erhält man auf der Seite `http://www.robotstxt.org`, die derzeit knapp 300 verschiedene Crawler auflistet. Mit einer Mischklassifikation möchte ich einen Überblick über diverse Einsatzgebiete von Crawler-Programmen geben und anschließend meinen eigenen Lexikon-Crawler einordnen:

- **Downloading**. Das Herzstück eines jeden Crawlers ist die Möglichkeit, im Web aufgesammelte Dateien lokal zu speichern, da dies die Voraussetzung für jede weitere Verarbeitung ist. Es gibt aber auch eine Reihe von Crawler-Programmen, die in erster Linie für eine Downloading-Möglichkeit konzipiert wurden, wie etwa das Open-Source-Programm wget [21]. Dank der einfachen Möglichkeit wget in UNIX-Shell-Skripte einzubetten, habe ich das Tool für erste Crawling-Experimente verwendet (vgl. dazu auch 9.4).

- **Web-Indexierung**. Crawler, die einen Volltextindex einer Suchmaschine aufbauen bzw. aktualisieren sind zahlenmäßig die größte Crawler-Gruppe und verursachen auch den meisten Crawling-Verkehr im Web. Fälschlicher Weise wird daher manchmal Crawling als gleichbedeutend mit Web-Indexierung gesetzt. Neben den großen Suchmaschinen wie z. B. AllTheWeb und Google, die den Anspruch haben, das ganze Web zu indexieren, gibt es eine Reihe sprachlich, lokal, thematisch oder anders spezialisierter Suchmaschinen. Ein Beispiel dieser spezialisierten Suchmaschinen ist der ResearchIndex CiteSeer [58], mit dessen Hilfe ein Großteil der zitierten Publikationen dieser Arbeit recherchiert wurde.

- **File gathering**. Im Web sind eine Reihe von Crawler-Programmen zu beobachten, die gezielt Dateien sammeln z. B. Texte, Bilder, Programme oder Musik, ohne dass diese Sammlungen in Form von Indexen an die Web-Gemeinde zurückgegeben werden. Motivationen derartige Crawler zu betreiben, reichen vom Aufbau privater Sammlungen bis zu wissenschaftlichen Untersuchungen.

- **Information gathering**. Informationen zu sammeln ist eine Spezialisierung des vorherigen Punktes. Oft sind innerhalb einer Datei nur einzelne

Informationen von Interesse. Beispielsweise werden Web-Seiten von Spam-Versendern nach Email-Adressen durchforstet. Aber nicht nur Missbrauch von Web-Seiten fällt in diese Kategorie, sondern auch alle Ansätze, das Web als riesiges Korpus für computerlinguistische Zwecke Nutzbar zu machen. Meinen eigenen Lexikon-Crawler ordne ich auch in diese Kategorie ein. Ein anderes Beispiel wird in [17] beschrieben. Das Ziel dieser Arbeit ist, aus dem Web Nominalphrasen zu extrahieren, die für eine linguistisch motivierte IR-Indexierung eingesetzt werden.

- **Known-Item Retrieval** ist eine intensive Form des Dateisammelns. Das Ziel ist nicht das Auflesen einer Sorte von Dateien, sondern das Aufspüren digitaler Kopien bestimmter Dateien. Es gibt z. B. Firmen[2], die solche Crawler betreiben, um Copyright-Verletzungen im Web aufzuspüren.

- **Browsing**. Arbeitet ein Browser mit einem Crawler zusammen, kann die Ladezeit von Web-Seiten beschleunigt und die Online-Zeit reduziert werden. Liest eine Surferin eine Web-Seite, wird sie später evtl. einen Link der Seite verfolgen. Die Lesezeit kann genutzt werden, um im Hintergrund die anreferenzierten Seiten rekursiv in einen Cache zu laden. Weiß die Surferin schon zu Beginn, welche Seiten sie besuchen will, können initial alle Seiten lokal geladen werden, so dass während der Lesezeit keine aktive Internetverbindung notwendig ist.

- **Spiegelung** ist eine redundante Rekonstruktion einer Web-Site auf einem anderen Web-Server. Dadurch wird eine bessere Antwortzeit für die Region des Spiegels, eine Lastreduktion auf dem Original-Server und eine höhere Ausfallsicherheit erreicht. Ein guter Mirroring-Crawler entdeckt automatisch Änderungen und lernt, wo er häufiger nach Änderungen Ausschau halten soll. Durch solch eine intelligente Adaption wird verhindert, dass der Crawler selbst zu viel Last produziert.

- **Betreiben einer Web-Site**. Web-Site-Betreiber können mit einem Crawler automatisch die Qualität ihrer Seiten überwachen. Basisanwendung ist die Validierung der HTML-Syntax und das Aufspüren toter Links. Für gute Web-Sites sind noch eine Reihe weiterer Kriterien wie bspw. Rechtschreibung oder die Eignung für Behinderte zu prüfen. Eine andere Anwendungsart ist, mit einem Crawler zu testen wie viele parallele Anfragen ein Web-Server bearbeiten kann.

- **Statistik**. Im Web gibt es zahlreiche Messgrößen, die sich mit Hilfe eines Crawlers statistisch erfassen lassen. Beispiele dafür sind Lebensdauer einer Web-Seite, durchschnittliche Zahl ein- und ausgehender Links, Sprachverteilung, Erreichbarkeit und Ladezeit von Seiten, etc.

- **Monitoring** bezeichnet das automatische, periodische Laden und Bearbeiten einer Web-Seite. Monitoring wird eingesetzt, wenn man den Verlauf

[2] BayTSP, http://www.baytsp.com

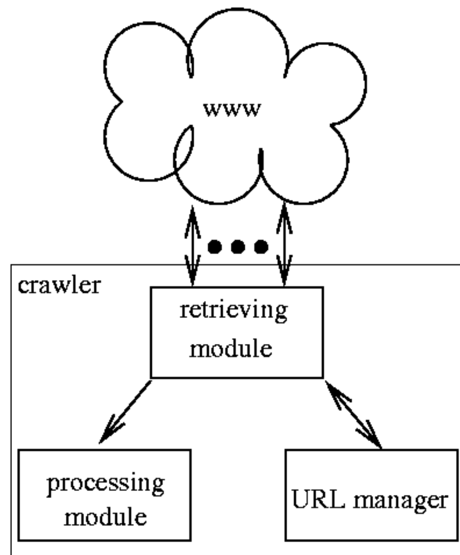


Abbildung 4.5: Crawler-Architektur.

einer Information aufzeichnen möchte, die auf einer Web-Seite zeitlich aufgefrischt publiziert wird, wie z. B. Wetterwerte oder Börsenkurse.

- **Hacking**. Hacking-Crawler lassen sich in zwei Sorten unterteilen. Zum einen Crawler, die systematisch nach Web-Servern Ausschau halten, die bekannte Sicherheitslöcher noch nicht beseitigt haben. Zum anderen Crawler, deren einziger Zweck die Erzeugung von Last durch wiederholte Seitenanfragen auf einem Web-Server ist. Solche Angriffe werden von zahlreichen Parallelinstallationen eines Crawlers zusammen ausgeführt (distributed denial of service attack, DDOS).

4.3.2 Architektur eines Crawlers

In den vielen unterschiedlichen Crawler-Implementierungen findet man mehr oder weniger folgende Architektur wieder (vgl. dazu [5], Kapitel 8):

Retrieving-Modul

Solange der URL-Manager noch weitere URLs liefert, arbeitet das Retrieving-Modul folgende Schrittfolge zyklisch ab:

1. Der URL-Manager wird nach der nächsten zu verfolgenden Adresse gefragt.

2. Das Dokument wird aus dem Web geholt.

3. Alle im Dokument enthaltenen URLs werden an den URL-Manager übergeben.

4. Das Dokument wird an das Processing-Modul übergeben.

Die Zeitverzögerung zwischen Anfrage und Erhalt eines Dokuments ist auf Grund der Unwägbarkeiten im Web nicht exakt a priori bestimmbar. Technische Probleme, fehlerhafte URLs und die Volatilität des Webs können sogar verursachen, dass eine Anfrage unbeantwortet bleibt. Um eine bessere Rechenzeitauslastung zu erzielen, ist es sinnvoll, das Retrieving-Modul in mehreren parallelen Threads zu organisieren. Mit Timern kann verhindert werden, dass das Retrieving-Modul durch ewiges Warten gestoppt wird. Schritt 3 ist nur notwendig, wenn der Crawler rekursiv arbeiten soll, d. h. wenn nicht schon zu Beginn alle zu besuchenden URLs bekannt sind.

URL-Manager

Das öffentliche Interface des URL-Managers enthält primär zwei Methoden: eine, die die nächste zu besuchende URL benennt und eine zum Einfügen neuer URLs. Auch die URLs von denen aus der Crawler startet, genannt Saat (*seed*), werden über diese Standardmethode gesetzt. Das interne Herzstück ist eine Datenstruktur zum Verwalten der URLs zusammen mit Meta-Informationen.
Beim Einfügen der URLs wird die Syntax geprüft und relative URLs werden in absolute überführt. Je nach Anwendung werden evtl. bereits enthaltene, bzw. bereits besuchte URLs ausgefiltert. An Hand des URL-Aufbaus werden evtl. anwendungsabhängig weitere URLs ausgefiltert. Aber nicht nur auf der Seite des Clients (Crawler) werden URLs ausgeschlossen, sondern auch ein Server hat Kennzeichnungsmöglichkeiten für URLs, die nicht besucht werden sollen, wie etwa temporäre oder dynamisch erzeugte Web-Seiten. Der HTML-Standard [82] nennt dazu zwei Optionen: ein File namens `robots.txt` im Wurzelverzeichnis der Web-Site, das angibt, welche URLs auf einer Site vom Crawling auszuschließen sind und ein `meta` Tag mit dem Attribut `robots`, mit dem gekennzeichnet werden kann, dass die aktuelle Seite oder auch Subseiten nicht von einem Crawler besucht werden sollen. Auf Crawler-Seite verwaltet der URL-Manager diese Filterregeln. Die Einhaltung dieser Benimmregeln lässt sich von Servern technisch jedoch nicht erzwingen.
Konzeptionell besteht der URL-Manager aus einer sortierten Liste. Die Einträge sind URLs zusammen mit Meta-Informationen wie z. B. Zeitstempel eines letzten Besuchs der Seite oder die beobachtete Anzahl der Links auf eine Seite. Diese Meta-Informationen werden für die Sortierung der URLs eingesetzt. In großen Anwendungen wie etwa einem Web-Indexierungs-Crawler ist der URL-Manager in einer Datenbank organisiert.

Processing-Modul

Ein Crawler wird durch das Retrieving-Modul und dem URL-Manager am Laufen gehalten. Das Processing-Modul bestimmt, wie die Dokumente weiterver-

arbeitet werden. Im einfachsten Fall, bei einem Downloading-Crawler, werden die Dokumente nur lokal gespeichert. Bei einem Web-Indexierungs-Crawler wird z. B. der invertierte Wort-Index aufgebaut.
Falls Parsing der HTML-Dokumente ohnehin Aufgabe des Processing-Moduls ist, kann die Extraktion weitere URLs, Schritt 3 des Retrieval-Moduls, in das Processing-Modul verlagert werden.

4.3.3 Aufbau meines Lexikon-Crawlers

Retrieving-Modul

Mein Retrieval-Modul habe ich in der Skriptsprache WebL ([41]) verfasst. WebL bietet sowohl Hochsprachkonstrukte für Crawling als auch für Wrapping[3]. In [46] wird für eine Text-Mining-Aufgabe die Verwendung einer vergleichbaren Sprache beschrieben (TPL, Text Parsing Language), die noch besser auf Vokabularextraktion zugeschnitten wäre, allerdings nicht verfügbar ist. Ein WebL-Hochsprachkonstrukt für Crawling ist bspw. der Timer für die Seitenanforderung, der in Form eines try/catch-Blocks zur Verfügung steht. In meiner Implementierung wird auf jede Seite maximal 10 Sekunden gewartet. Im Fehlerfall wird die Seite einfach ignoriert. Auch in Fehlerfällen der anderen beiden Module, bei der Bearbeitung von URLs bzw. Seiten werden diese nicht weiter beachtet. Diese pragmatische Vorgehensweise ist möglich, da der Lexikon-Crawler nicht an gewissen Einzelseiten, sondern an einem Gesamtvokabular interessiert ist. Der Schwund an URLs bzw. Seiten kann bei einer Parametrisierung des Crawlers miteingerechnet werden, d. h. es kann z. B. die zu sammelnde Textmenge bzw. Lexikongröße vorgegeben werden. Auf eine Parallelisierung der Seitenanfragen habe ich verzichtet, da keine zeitkritischen Anforderungen vorliegen; für den Aufbau eines Crawl-Lexikons werden nur in der Größenordnung von 10^3 Seiten geladen. Das WebL-Skript übernimmt die Steuerung des Crawlers und ruft den URL-Manager zur Anfrage der nächsten URL sowie das Processing-Modul in einer einfachen while-Schleife wiederholt auf. Der Crawler arbeitet nicht rekursiv. Dem URL-Manager wird initial eine Liste an URLs übergeben, die er filtert und Schritt für Schritt in der Schleife zurückgibt. Die initiale Liste der zu besuchenden Seiten wird aus einer automatisierten Suchmaschinenanfrage gewonnen (siehe Kapitel 9). In meinen Experimenten habe ich dazu folgende Einstellungen vorgenommen:

- Aus dem zu korrigierenden Text wurden die 25 höchstfrequentesten Wörter exklusive der Funktionswörter bestimmt. An die Suchmaschine wurde eine disjunktive Anfrage dieser 25 Wörter gesendet. Die Auswahl der Anfragewörter über die Frequenz bedient parallel zwei Strategien: Bestimmung einschlägiger Stichwörter der Domäne sowie Ausschluss etwaiger OCR-Lesefehler von der Anfrage.

- Das gewünschte Format der Dokumente wurde auf HTML beschränkt. Das Processing-Modul muss daher nur den Umgang mit einem Format

[3]In 9.4 wird die Verwendung von WebL als Wrapper-Sprache kurz vorgestellt.

Themengebiet	Deutsch	Englisch
Bauern	553 481	132 699
Informatik	376 361	78 222
Fische	557 403	505 678
Holocaust	411 696	227 446
Jagd	622 907	114 730
Kochen	616 982	79 513
Meteorologie	415 657	257 573
Mittelalter	581 178	196 351
Mykologie	328 348	423 860
Neurologie	364 094	283 530
Oper	574 177	156 479
Philosophie	588 742	136 824
Rom	502 117	374 062
Speisepilze	458 787	330 597

Tabelle 4.2: Crawl-Lexika mit Anzahl der Einträge.

beherrschen. Eine Ausdehnung auf Dokumentenformate wie PDF oder Textverarbeitungsformate wie DOC (Microsoft Word) ist sinnvoll, da von Dokumenten dieses Formats zu erwarten ist, dass sie textintensive Inhalte besitzen. Da dies z. T. jedoch prozedurale Formate sind, ist Vorsicht bei der Textextraktion geboten. Bedingt durch Worttrennung am Zeilenumbruch kann man z. B. die beiden Tokens Zuk- und ker erhalten. Um ein Lexikon nicht zu verunreinigen, sind in solchen Fällen mächtige Heuristiken zur Erkennung notwendig. Wenngleich selten, können solche Fälle auch in HTML-Dokumenten auftreten.

- Es wurde die Sprache des zu korrigierenden Texts in der Suchmaschine eingestellt, d. h. in meinem Fall Deutsch bzw. Englisch.
- Da von langen Texten ein breiteres Vokabular zu erwarten ist, wurde die minimale Seitengröße mit 100KB festgelegt.
- Es wurden Seiten, die Audio-Files, Video-Files oder aktive Elemente (Applets, Flash, JavaScript oder VBScript) enthalten von der Suche ausgeschlossen, da deren Fokus offensichtlich nicht auf Text- sondern Multimedia-Inhalten liegt.
- Die Länge der Rückgabeliste wurde auf 1000 URLs festgelegt.

Die gewählten Parameter sind nur eine Ad-hoc-Belegung der vorgefundenen Einstellungsmöglichkeiten. Es ist sinnvoll die einzelnen Parametereinstellungen mit Hilfe weiterer Experimente zu optimieren. In der Evaluation wird jedoch gezeigt, dass damit schon gute Korrekturresultate erzielt werden.

URL-Manager

Kern meines URL-Managers ist eine einfache, in Java implementierte Queue-Datenstruktur (FILO). Die Elemente sind vom Datentyp `java.net.URL`. Der Eingabemethode sind zwei Filter vorangestellt, einer um zu prüfen, ob die Seite geladen werden darf und ein weiterer, um zu prüfen, ob die Seite die Groundtruth enthält.
Für die Anfrage, ob ein Besuch auf einer Web-Seite erwünscht ist, wurde die entsprechende Komponente des Crawlers SPHINX [54] wiederverwendet. SPHINX ist ein in Java geschriebenes, open-source-lizenziertes, parametrisierbares, visualisiertes Crawler-Toolkit. Fast ohne Modifikationen ließ sich die für die Überprüfung zuständige Klasse aus dem Software-Paket herauslösen und in meinen URL-Manager integrieren.
Der Einsatz des zweiten Filters ist durch meine Versuchsanordnung bedingt. Da mein Groundtruth-Korpus aus dem Web stammt, muss verhindert werden, dass die Originalseiten zum Lexikonaufbau herangezogen werden. Einen deutlichen Hinweis auf einen solchen methodischen Fehler erhält man bei einer hundertprozentigen Abdeckungsrate des Lexikons auf einem Teilkorpus. Da einige Seiten mehrfach kopiert im Web vorhanden sind, genügt es nicht, die Originaladressen auszufiltern. Anstatt dessen wird von jedem Dokument ein sog. Fingerabdruck, eine längere Phrase gewählt (in meinen Experimenten wurde dazu die Länge auf 10 Wörter gesetzt) und als Wortgruppe automatisiert an eine Suchmaschine angefragt. Alle URLs dieser Rückgabemenge werden von meinem URL-Manager ausgefiltert. Auf Grund der Dynamik des Webs ist diese Exklusionsliste direkt vor einem Lexikonaufbau zusammenzustellen. Ausserdem muss dieselbe Suchmaschine verwendet werden wie beim Aufbau der Crawl-Liste.
Alle URLs bei denen der Konstruktor der Klasse `java.net.URL` eine Exception wirft, d. h. fehlerhafte und auch relative URLs, werden nicht in meine Queue eingefügt.

Processing-Modul

Die Aufgabe des Processing-Moduls wird in folgende Teilschritte zerlegt:

- Extraktion des Textanteils der HTML-Dokumente,
- Tokenisierung der Texte und
- Aufbau einer Datenstruktur, die alle Tokens zusammen mit ihrer Frequenz innerhalb der Dokumentenmenge verwaltet.

Die Teilaufgabe, aus einer HTML-Seite den enthaltenen Text zu extrahieren erscheint auf den ersten Blick trivial. Bedingt durch fehlerhafte HTML-Syntax, inkonsistente Codierung und die enorme Größe einiger Seiten, scheitern Standard-Werkzeuge wie der HTML-Parser der Java-API oder das UNIX-Tool

html2text ([79]) an einem erstaunlich großen Anteil (fast 25%) der HTML-Seiten. Für meinen Prototypen ist es jedoch ausreichend, nur aus den verarbeitbaren Seiten ein Lexikon aufzubauen. Pragmatisch wurde das Processing-Modul nur auf Skriptebene verzahnt, d.h. der Crawler arbeitet zuerst wie ein reiner Downloading-Crawler. Anschließend wird html2text für jedes geladene HTML-Dokument aus Perl heraus mit einem Timer und einem try/catch-Mechanismus gestartet. Mit dem UNIX-Tool recode ([64]) wird die Codierung auf UTF-16BE vereinheitlicht. Mit einer Java-Implementierung der Methode `getQueryTokens()` aus dem in 3.2.1 vorgestellten Interface `Tokenizer` werden alle Textdateien tokenisiert und zusammen mit einem Frequenzzähler in eine `Hashtable` eingefügt.

4.3.4 Ausbaumöglichkeiten meines Lexikon-Crawlers

Für einen Ausbau meines Lexikon-Crawlers sehe ich zwei Hauptziele:

- Unabhängigkeit von einer Suchmaschine und
- Erzeugung qualitativ hochwertigerer Lexika.

Insbesondere für einen kommerziellen Ausbau ist die derzeitig verwendete Anfragetechnik an Suchmaschinen unzureichend. Ohne eine verlässliche, produktreife Anfragemöglichkeit via Webservice ist eine Abkopplung von Suchmaschinen notwendig. Anstatt vorgegebene Links abzuwandern, ist eine rekursive Crawling-Technik notwendig. In den URL-Manager muss eine Steuerung eingebaut werden, die mit Hilfe des Ankertexts und eines auf der referenzierenden Seite angewandten Lexikongütemaßes beurteilt, ob ein Verweis verfolgt werden soll. Als Saat eignen sich themenspezifisch, manuell gewählte Seiten oder Top-Level-Hubs, d. h. Web-Seiten mit einer Vielzahl von ausgehenden, thematisch unterschiedlichen Verweisen.
Die Evaluation hat gezeigt, dass nur wenige falsche Freunde oder unglückliche Korrekturen auf das Konto falsch geschriebener Wörter im Web gehen. Trotzdem liegt an dieser Stelle ein Potential für eine noch höhere Präzision der Nachkorrektur. Es gibt eine Reihe von Dokumentklassen im Web, die man vom Lexikonaufbau ausschließen möchte, wie z. B. :

- Dokumente, die selbst aus einem OCR-Erkennungslauf stammen.
- Zügig verfasste Dokumente, die viele Tippfehler enthalten, wie z. B. Chat-Protokolle.
- Dokumente, die aus der Feder eines orthographieschwachen Verfassers stammen.
- Dokumente mit einer systematischen Codierungs-Problematik. Die tritt z. B. häufig auf, wenn ein deutsches Dokumente mit einer amerikanischen Tastatur verfasst wird und Umlauttranskriptionen verwendet werden, wie etwa `Loewe` an Stelle von `Löwe`.

- Unspezifische Dokumente, die überproportional oft bei Suchanfragen zurückgeliefert werden. Für eine Vokabularextraktion ist es dabei egal, ob die Seiten von der Suchmaschine gewollt (bezahlte Werbeeinblendungen) oder ungewollt (sog. Link-Spamming) in den Trefferlisten enthalten sind. Das negative Potenzial unspezifischer Dokumente ist geringer als das der zuvor angeführten Dokumentklassen, da nicht systematisch fehlerhafte Tokens eingeschleppt werden, sondern lediglich das Lexikon unnötig aufgebläht wird und die Frequenz- sowie Kontextprofile der Wörter verwässert werden.

Ein Ansatz, solche Seiten auszufiltern ist, für jeden dieser Dokumententypen einen Erkennungsexperten bereitzustellen, der z. B. mit spezifischem Fehlerlexikon oder n-Grammprofil arbeitet.
Die anerkanntesten Seiten zu einem Thema aufzusammeln ist ein alternativer Verbesserungsansatz. Es besteht die Hoffnung, dass die negativen Dokumentenklassen nicht dazu gehören. Dieser Ansatz macht sich die Beobachtung zu Nutze, dass sich in dem nur auf den ersten Blick chaotisch organisierten Web von selbst sog. *Web-Communities* bilden. Eine Web-Community besteht aus sog. *Authorities* und *Hubs*. Authorities sind Seiten mit einflussreichem Inhalt, auf die häufig verwiesen wird. Hubs sind gute Überblicksseiten, die auf Authorities verweisen. An Hand dieser Link-Topologie lassen sich Web-Communities auch automatisch auffinden. In [18] wird der Begriff der Web-Communities näher vorgestellt. Dazu werden Techniken präsentiert, wie man unter Einsatz von Suchmaschinen Web-Communities ausfindig machen kann. Folgende Algorithmusidee ist dabei zentral. Beginnend von einer Saat werden alle Seiten bestimmt, die von diesen Saat-Seiten aus anreferenziert werden, aber auch die Seiten, die auf die Seiten der Saat verweisen. Mit Hilfe der Verlinkung werden Hub- und Authority-Eigenschaften der Seiten gemessen und die schwächsten aus der Menge entfernt. Diese Schritte werden mehrfach wiederholt, bis die Menge eine stabile Zusammensetzung erreicht. Der Zugriff auf die Rückwärtsverlinkung von Web-Seiten wird von Suchmaschinen wie AllTheWeb und Google zur Verfügung gestellt.

4.4 Flexions- und Orthographievarianten

In der Studie [86] wird die wechselseitige Überdeckung der Wörter eines Lexikons (Merriam Webster Seventh Collegiate Dictionary) und eines Zeitungskorpus (New York Times) untersucht. Überraschendes Ergebnis der Studie ist, dass jeweils fast zwei Drittel der Wörter nicht in der anderen Quelle enthalten sind. Von den fehlenden Wort-Tokens aus dem Zeitungskorpus lassen sich rund ein Viertel (16,6% aller Zeitungs-Tokens) dadurch erklären, dass im Lexikon nur die Grundformen der Wörter ohne Flexionsformen enthalten sind. In der Buchkritik [69] werden auf Grund von Vergleichsexperimenten die Zahlen insgesamt zwar angezweifelt, jedoch die Aussagen bzgl. der Flexionsformen bestätigt.
Die in meiner Arbeit eingesetzten, statischen Lexika sind z. T. Vollformenlexika.

Bei der beschriebenen Generierung der Crawl-Lexika ist es hingegen unvermeidlich, dass niederfrequentere Wörter nicht in allen Flexionsformen enthalten sind, z. T. fehlen auch Grundformen. Folgendes beobachtete Beispiel aus dem Korpus zeigt, wie dieses Phänomen ohne weitere Vorkehrungen zu einer unglücklichen Korrektur führt. Die OCR-Engine liest das Wort `würmzeitlichen`, das auch so in der Groundtruth enthalten ist. Im Crawl-Lexikon ist lediglich die Flexionsvariante `würmzeitlicher` enthalten. In den statischen Lexika fehlt dieses Wort zusammen mit all seinen Flexionsvarianten. Die Flexionsvariante ist das einzige Token mit Levenshtein-Abstand eins, so dass es unweigerlich die Korrekturkandidatenliste mit einem hohen Score anführt.
Mit folgender Strategie wird versucht, solche unglücklichen Korrekturen zu verhindern: Falls der Spitzenkandidat einer Korrekturliste mit Konfidenzwert größer eins eine Flexionsvariante des zu korrigierenden Wort-Tokens ist, wird die Korrektur verworfen.
Der Flexionsvariantentest hat sprachabhängig zu erfolgen und wurde über ein Java-Interface formuliert.

```
public interface Variants {

/* ISO 639 language codes */
   // [...] //
final static int DE =  23;  //German
final static int EN =  26;  //English, American
    // [...] //

boolean isFlexion(String s1, String s2, int language);
boolean isOrthographicVariant(String s1, String s2, int language);

}
```

Von den an den ISO 639 language codes angelehnten Konstanten sind hier nur die beiden angeführt (DE und EN), für die auch eine prototypische Referenzimplementierung des Flexions-Varianten-Tests vorhanden ist. Der Test ist eher vollständig als korrekt und in erster Linie einfach gehalten, erzielt jedoch gute Resultate. Es wird eine Liste deutscher Suffixe gebildet:

```
["", "e", "an", en", "in", "on", "ung", "rn", "ln", "es", "r", "s"].
```

Ebenso wird eine Liste englischer Suffixe gebildet:

```
 ["", "s", "ed", "ing", "ly", "less", "er"].
```

Den beiden zu vergleichenden Strings werden von der Kopfseite solange Zeichen abgetrennt, solange diese gleich sind. Sind die übrig bleibenden Rümpfe beide in der deutschen, bzw. englischen Suffixliste enthalten, wird auf eine Flexionsvariation geschlossen und der Korrekturvorschlag verworfen.

Analog zu den Flexionsvarianten können auch orthographische Variationen zwischen Lexikoneintrag und gelesenem Text bei fehlender Vorkehr zu unglücklichen Korrekturen führen.
Im Deutschen sind nach der Rechtschreibreform[4] von 1996 für einige Wörter Varianten erlaubt.
Am auffälligsten davon sind die reformierten Laut-Buchstaben-Zuordnungen, wie z. B. die Ersetzung des `ß` durch `ss` nach kurzem Vokal, so dass nach parallel geltender, alter Rechtschreibung in diesem Satz auch `daß` stehen könnte. Bei Bedarf kann die relativ kleine Menge betroffener Wörter in Form eines statischen Lexikons hinzugeschaltet werden. Beim Aufbau von Crawl-Lexika werden in der Regel automatisch beide Varianten ins Lexikon aufgenommen.
Der neue Vorzug von getrennten Schreibungen gegenüber Zusammenschreibungen wie z. B. `Rad fahren` an Stelle von `radfahren` ist für die Nachkorrektur unproblematisch, da es sich ausschließlich um Zerlegungen in lexikalische Wörter handelt. Auch Änderungen der Groß- und Kleinschreibung haben keine Auswirkung auf meine Referenzimplementierung, da derzeit alle Buchstaben (noch) auf Kleinschreibung vereinheitlicht werden.
Ein in der Öffentlichkeit weniger beachteter Reformschritt ist die starke Liberalisierung der Verwendung des Bindestrichs für die Kompositumbildung. Für eine bessere Übersichtlichkeit kann z. B. im vorherigen Satz auch das Wort `Kompositum-Bildung` stehen. Daraus resultiert ein Variantenreichtum v. a. für seltene Wörter. In [49] werden Gebrauch und Variantenreichtum der Bindestrichwörter quantifiziert. Die Zahlen zeigen deutlich, dass es sich keineswegs um ein vernachlässigbares Randphänomen handelt. Das vorgestellte Projekt Wortwarte sammelt aus Online-Zeitungen systematisch neu auftretende Wörter. Davon enthalten 40.2% einen Bindestrich; davon enthalten wiederum 14.8% mehr als einen Bindestrich wie z. B. `Set-Top-Box-Software-Konzept`. Durch Weglassen und Setzen der Bindestriche an anderer Stelle sowie Variation der Groß-/Kleinschreibung wurden Variantengruppen mit bis zu 9 Varianten pro Wort beobachtet.
Die Methode `isOrthographicVariant()` wurde exemplarisch für das Deutsche implementiert. Aus beiden Eingabe-Strings wird eine kanonische Form ohne Bindestriche gebildet. Stimmen die beiden daraufhin überein, wird auf eine orthographische Variante geschlossen und der Korrekturvorschlag verworfen.

4.5 Perfektes Lexikon

Aus dem Verlagswesen ist bekannt, dass trotz wiederholtem Korrekturlesen Menschen Fehler übersehen, denn auch zu Büchern, die schon in mehrfacher Auflage gedruckt wurden, werden noch Errata-Listen veröffentlicht. Eine Präzision von 100% ist daher auch in der OCR-Nachkorrektur i. Allg. nicht erreichbar. Man muss aber nicht die menschliche Fehlbarkeit bemühen, um zu zeigen, dass eine Genauigkeit von 100% unerreichbar ist, denn der automatischen Nachkorrektur selbst ist eine obere Schranke immanent. Mit künstlich erzeugten, per-

[4] Eine Zusammenfassung der Reform findet man in [25].

fekten Nachkorrektur-Ressourcen lässt sich diese obere Schranke auf Trainingsmaterial untersuchen. In erster Linie handelt es sich bei diesen Ressourcen um ein perfektes Lexikon, das exakt die Wörter des Dokuments enthält. Ausserdem verwendet man perfekte Scores, die genau das Verhalten der Rohmaße innerhalb des Dokuments widerspiegeln. Perfekte Scores zu allen String-Abstandsmaßen lassen sich dabei alleine aus dem perfekten Lexikon bestimmen; andere, wie z. B. kontextuelle Maße lassen sich nur aus dem Originaldokument ableiten.
Trotz Verwendung eines perfekten Lexikons zur Nachkorrektur, kann diese zu Fehlern führen, im aktiven wie im passiven Fall. Dazu ein Beispieldokument, das nur aus einem Satz besteht: `man las mein buch`. Das perfekte Lexikon enthält genau diese vier Wörter.
Angenommen eine OCR-Engine liest an Stelle dessen `man las man buch`. Eine am Levenshtein-Abstand orientierte Nachkorrektur bleibt passiv und übergeht diesen *false friend*. Falls eine OCR-Engine an Stelle des Originals `mcin las mein buch` liest, wird eine am Levenshtein-Abstand orientierte Nachkorrektur aktiv. Allerdings wird die Kandidatenliste vom falschen Vorschlag angeführt - das ist die Fehlerklasse *wrong candidate* - und zu `mein las mein buch` korrigiert, da $lev(\texttt{mcin}, \texttt{mein}) = 1 < 2 = lev(\texttt{mcin}, \texttt{man})$. Führt man noch eine Korrekturgrenze ein, können bedingt durch Fehler der beiden vorherigen Klassen zusätzlich auch noch Fehler der Klassen *too cautious* und *wrong candidate and threshold* auftreten. Die Fehlerklassen des rechten Astes meines Klassifikationsschemas sind hingegen ausgeschlossen, da die Bedingung $w^{orig} \in D$ für das perfekte Lexikon per Definition immer gilt.
Mit perfekten Frequenz-Scores lassen sich die beiden Fehler im Beispiel nicht vermeiden; hingegen mit perfekten Kontext-Scores, z. B. Scores bzgl. der Häufigkeiten von Wort-n-Grammen werden die Fehler korrigierbar.
Für die Evaluation wurde zu jedem thematischen Teilkorpus in jeder Sprache ein perfektes Lexikon mit perfekten Frequenzinformationen aus den Groundtruth-Dokumenten erzeugt.

Kapitel 5

Rohmaße und Scores

5.1 Motivation

In meiner Nachkorrektur-Software spricht die Komponente zur Parameteroptimierung für einen Teil der eingehenden Korrekturvorschläge eine Empfehlung aus. Für diesen Schritt der Ordnung und des Aussortierens werden in diesem Kapitel die notwendigen Nachkorrekturhilfen vorgestellt, d. h. die Rohmaße, deren Repräsentation als Scores sowie die eingesetzte Kombinationstechnik für Scores. Zu jedem OCR-Token wird in der Kandidatenerzeugungskomponente eine Liste potentieller Korrekturvorschläge generiert, die im nächsten Arbeitsschritt nach Güte geordnet werden sollen. Liest beispielsweise eine OCR-Engine `kave` und die Kandidatenerzeugung generiert dazu aus englischen Lexika die Vorschlagsliste [`ate`, `have`, `nave`], würde ein Mensch wohl die Vorschläge intuitiv folgendermaßen ordnen: `ate` an letzter Stelle, da es mit dem OCR-Token weniger Symbole gemeinsam hat als die anderen beiden; `nave` an zweiter Stelle, da es ein eher seltenes Wort ist und `have` als Spitzenkandidat. Die Intuition lässt sich auch mit einem Abstandsmaß[1] und einer Frequenzmessung[2] quantifizieren, sog. Rohmaße:

$$lev(\texttt{kave}, \texttt{ate}) = 2 > 1 = lev(\texttt{kave}, \texttt{have}) = lev(\texttt{kave}, \texttt{nave})$$

und die Frequenzwerte ordnen die beiden Kandidaten mit gleichem Levenshtein-Abstand:

$$freq(\texttt{nave}) = 387 < 113\ 747 = freq(\texttt{have})$$

Als Rohmaße bezeichne ich alle numerischen Zusatzinformationen, die helfen eine sinnvolle Ordnung auf den Korrekturkandidaten herzustellen.
Dieser erste skizzierte Kombinationsansatz ist noch ausbaufähig, da es eine Reihe weiterer Rohmaße gibt, die zur Desambiguierung von Kandidatenlisten eingesetzt werden können wie z. B. kontextuelle Informationen oder alternative Abstandsmaße. Um mehrere Rohmaße bequem miteinander kombinieren zu

[1] im Beispiel wird der klassische Levenshtein-Abstand verwendet; für eine Def. siehe 8.3.1
[2] die Frequenzwerte stammen aus dem statischen Korpus WebInTheBox

können, wird jedes Rohmaß in einen Score transformiert.
Ein Score ist immer Element des Intervalls [0; 1]. Es gilt, je besser ein Kandidat bzgl. eines Rohmaßes eingeschätzt wird, desto höher ist der zugehörige Score. Durch die Vereinheitlichung der Rohmaße zu Scores ist kein zusätzliches Wissen über den Wertebereich der Rohmaße für eine Weiterverarbeitung erforderlich. So gilt z. B. beim Rohmaß Levenshtein-Abstand: je kleiner desto besser, wobei 0 der beste Wert ist. Hingegen beim Rohmaß Frequenzwert gilt: je größer desto besser, wobei der maximale Wert von der Größe des Korpus abhängt. Mit Scores wird eine einfache Schnittstelle für verteilte Entwicklung von Wissensquellen geschaffen, indem alle Wertebereichsbetrachtungen von Rohmaßen an die Score-Lieferanten ausgelagert werden.
Es folgt eine Vorstellung diverser Rohmaße mit zugehörigen Scores.

5.2 Levenshtein-Abstand

Im Kapitel 8.3 werden diverse Varianten des Levenshtein-Abstands vorgestellt. Davon eignen sich alle Varianten auf Symbolebene zur Selektion von Korrekturkandidaten. Allen gemeinsam ist, dass kleine Werte als gut gelten, der Wert für den günstigsten Fall bei 0 liegt und die auftretenden Werte nicht extrem verteilt sind im Sinne einer maschinellen Handhabbarkeit. Eine sichere, obere Schranke kann durch Systemparameter bestimmt werden. Daher lässt sich der Score einfach mit einer Division durch diese Schranke berechnen:

$$score_{lev}(v^{cand}, w^{ocr}) = \frac{lev(v^{cand}, w^{ocr})}{lev_{system}^{max}}$$

Der Levenshtein-Abstand selbst wird mit zwei geschachtelten Schleifen berechnet, indem die Übergangsmatrix aufgebaut wird.

5.2.1 Klassischer Levenshtein-Abstand

Der maximale, klassische Levenshtein-Abstand zwischen zwei Strings der Länge m und n ist $max(m, n)$. Die maximale Länge normaler Tokens ist auf 64 festgelegt worden. Durch die Grobfiltereinstellung bei der Kandidatenauswahl ist der maximale Levenshtein-Abstand lev_{filter}^{max} noch deutlich niedriger adjustiert. In meiner Referenzimplementierung gilt $lev_{filter}^{max} = 2$. Für einen Ausbau der Kandidatenerzeugung stehen derzeit schon Levenshtein-Automaten bis $lev_{filter}^{max} = 3$ zur Verfügung. Es gilt also für den zugehörigen Score:

$$score_{lev_{classic}}(v^{cand}, w^{ocr}) = \frac{lev_{classic}(v^{cand}, w^{ocr})}{lev_{filter}^{max}}$$

Der klassische Levenshtein-Abstand und damit auch der $score_{lev}$ nehmen nur sehr wenige unterschiedliche Werte an. Da mit den Scores einerseits Korrekturkandidaten innerhalb einer Liste sortiert werden sollen und andererseits auch Konfidenzwerte für Korrekturvorschläge berechnet werden sollen, ist der klassische Levenshtein-Abstand zu grobkörnig; die nächsten beiden vorgestellten Varianten beheben dieses Problem.

5.2.2 Längensensitiver Levenshtein-Abstand

Der längensensitive Levenshtein-Abstand ist zwischen einem leeren String und jedem nichtleeren String immer konstant $lev_{length}(\epsilon, x) = 2$. Dies ist gleichzeitig der maximale Wert, den der längensensitive Levenshtein-Abstand annehmen kann. Das Maximum lässt sich durch meine Systemparameter noch weiter eingrenzen. Sei beispielsweise die Grobfilterung mit $lev_{filter}^{max} = 3$ festgelegt. Ausserdem werden nur normale Tokens für die OCR-Nachkorrektur angefragt und auch nur normale Tokens sind in einer Kandidatenliste möglich, so dass die minimale Wortlänge auf 2 beschränkt ist. Daher liegt der maximale längensensitive Levenshtein-Abstand zwischen zwei Strings a und b mit $|a| = 2$ und $|b| = 3$, die kein Symbol gemeinsam haben bei $lev_{length}(a, b) = 1,2$. Trotzdem verwende ich als obere Schranke den Wert 2, da die Abhängigkeiten zwischen der Definition der Länge normaler Tokens, dem maximalen grobfilterbedingten Abstand und dem maximalen längensensitiven Levenshtein-Abstand unverhältnismäßig aufwändig zu codieren sind. Daher gilt für den zugehörigen Score:

$$score_{lev_{length}}(v^{cand}, w^{ocr}) = \frac{lev_{length}(v^{cand}, w^{ocr})}{2} = \frac{lev_{classic}(v^{cand}, w^{ocr})}{|v^{cand}| + |w^{ocr}|}$$

5.2.3 Gewichteter Levenshtein-Abstand

Der maximale gewichtete Levenshtein-Abstand entspricht dem Produkt der maximalen String-Länge und dem maximalen Editiergewicht, also in der Referenzimplementierung $64 \cdot weight^{max}$. Durch die Grobfiltereinstellung bei der Kandidatenauswahl wird der maximal zu betrachtende, gewichtete Levenshtein-Abstand noch deutlich weiter auf $lev_{filter}^{max} \cdot weight^{max}$ beschränkt. In [67] wird die Berechnung der Editiergewichte beschrieben, die mit meinen Werkzeugen zu Score vereinheitlicht und anschließend angewandt werden. Es wurden verschiedene Verfahren zur Bestimmung der Gewichte eingesetzt, die auch verschiedene Maximalwerte aufweisen. Allgemein gilt:

$$score_{lev_{weight}}(v^{cand}, w^{ocr}) = \frac{lev_{weight}(v^{cand}, w^{ocr})}{lev_{filter}^{max} \cdot weight^{max}}$$

5.2.4 Gewichteter längensensitiver Levenshtein-Abstand

Der Score zur Relativierung des gewichteten Levenshtein-Abstands gegenüber der Wortlängen lässt sich unter Zuhilfenahme von $score_{lev_{weight}}$ definieren:

$$score_{lev_{weightLength}}(v^{cand}, w^{ocr}) := \frac{score_{lev_{weight}}(v^{cand}, w^{ocr})}{|v^{cand}| + |w^{ocr}|}$$

5.3 Frequenz

Diesem Rohmaß liegt die Idee zu Grunde, dass man bei der Kandidatenauswahl besser abschneidet, wenn man häufig auftretende Wörter gegenüber selten auftretenden bevorzugt.

Es wird zwischen statischen und dynamischen Lexika unterschieden. Diese Unterscheidung gilt nicht nur für die enthaltenen Wörter, sondern auch für die zugehörige Frequenzinformation.
Die Frequenzinformation zu den statischen Lexika wurde aus der Korpus-Datenbank WebInABox gewonnen. Diese DB steht institutsintern zur Verfügung. Der Korpus entstammt einem ca. 2TB großen Crawl des Webs aus dem Jahr 2000. Die Korpus-DB umfasst auch Frequenzlisten des Deutschen und des Englischen. Dazu wurden die deutschen bzw. englischen Web-Seiten des Crawls mit einem Spracherkenner ausgewählt und Tokenisiert. Tokens, die mehr als dreimal vorhanden waren, wurden in die Frequenzlisten mitaufgenommen. Alle statischen Lexika, die in dieser Arbeit zum Einsatz kommen, wurden mit den WebInABox-Frequenzinformationen angereichert. Wörter, die nicht in den WebInABox-Frequenzlisten enthalten sind, erhielten per Default den Frequenzwert 1. Die Kombination mit den WebInABox-Frequenzinformationen schließt den Wert 2 für meine statischen Lexika also aus.
Die Frequenzinformation der dynamischen Lexika wird direkt aus der Tokenisierung der eingesammelten Web-Seiten ermittelt.
Die Frequenzinformation wird immer bzgl. *eines* Lexikons bestimmt. Werden in der Nachkorrektur statische und dynamische Lexika kombiniert, hat der gleiche Korrekturkandidat i. Allg. zwei unterschiedliche Frequenzwerte. Davon wird der bessere ausgewählt. Die Wahl basiert auf den zugehörigen Scores $score_{freq}$. Ein Korrekturkandidat, der nicht im Lexikon enthalten ist, erhält den Frequenzwert 0, ansonsten den Frequenzwert des Lexikoneintrags. Ein Frequenzwert liegt also immer zwischen 0 und einem maximal beobachteten Wert $freq^{max}$. Da eine hohe Frequenz ein Indikator für einen guten Korrekturvorschlag ist, liegt die Polarität umgekehrt wie beim Levenshtein-Abstand. Naheliegende Ansätze zur Normierung auf das Intervall [0; 1], der Kehrbruch oder die Relativierung zum Maximalwert abgezogen von 1, scheitern an der extremen Verteilung der Worthäufigkeiten. Wie in einzelnen natürlichsprachlichen Dokumente, sind auch in meinen Lexika die Worthäufigkeiten nach dem Zipf'schen Gesetz [97] verteilt, da sie natürlichsprachlichen Dokumentensammlungen entstammen. Dadurch ergeben sich so hohe Frequenzwerte einzelner Wörter, dass Berechnungen mit Standard-Datentypen diverser Programmiersprachen in numerisch instabile Bereiche gelangen. Englische Wörter wie *and* und *the* haben Frequenzwerte $> 10^9$. Zur Verdeutlichung zeige ich die Frequenzverteilung der Tokens im Groundtruth-Korpus (6 Themengebiete). Die beiden Plots zeigen mit ihrer logarithmischen x-Achse auch, dass der extreme Ausschlag der Zipf'schen Kurve mit einem Logarithmus gedämpft und nahezu linearisiert werden kann, im Deutschen besser als im Englischen.

Anders als bei den Levenshtein-Abstandsmaßen hängt der Frequenz-Score nur von v^{cand} und nicht von w^{ocr} ab. Um beim Logarithmieren keine Probleme mit Null-Frequenzwerten zu erlangen, werden alle Frequenzen künstlich inkrementiert. Daher gilt für den zugehörigen Score:

$$score_{freq}(v^{cand}) = 1 - \frac{\ln(freq(v^{cand}) + 1)}{\ln(freq^{max} + 1)}$$

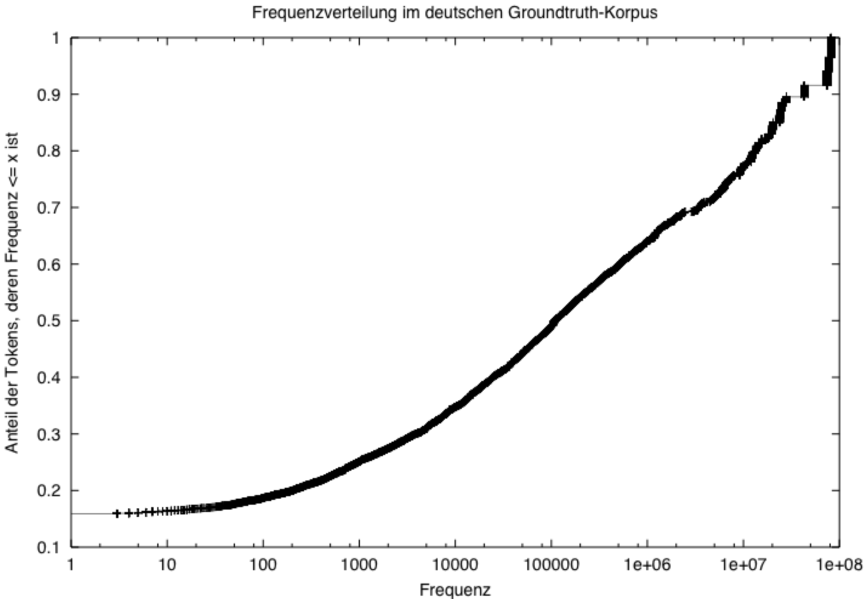


Abbildung 5.1: Frequenzverteilung im deutschen Groundtruth-Korpus.

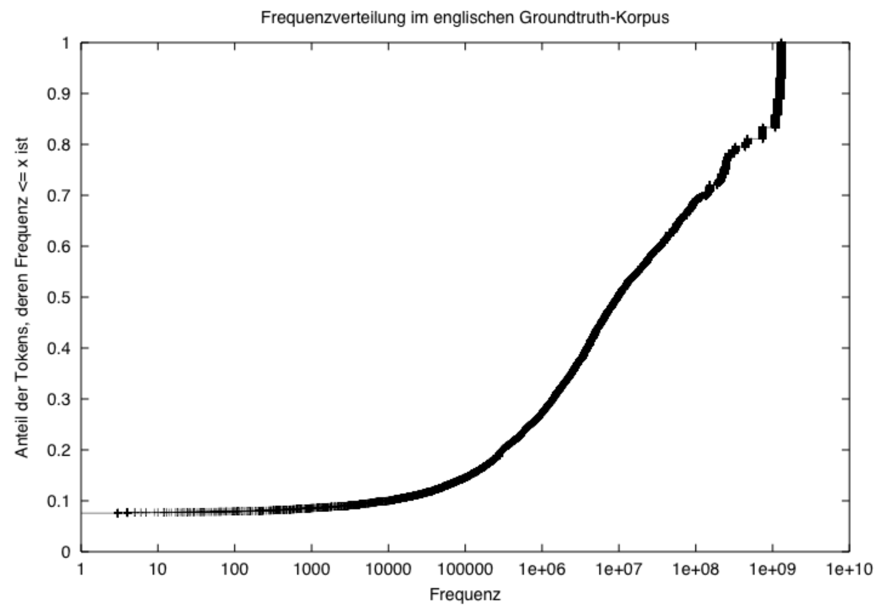


Abbildung 5.2: Frequenzverteilung im englischen Groundtruth-Korpus.

5.4 Kollokation und Kookkurenz

5.4.1 Rohmaße und Scores für Korpora

Diesen Rohmaßen liegt die Idee zu Grunde, dass man bei der Kandidatenauswahl statistisch misst wie gut sich ein Kandidat in seine Umgebung einpasst. Diese Maße können als eine Erweiterung des Begriffs Frequenz auf die Umgebung eines Wortes aufgefasst werden.
Mit Kookkurenz wird die Wahrscheinlichkeit bezeichnet, dass ein Wort w_1 zusammen mit einem anderen Wort w_2 innerhalb eines sog. Fensters von Wörtern auftritt $P(w_1\&w_2)$. Als Kollokation bezeichnet man dagegen eine Kookkurenz, die das zufällige Zusammentreffen zweier Wörter signifikant übersteigt. Dazu muss die Nullhypothese H_0 der zufälligen Kookkurenz, die sich aus dem Produkt der beiden einzelnen Auftrittswahrscheinlichkeiten berechnet, zu einem vorgegebenen Signifikanzniveau abgelehnt werden. $H_0 : P(w_1\&w_2) = P(w_1)P(w_2)$
In der Literatur (vgl. z. B. [52]) werden dazu eine Reihe statistischer Tests vorgestellt. In [14] wird gezeigt, dass die Klasse der *likelihood ratio tests* sich besser eignet als die weitverbreiteten, klassischen Tests wie z. B. der t-Test, da keine Normalitätsannahmen einfließen und auch für kleine Datenmengen gute Ergebnisse erzielt werden können. Diese Tests sind zu bevorzugen, da selbst bei Einsatz des derzeit größten, verfügbaren Korpus – dem Web – die Datenmenge zur Messung von Kollokationen und Konkurrenzen relativ dünn ist (s. u.). Eine einfache Variante ist die *pointwise mutual information* (vgl. [52], S.178f oder [31], S.112f):

$$MI(w_1, w_2) = \log_2 \frac{P(w_1\&w_2)}{P(w_1)P(w_2)}$$

Als Schätzer für die Wahrscheinlichkeiten werden Frequenzmessungen eingesetzt:

$$MI(w_1, w_2) = \log_2 \frac{\frac{freq(w_1\&w_2)}{|C|}}{\frac{freq(w_1)}{|C|}\,\frac{freq(w_2)}{|C|}} = \log_2 |C| + \log_2 freq(w_1\&w_2) - \log_2 freq(w_1) - \log_2 freq(w_2)$$

Wobei $|C|$ die Größe des Korpus gemessen in der Anzahl von Wörtern angibt.
Ein Maß für Kookkurenz erhält man, indem man die logarithmierten Einzelfrequenzen weglässt; dadurch kann auch der konstante Summand weggelassen werden, ohne dass das Maß negativ werden kann: $\log_2 freq(w_1\&w_2)$
Nach Untersuchungen von [31] eignen sich auch Kookkurenzen zur Kandidatenauswahl für die OCR-Nachkorrektur.
Um aus den beiden Rohmaßen für Kollokation und Kookkurenz Scores zu gewinnen, wird analog zu 5.3 ein maximal beobachteter Frequenzwert zur Relativierung verwendet. Da $|C|$ im Web nur schwer zu bestimmen ist, wird es ebenfalls mit $freq^{max}$ approximiert. Um beim Logarithmieren keine Probleme mit Nullwerten zu erlangen, werden alle Frequenzen künstlich um eins erhöht. Die gewünschte Polarität der Scores wird durch die Subtraktion von 1 erreicht:

$$score_{colloc}(v^{cand}, w^{ocr}) := \frac{1 + \log_2(freq(w_1) + 1) + \log_2(freq(w_2) + 1) - \log_2(freq(w_1\&w_2) + 1)}{2 \log_2(freq^{max} + 1)}$$

und

$$score_{coocc}(v^{cand}, w^{ocr}) := 1 - \frac{\log_2(freq(w_1\&w_2) + 1)}{log_2(freq^{max} + 1)}$$

In den beiden Formeln steht w_1 für den Korrekturkandidaten selbst und w_2 für das zu betrachtende Nachbarwort der OCR-Erkennung, also
$w_1 = v^{cand}$ und $w_2 = neighbour(w^{ocr})$
In der Regel ist mit Nachbarwort das direkt vorangehende oder folgende Wort gemeint, kann aber z. B. auch das nächste Wort sein, das kein Stoppwort ist. Die Score-Definitionen sind nicht nur für das Web, sondern allgemein für Kollokationen und Kookkurenzen in Korpora einsetzbar. Als Parameter bleibt noch die Fenstergröße für die betrachteten Kookkurenzen festzulegen, d. h. die Anzahl der vorhergehenden bzw. nachfolgenden Wörter, die für die Entscheidung herangezogen werden, ob zwei Wörter zusammen auftreten.

5.4.2 Machbarkeitsstudie für das Web als Korpus

Bei Einsatz des Web als Korpus sowie bei Einsatz von Suchmaschinen für die Lokalisierung von Kookkurenzen innerhalb des Web, orientiert sich die Fenstergrößenwahl an der Ausdrucksstärke der Suchmaschinenanfragen. Die Verwendete Technik einer automatisierten Anfrage an Suchmaschinen ist in beschrieben. Die beiden eingesetzten Suchmaschinen – Google und AllTheWeb – erlauben eine Fenstergröße 1 mit Hilfe genauer Wortgruppenanfrage sowie eine weitläufige Kookkurenz, ein gemeinsames Auftreten innerhalb eines Dokuments durch die implizite UND-Verknüpfung bei Anfragen zweier Begriffe. Will man bspw. die Kookkurenzen der Begriffe Kohl und Birne mit Fenstergröße 1 zählen, sind folgende zwei Wortgruppenanfragen notwendig: `"Birne Kohl"` und `"Kohl Birne"`. Die Dokument-Kookkurenz lässt sich mit der Anfrage `Kohl Birne` (ohne Anführungszeichen) ermitteln. Ausserdem ermöglicht Google mit dem $*$ als Auslassungszeichen innerhalb einer Wortgruppenanfrage den Zusammenbau beliebiger Fenstergrößen durch mehrere Anfragen. Will man bspw. die Kookkurenzen der beiden Begriffe auf die Fenstergröße 2 ausweiten, sind folgende vier Wortgruppenanfragen notwendig: `"Birne Kohl"`, `"Birne * Kohl"`, `"Kohl Birne"` und `"Kohl * Birne"`. Einige Suchmaschinen wie z. B. AltaVista bieten ausserdem in ihrer Syntax noch den Operator `near` an, der Kookkurenzen innerhalb einer vorgegebenen Fenstergröße (bei AltaVista: 10) sucht. Die Syntax der Suchmaschinen erlaubt noch eine erweiterte Variante der vorgestellten Kookkurenzen, die Kookkurenz eines Korrekturkandidaten als Zentralwort zusammen mit einem Prä- und/oder Postkontext[3] unter vorgegebener Länge. Dazu werden Wort-n-Gramme als genaue Wortgruppen angefragt. Als Frequenzinformation lässt sich jeweils direkt mit der Suchmaschine die Anzahl der Dokumente ermitteln, die eine Kookkurenz enthalten, nicht jedoch die Anzahl der Kookkurenzen innerhalb eines Dokuments; dazu müsste man jedes Dokument einer Treffermenge auf die lokale Maschine laden und parsen.
Auf Grund der Einschränkung auf wenige automatisierte Suchmaschinenzugriffe

[3]die Begriffe werden in 8.5 in anderem Zusammenhang eingeführt

pro Tag (siehe 5.4.2) konnten die Scores $score_{colloc}$ und $score_{coocc}$ nicht im gleichen Maße wie die Levenshtein-Abstände und Frequenzen evaluiert werden. Es wurde lediglich in einer kleinen Studie geprüft, für welche Anfragetypen überhaupt eine hinreichend große Frequenz gemessen werden kann. In der Literatur wird der Einsatz von Kookkurenzen als kritisch beschrieben, da alle Korpora bislang eine zu geringe Datenmenge enthalten. In meiner Studie habe ich untersucht, inwieweit diese Einschätzungen auch für das heutige Web gelten.
Aus dem OCR-Korpus habe ich einen 603 Wörter langen Text aus dem Themenkomplex Neurologie ausgewählt. Dieser Text wurde gewählt, da er zum Zeitpunkt der Kookkurenz-Experimente nicht mehr via Web verfügbar war, obwohl er wie alle meine Groundtruth-Dokumente ursprünglich aus dem Web stammt. Die Volatilität des Webs begünstigt so meinen Versuchsaufbau, da die gemessenen Frequenzwerte nicht weiter bereinigt werden müssen. Der Text ist in Deutsch abgefasst, enthält aber einen erheblichen Anteil englischen Vokabulars, auf Grund enthaltener Fachtermini und Literaturangaben. Sofern vorhanden wurden jedem Wort zwei Korrekturkandidaten mittels Grobfilter aus einem deutschen Standardlexikon zugeordnet. Insgesamt enthält der Text 920 Korrekturkandidaten. Die Länge des gewählten Texts wurde durch die Obergrenze von 1000 erlaubten Zugriffen pro Tag bei [23] bestimmt. In einer Experimentreihe zu Kookkurenzen wurde jeweils die Verteilungsfunktion aller Anfrageresultate berechnet. Zur Dämpfung der Kurve wurden die Frequenzen logarithmiert; ausserdem wurden die Frequenzen um eins inkrementiert, um Nullwerte zu umgehen.
Im ersten Experiment wurde die Fenstergröße auf eins gesetzt und die Betrachtung auf den Postkontext beschränkt, d. h. es wurde für jeden Korrekturkandidaten das Wort-Bigramm mit dem nachfolgenden OCR-Token angefragt.

Im zweiten Experiment wurde für jeden Korrekturkandidaten ein Wort-Trigramm angefragt, bestehend aus einem Wort als Präkontext, einem Wort als Postkontext und dem Kandidaten als Zentralwort.

Im dritten Experiment wurde für jeden Korrekturkandidaten ein Wort-4-Gramm angefragt, bestehend aus einem Wort als Präkontext, zwei Wörtern als Postkontext und dem Kandidaten als Zentralwort.

Im vierten Experiment wurde die Dokument-Kookkurenz jedes Kandidaten mit einem charakteristischen Wort der Domäne gemessen. Dazu wurde das höchstfrequenteste Wort des Textes gewählt, das kein Stoppwort ist: `induction` mit 12 Vorkommen. Das Experiment wurde zur Sicherheit mit dem zweithäufigsten Wort `symposium` wiederholt, ohne dass nennbare Differenzen in der Verteilung aufgetreten sind.

Fazit: Alle Verteilungsfunktionen zeigen, dass durch Logarithmieren die Kurven nahezu linearisiert werden können. Der y-Wert an der Stelle 0 gibt die Prozentzahl aller Anfragen wieder, die eine leere Trefferliste zurücklieferten. Der Anteil der richtigen Kandidaten im Kontext, die eine leere Web-Trefferliste liefern, fällt moderater, aber parallel im Trend aus:

Der Text enthält insgesamt 8 OCR-Fehler. Für 2 davon besteht keine Chance sie zu korrigieren, da die korrekten Tokens nicht in der Kandidatenliste enthalten sind. Verwendet man die Frequenzen zur Desambiguierung der restlichen 6

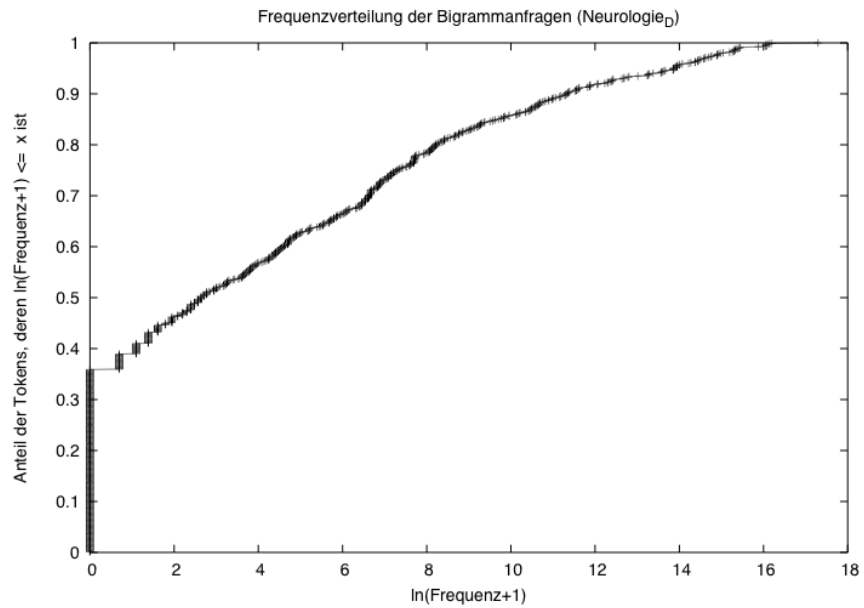


Abbildung 5.3: Beispielhafte Frequenzverteilung von Bigrammanfragen.

Bigrammanfrage	12.8%
Trigramm-Anfrage	42.4%
4-Gramm-Anfrage	66.7%
Dokument-Kookkurenz	1.0%

Tabelle 5.1: Erfolglose Anfragen korrekter Kandidaten im Kontext.

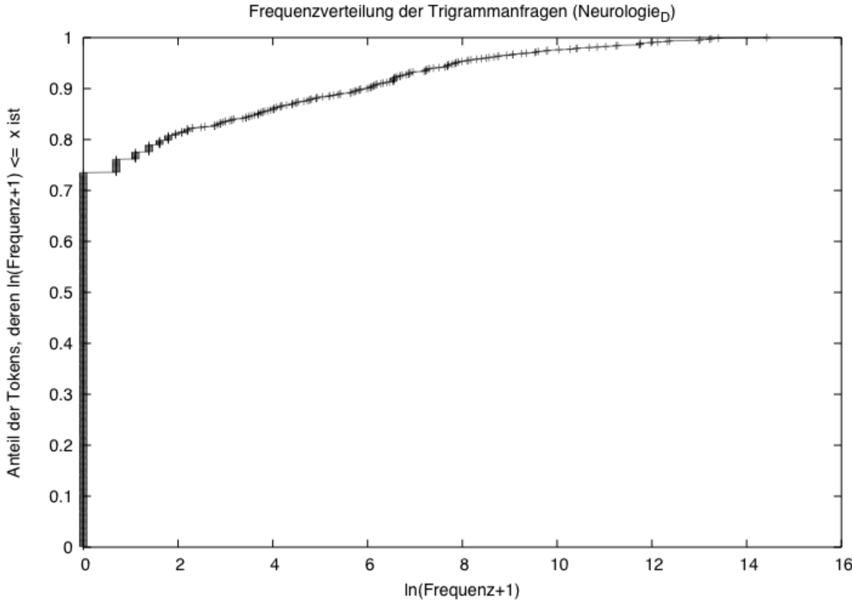


Abbildung 5.4: Beispielhafte Frequenzverteilung von Trigrammanfragen.

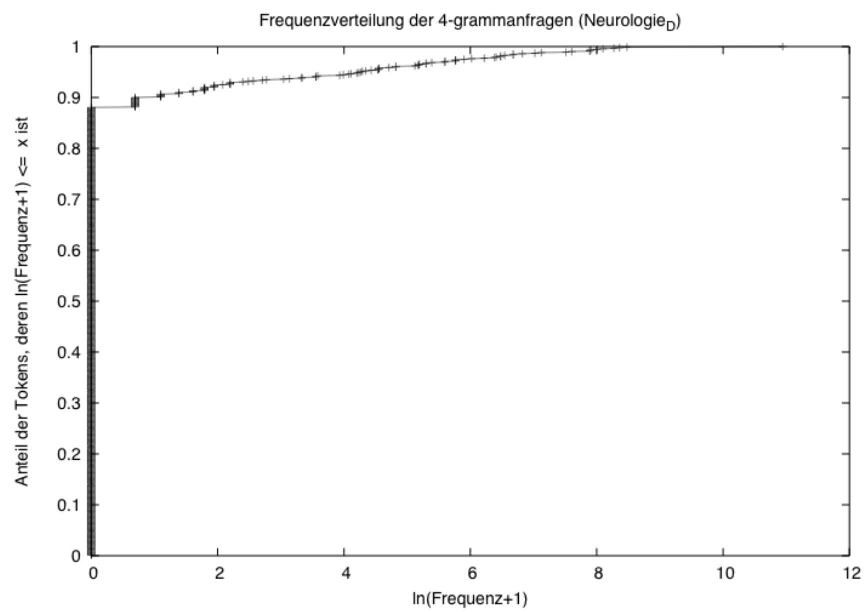


Abbildung 5.5: Beispielhafte Frequenzverteilung von 4-Grammanfragen.

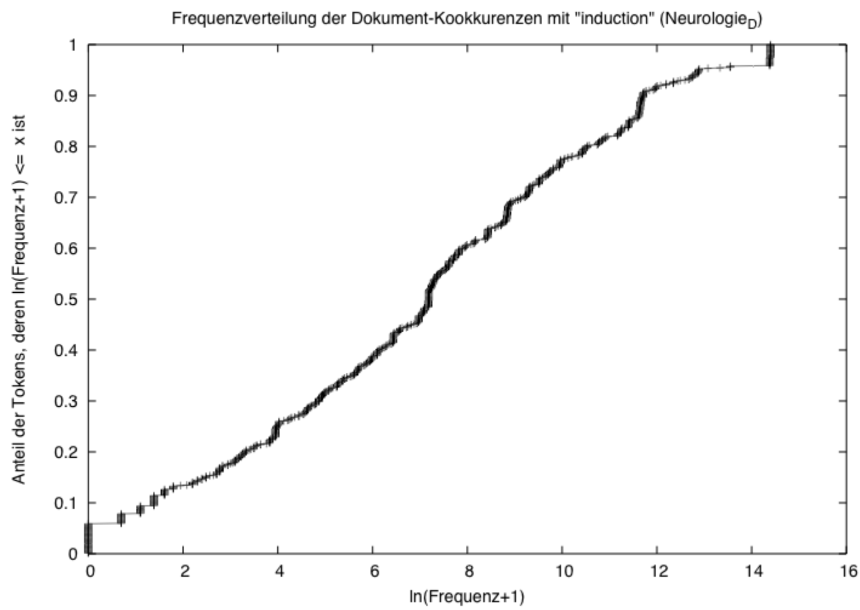


Abbildung 5.6: Beispielhafte Frequenzverteilung von Dokument-Kookkurenzen.

Fehler, werden bei Bigrammanfragen 3, bei Trigrammanfragen 2, bei 4-Gramm-Anfragen 0 und bei Dokument-Kookkurenzen in beiden Varianten des Experiments alle Kandidaten richtig zugeordnet. In keinem der Fälle wird der falsche Kandidat ausgewählt, sondern eine Auswahl wird durch Nullwerte in beiden Fällen verhindert.
Die Ergebnisse führen zu folgenden Hypothesen. Das indexierte Web von heute ist groß genug, um mit einfachen Dokument-Kookkurenzen einen Beitrag für die OCR-Nachkorrektur zu leisten. Hingegen bei n-Gramm-Techniken wird mit zunehmenden n die Datenmenge dünner, so dass jenseits von Trigrammen keine für die OCR-Nachkorrektur brauchbare Resultate zu erwarten sind.
In [61] wurde auf einem anderen Teilausschnitt des in 2.1.2 beschriebenen Korpus mit Trigrammen experimentiert. Alle korrigierbaren OCR-Fehler[4] aus 10 englischen Themengebieten des Korpus wurden bzgl. ihres Trigrammkontexts untersucht. Die beschriebene Datendichte deckt sich mit meinen Beobachtungen. Ausserdem wurde in der Arbeit gezeigt, dass eine Kandidatenauswahl für diese OCR-Fehler mit Hilfe einer Kombination aus klassischem Levenshtein-Abstand, Web-Frequenzen und Trigramm-Web-Frequenzen in knapp neun von zehn Fällen richtig liegt.

5.5 Weitere Rohmaße

In der Literatur werden viele weitere Rohmaße vorgestellt, die erfolgreich zur Aufbesserung von OCR-Leseresultaten eingesetzt werden. Eine Integration in meine Nachkorrekturarchitektur erfordert lediglich ihre Transformation in Scores.
Aus der Familie der **Wort-Abstandsmaße** habe ich diverse Varianten des Levenshtein-Abstands näher betrachtet, da dieses Abstandsmaß direkt mit meiner Komponente zur Kandidatenerzeugung korrespondiert. Alternative Wort-Abstandsmaße basieren z. B. auf n-Grammen auf Zeichenebene – oft werden Trigramme verwendet – oder probabilistischen Modellen. Ein Maß, das die übereinstimmenden n-Gramme in Relation setzt, wird häufig verwendet, wenn die Kandidatenerzeugung mit einem invertierten n-Gramm-Indexfile realisiert ist, und ein Maß, das die Wahrscheinlichkeit des Übergangs von einem String in einen anderen durch einen OCR-Lauf angibt, wird häufig eingesetzt, wenn die Kandidaten mit Hilfe einer Konfusionsmatrix erzeugt werden. Weitere Wort-Abstandsmaße werden auch aus längsten gemeinsamen Teil-Strings, Prä- oder Suffixen berechnet. Ausserdem gibt es zahlreiche Methoden, aus Wörtern Feature-Vektoren zu gewinnen, so dass sie sich mit Standard-Vektorabstandsmaßen, wie etwa dem Skalarprodukt vergleichen lassen. Eine ausführliche Darstellung zu diesen Wort-Abstandsmaßen findet man in [71]
Kontextbasierte Kandidatenbewertungen beruhen entweder auf statistischen Schätzungen wie bei den oben vorgestellten Kookkurenzen und Kollokationen oder auf linguistischen Informationen. In [61] wurden bspw. OCR-Texte vorab mit einem Tagger annotiert und bei der Kandidatenauswahl wurde die

[4] insgesamt 589 OCR-Fehler

Eignung der lexikalischen Kategorie jedes Kandidaten an der fraglichen Stelle mitberücksichtigt. Die Annotations-Resultate derzeit verfügbare Tagger sind allerdings noch zu lückenhaft, um einen wertvollen Beitrag in der Kandidatenkür zu leisten.
Zum Teil können auch **Zusatzinformationen der OCR-Engines** zur Nachkorrektur herangezogen werden. Manche OCR-Engines liefern zu jedem Wort schon fertige Kandidatenlisten mit Bewertungen, so dass zur Integration in meine Software nur ein simpler Transformationsschritt erforderlich ist. Manche OCR-Engines liefern zu jedem erkannten Wort eine Konfidenzwert, der angibt, wie sicher sich die Engine bei der Erkennung war. Berechnet man daraus einen Score, kann dieser zwar nicht bei der Kandidatenkür helfen, jedoch bei der Bestimmung der Korrekturgrenze. Eine weitere Variante sind Konfidenzwerte zu jedem erkannten Einzelzeichen. In Kombination mit der Zeichenzusammensetzung der Korrekturkandidaten können daraus Scores gebildet werden.
Bild-Abstandsmaße können durch einem Rückgriff auf die Pixelrepräsentation der Wörter ebenfalls in der Nachkorrektur eingesetzt werden. In [30] wird bspw. ein Kandidatenauswahlverfahren beschrieben, das visuelle Ähnlichkeiten zu anderen erkannten Wörtern miteinbezieht, die gleiche Teil-Strings enthalten.

5.6 Kombination von Scores

5.6.1 Kombinationsmodell

In [81] werden drei Effekte vorgestellt, die man sich bei der Kombination verschiedener IR-Retrieval-Systeme mehr oder minder erhofft; die Effekte lassen sich auch auf meine Aufgabenstellung übertragen, der Kombination von Scores und damit einer Kombination von Rohmaßen, aber auch einer Kombination von OCR-Engines:

- **Skimming Effect**. Verschiedene Wissensquellen liefern zwar die gleichen relevanten Kandidaten, aber weitere, verschiedene irrelevante Kandidaten. Durch eine Kombination wird die Ergebnismenge geglättet, indem die irrelevanten Kandidaten, die nur in einer Ergebnismenge enthalten sind entfernt werden.

- **Chorus Effect**. Schlagen mehrere Wissensquellen den gleichen relevanten Kandidaten vor, wird die Trennlinie zwischen relevant und irrelevant schärfer.

- **Dark Horse Effect**. Eine Wissensquelle liefert bei einigen Votings ein ungewöhnlich gutes (bzw. auch schlechtes) Ergebnis und kann durch die Kombination an dieser Stelle bevorzugt (bzw. gedrosselt) werden.

In [81] werden die Grenzen einer klassischen, linearen Kombination untersucht. Primär sich mit einer linearen Kombination der Chorus Effect erzielen. Hingegen der Dark Horse Effect ist durch dieses Fusionsmodell vollkommen

ausgeschlossen. Mit meinem Ansatz, alle Rohmaße in Scores zu vereinheitlichen, lassen sich auch überlegene, vorgeschlagene Kombinationsmöglichkeiten wie Neuronale-Netze realisieren. In meiner Referenzimplementierung beschränke ich mich jedoch auf ein lineares Kombinationsmodell:

$$score_{combined} := \sum_{scores} w_i \cdot score_i$$

mit $\sum_{scores} w_i = 1$, so dass $score_{combined}$ selbst wieder im Intervall [0; 1] enthalten ist.
Alternativ verwende ich noch ein zweites, noch einfacheres Kombinationsmodell, das bereits in der Einleitung zu diesem Kapitel beschrieben wurde. Sortiert wird nach einem primären Score und nur im Zweifelsfall wird zur Desambiguierung ein Sekundär-Score herangezogen. Mit diesem Kombinationsmodell können auch heute schon, trotz restriktierter Zugriffsmöglichkeit, Kollokations- und Kookkurenz-Scores aus dem Web als Sekundär-Scores eingesetzt werden.

5.6.2 Implementierung

In der Referenzimplementierung ist eine lineare Kombination zweier Scores realisiert.

$$score_{combined} := \alpha \cdot score_0 + (1 - \alpha) \cdot score_1$$

In der GUI werden die beiden Scores aus einer Liste aller aktuell verfügbaren Scores ausgewählt.

Die Gewichtung kann direkt mit einer Scroll-Bar eingestellt werden. Eine weitere Möglichkeit ist eine Suche der besten Einstellung durch Iteration des Intervalls [0; 1] in n Schritten. In jeder Iteration wird die im Abschnitt 5.7.2 vorgestellte Optimierung vorgenommen, und davon wiederum die beste Einstellung von α gesucht. Die Abarbeitung dieser geschachtelten Schleife erfolgt nicht in Echtzeit. Auf meinem Arbeitsplatzrechner dauert eine Optimumsuche in 20 Schritten bei der durchschnittlichen Teilkorpusgröße von ca. 8000 OCR-Tokens und einer durchschnittlichen Länge der Kandidatenliste von ca. 10 etwas mehr als eine halbe Minute. Eine Beschleunigung könnte unter Einsatz eines Hill-Climbing-Algorithmus erreicht werden, da der Kurvenverlauf bei den durchgeführten Tests meist nur ein einzelnes Maximum aufweist.
Das Kombinationsmodell mit Primär- und Sekundär-Score wird in meiner Implementierung nicht über eine graphische Oberfläche gesteuert, sondern die beiden werden schon bei Programmstart festgelegt.

5.7 Anwendung von Scores

Der kombinierte Score wird zu drei Aufgaben herangezogen:

- Sortierung aller Korrekturkandidaten eines OCR-Tokens
- Sortierung aller OCR-Tokens bzgl. einer Güteeinschätzung von Korrekturvorschlägen

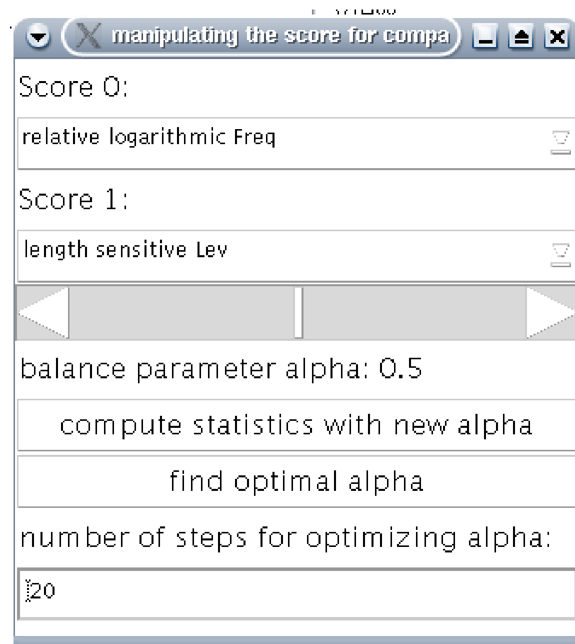


Abbildung 5.7: Score-Zusammensetzung.

- Konfidenzwertberechnung

5.7.1 Sortierung der Kandidatenliste

Bei der Sortierung von Korrekturkandidaten eines OCR-Tokens spielt die Bestimmung des Spitzenkandidaten $topCand(w^{ocr})$ eine besondere Rolle, da dieser bei Ausführung eines Korrekturvorschlags an die Stelle von w^{ocr} tritt. Falls die Kandidatenliste leer ist, gilt $w^{ocr} = topCand(w^{ocr})$. Der kombinierte Score wird künstlich auf 0 gesetzt. Dieser Fall tritt nur auf, wenn weder das OCR-Token noch ein benachbarter Korrekturkandidat im Lexikon enthalten sind. Ist das OCR-Token selbst in der Kandidatenliste enthalten, besitzt es einen gewöhnlichen kombinierten Score und ist in der Regel auch Spitzenkandidat.
In der Referenzimplementierung bildet die Klasse `Cand` einen Datentyp für Kandidaten. Diese Klasse implementiert das Java-Interface `Comparable`, indem die kombinierten Scores miteinander verglichen werden. Dadurch kann direkt auf implementierte Standardalgorithmen der Java-API für Sortierung, Maximumbestimmung, etc. zurückgegriffen werden.

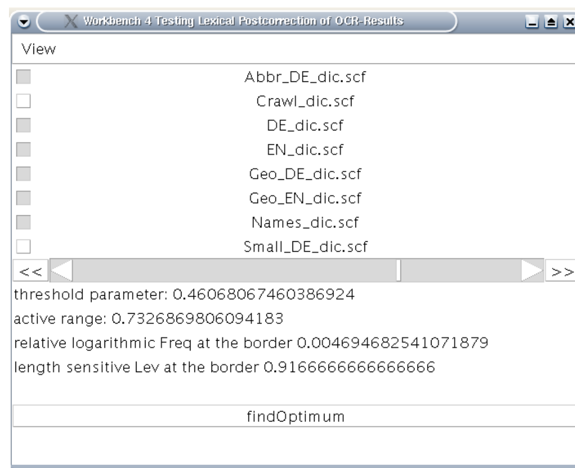


Abbildung 5.8: Bestimmung des Korrekturgrenzpunts.

5.7.2 Sortierung aller Korrekturvorschläge

Eine aggressive Korrekturkomponente, die alle Korrekturvorschläge ausführt, würde das Gesamtergebnis durch zu viele infelicitous corrections mehr zerstören als verbessern. Daher ist es sinnvoll, alle Korrekturvorschläge nach einer Güteeinschätzung zu ordnen und einen Grenzwert $score_{border}$ zu bestimmen, bis zu dem Korrekturen mehr helfen als schaden.
Der kombinierte Score liefert eine solche Ordnung, indem die Spitzenkandidaten jedes OCR-Tokens miteinander verglichen werden, d. h. $w_1^{ocr} > w_2^{ocr}$ falls $topCand(w_1^{ocr}) > topCand(w_2^{ocr})$.
In der Referenzimplementierung wird mit der Klasse `Record` ein Datentyp für OCR-Tokens zusammen mit ihrer Kandidatenliste gebildet. Auch diese Klasse implementiert das Java-Interface `Comparable`.
In der Trainingsphase werden alle OCR-Tokens entsprechend dieser Ordnung in einer Liste aufgereiht. Jeder Punkt der Liste lässt sich als Grenzpunkt einstellen, bis zu dem alle Korrekturen ausgeführt werden. Parallel dazu wird die Gesamtzahl der Fehler ermittelt. Mit der permanenten Überwachung der Gesamtauswirkung kann ein optimaler Grenzpunkt ermittelt werden. Optimal heißt in diesem Zusammenhang also eine minimale Gesamtfehlerbilanz bei automatischer Ausführung der Korrekturen bis zum Grenzpunkt. In der graphischen Oberfläche sind die Grenzpunktverschiebung und die Optimumsuche mit einem Schieberegler und einem Button realisiert. Im Allgemeinen kann es mehrere optimale Punkte geben, die in der Referenzimplementierung in einer Liste gespeichert werden; davon wird immer der erstgefundene eingestellt.

Sowohl die Grenzpunktverschiebung als auch die Optimumsuche laufen bei den gewählten Teilkorpusgrößen (ca. 8000 Wort-Tokens) für Benutzer scheinbar ohne Rechenzeitverzögerungen.

Alternativ sind auch andere Optimierungskriterien denkbar. Die Optimierung lässt sich dahingehend verallgemeinern, so dass zu möglichst vielen OCR-Tokens innerhalb der ersten n Korrekturvorschläge auch der richtige enthalten ist. Während sich der Spezialfall $n = 1$ für eine automatische Korrektur eignet, bietet sich für eine interaktive Korrektur $n > 1$ an. Eine weitere Idee ist, heuristische Relevanzschätzungen zu den Wörtern in die Fehlerbilanz miteinzubeziehen, so dass z. B. Fehler auf Nomen schwerer wiegen als Fehler auf Funktionswörtern.

5.8 Konfidenzwertberechnung

Der Grenzpunkt teilt die Liste der OCR-Tokens bzgl. der Korrektur in einen aktiven und einen passiven Teil. Der niedrigste Score, bei dem es noch gewinnbringend ist, den Vorschlag auszuführen, wird mit $score_{active}$ bezeichnet. Der höchste Score, bei dem eine Ausführung des Vorschlags nicht mehr gewinnbringend ist, wird mit $score_{passive}$ bezeichnet.

$$score_{border} := \frac{score_{active} + score_{passive}}{2}$$

An den Rändern ist $score_{border}$ mit 0 bzw. 1 definiert. Der numerische Wert dieses Grenzpunkts hängt stark von Eigenschaften des zu korrigierenden Dokuments ab. Um diesen Nachteil zu begegnen, wird in 3.2.2 ein anwendungsübergreifender Konfidenzwert definiert. Unter Zuhilfenahme von $score_{border}$ lässt sich der kombinierte Score jedes OCR-Tokens und auch jedes Kandidaten in einen solchen Konfidenzwert umrechnen:

$$conf := \frac{score_{combined}}{score_{border}}$$

Im Sonderfall $score_{border} = 0$ wird $conf := 2$ definiert, so dass alle Korrekturvorschläge empfohlen werden.

Kapitel 6

Fehlerklassifikation

6.1 Motivation

Aufgabe der Nachkorrektur-Software ist, bei der Feineinstellung verschiedener Einflussparameter wie z. B. Selektion geeigneter Lexika oder Gewichtung von Frequenzinformation für die Kandidatenauswahl behilflich zu sein. Um die Effekte der Parametereinstellungen genau beobachten zu können, wurden die Nachkorrekturfehler in Klassen unterteilt. Die Fehlerklassifikation wurde erstmals in [75] vorgestellt.
Die Klassifikation bezieht sich auf eine automatische Nachkorrektur. Nach der Korrektur liegt das Token w^{cor} vor:

$$w^{cor} := \begin{cases} w^{ocr} & falls\ conf(w^{ocr}) \leqslant 1 \\ v^{cand}_{max} & sonst \end{cases}$$

Das korrespondierende Token aus der Groundtruth heißt w^{orig}. Ein Fehler liegt vor, falls $w^{cor} \neq w^{orig}$.

6.2 Fehlerklassen der automatischen Nachkorrektur

Für die Fehlerklassifikation wird das verwendete Lexikon D untersucht, ob es das Groundtruth-Token enthält $w^{orig} \in D$ und ob es das OCR-Token enthält $w^{ocr} \in D$. Das Lexikon D kann dabei auch eine Vereinigung mehrerer Einzellexika sein. Vereinfachend wird für meine lexikonzentrierte Fehlerklassifikation angenommen, dass sobald ein OCR-Token im Lexikon gefunden wird, es auch von keinem anderen Korrekturkandidaten mehr verdrängt wird:

$$\mathfrak{A} := w^{ocr} \in D \Rightarrow w^{cor} = w^{ocr}$$

Annahme $\mathfrak{A}$ ist erfüllt, solange die Kandidatenauswahl maßgeblich an Hand eines Abstandsmaßes zu Lexikonwörtern erfolgt. Im Anschluss an die Vorstellung der Klassifikation wird $\mathfrak{A}$ weiter diskutiert.

Rechts der Knoten im Baum sind die Eigenschaften eingetragen, die jeweils für den gesamten Subbaum gelten. Die Markierung $w^{orig} \neq w^{cor}$ am Wurzelknoten besagt beispielsweise, dass der gesamte Baum die Fehler der Nachkorrektur zergliedert. Die Blätter des Baums beschreiben sieben disjunkte Fehlerklassen:

1. **Falscher Freund (false friend)**. Falsche Freunde sind eine nur schwer aufzuspürende Fehlerklasse. Die Nachkorrektur hat keinen Anlass, an dem vorgefundenen, falschen OCR-Token zu zweifeln, da es lexikalisch ist. Entdeckungsstrategien sind die Verwendung mehrerer OCR-Engines, die Berücksichtigung von Konfidenzwerten der Wort- bzw. Charakter-Erkennung der OCR-Engine selbst und stärkere Gewichtung von Kontextinformationen. Eine Vermeidungsstrategie ist die Verkleinerung des Lexikons. Es gilt, je kleiner das Lexikon D ist, desto geringer ist die zu erwartende Anzahl falscher Freunde.

2. **Zu vorsichtig (too cautious)**. Das korrekte Wort ist Spitzenkandidat der Vorschlagsliste. Allerdings wird die Korrektur nicht ausgeführt, da der Konfidenzwert zu gering ist $conf(w^{ocr}) < 1$. Vermeidungsstrategie ist eine mutigere Nachkorrektur, die alle Konfidenzwerte erhöht. Allerdings verhält sich die Fehlerklasse der unglücklichen Korrekturen (5) antagonistisch zu dieser Klasse. Die Einstellung des Konfidenzwertes $conf$ muss zwischen diesen beiden Klassen austariert werden.

3. **Falscher Kandidat und falsche Korrekturgrenze (wrong candidate and threshold)**. Das richtige Wort ist im Lexikon enthalten. Allerdings ist es nicht auf die Position des Spitzenkandidaten der Liste gerankt. Ausserdem sind die Konfidenzwerte aller Kandidaten des OCR-Tokens zu niedrig, so dass keine Korrektur vorgenommen wird. Fehlt das richtige Wort ganz in der Kandidatenliste, wurde es entweder zu stark von der OCR-Engine verunstaltet oder die Grobfilterung für die Kandidatenliste ist mangelhaft. Ansonsten ist die Vermeidungsstrategie eine verbesserte Konfidenzwertberechnung, d. h. geeignetere und/oder mehr Faktoren zur Kandidatenbewertung sowie eine bessere Kombination dieser Faktoren.

4. **Falscher Kandidat (wrong candidate)**. Die Nachkorrektur erkennt richtig, dass ein Korrekturvorschlag ausgeführt werden soll und auch das richtige Wort ist in Lexikon enthalten, jedoch nicht an erster Stelle der Kandidatenliste. Auch in diesem Fall sollte die Grobfilterung überprüft werden, wenn das richtige Wort in der Kandidatenliste fehlt. Ansonsten ist die Vermeidungsstrategie wie für die vorherige Fehlerklasse (3) eine verbesserte Konfidenzwertberechnung.

5. **Unglückliche Korrektur (infelicitous correction)**. Korrekturvorschläge, die ausgeführt werden, obwohl das Groundtruth-Token nicht im Lexikon war, führen immer zu einem Fehler ($w^{orig} \notin D$). Der drastischere

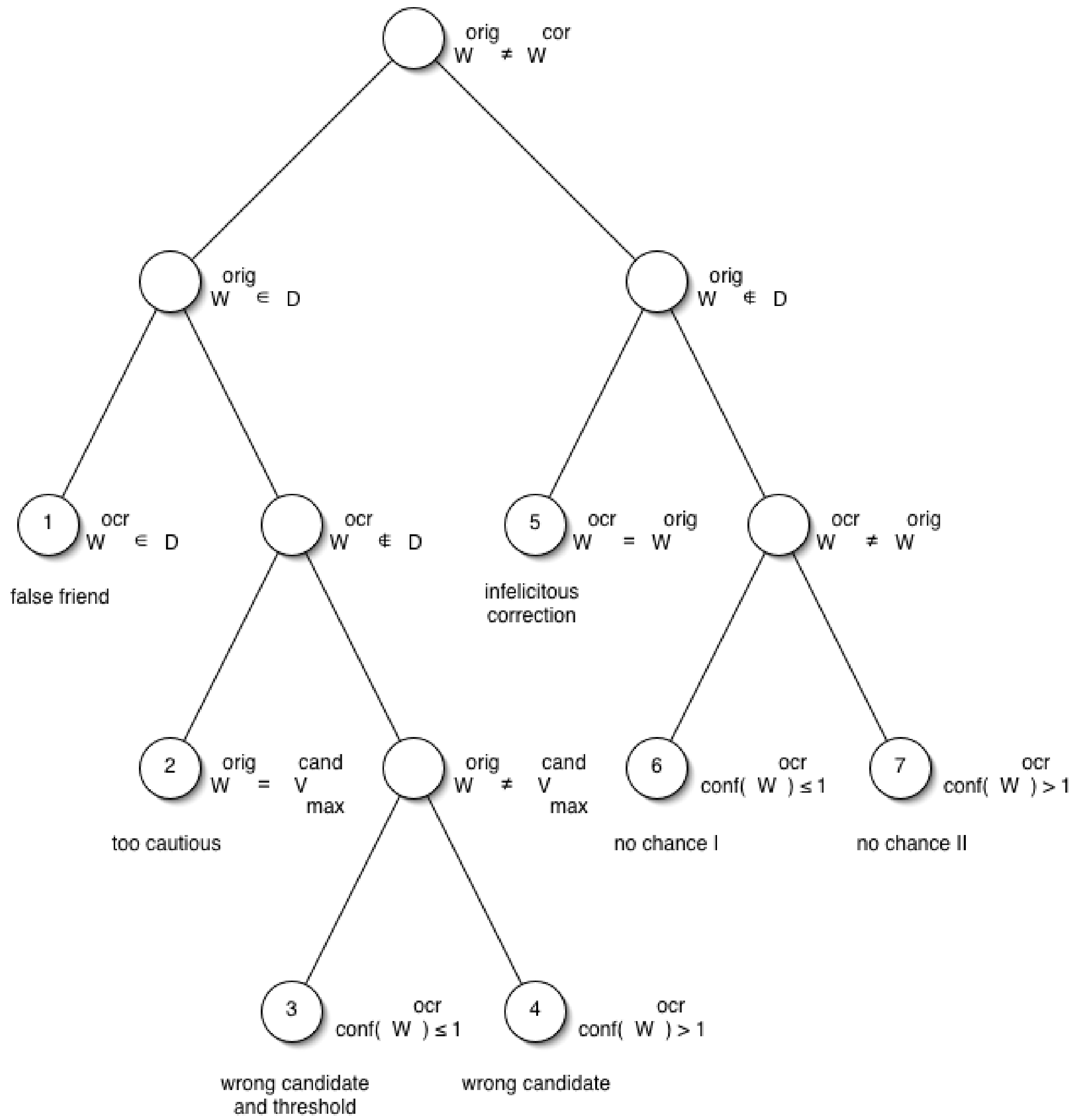


Abbildung 6.1: Fehlerklassifikation.

Fall tritt ein, wenn die OCR-Engine das Originaltoken bereits richtig gelesen hat $w^{ocr} = w^{orig}$. Dadurch verschlechtert die Nachkorrektur das Endergebnis. Vermeidungsstrategien sind die Vergrößerung des Korrekturlexikons und eine vorsichtigere Berechnung des Konfidenzwertes. Allerdings steigen mit einer Lexikonvergrößerung die potentiellen, false friends (1) und mit einer vorsichtigeren Konfidenzwertberechnung die potentiellen Fehler der Klasse too cautious (2).

6. **Unvermeidbar I (no chance I)**. Die Nachkorrektur wird nicht aktiv, obwohl eine Fehlerkennung vorliegt. Da aber das richtige Wort nicht im Lexikon und daher auch nicht in der Kandidatenliste ist, ist die Nachkorrektur ohnehin chancenlos. Die einzige Vermeidungsstrategie ist eine Vergrößerung des Korrekturlexikons D.

7. **Unvermeidbar II (no chance II)**. Die Nachkorrektur erkennt richtig, dass ein Korrekturvorschlag ausgeführt werden soll und wird aktiv. Es besteht jedoch keine Chance richtig zu korrigieren, da das originale Wort nicht im Lexikon enthalten ist $w^{orig} \notin D$. Auch hier ist die einzige Vermeidungsstrategie eine Vergrößerung des Korrekturlexikons D.

Änderungen an den Einflussgrößen und deren Auswirkung auf die Fehlerverteilung lassen sich folgendermaßen zusammenfassen:

- **Lexikongröße**. Je größer das Korrekturlexikon desto geringer die Fehleranzahl des gesamten rechten Astes (Klasse (5), (6) und (7)), aber desto größer ist auch die Anzahl falscher Freunde, Klasse (1). Dies ist allerdings kein streng funktionaler Zusammenhang, sondern eine statistische Gesetzmäßigkeit. Die Fehlerverteilung innerhalb der restlichen Klasse (2), (3) und (4) bleibt im statistischen Sinne konstant.

- **Konfidenzwertgröße**. Bei einer systematischen Vergrößerung aller Konfidenzwerte durch eine lineare Transformation sinken die Fehlerzahlen der Klasse (2), während die Zahlen der Klasse (5) im Gegenzug steigen. Dieser Zusammenhang ist statistischer Natur. In den Klassen (3) und (4) sowie (6) und (7) findet eine Verschiebung der Fehlerzahlen statt, da zwischen ihnen eine funktionale Abhängigkeit besteht. Die beiden Summen der Fehler aus den Klassen (3) und (4) sowie den Klassen (6) und (7) bleiben echt konstant.

- **Zusammensetzung und Berechnung der Konfidenzwerte**. Durch Modifikation dieser Einflussgrößen können sich die Fehlerzahlen aller Klassen bis auf (1) verändern, und diese Ausnahme ist auch nur durch die vereinfachende Annahme $\mathfrak{A}$ begründet. Das primäre Augenmerk für eine Parameteroptimierung liegt auf Klasse (4). Ausserdem stehen die Auswirkungen auf die Klassen (2), (5), und (5) im Interesse. Die Fehler der beiden Klassen (6) und (7) sind mit diesen Maßnahmen nicht zu bekämpfen; ihre Summe bleibt stets konstant.

Die Fehlerklassen lassen sich auch bzgl. der Aktivität der Nachkorrektur gruppieren:

- die Nachkorrekturkomponente korrigiert aktiv: (4), (5) und (7)
- die Nachkorrekturkomponente bleibt passiv: (1), (2), (3) und (6)

Die vereinfachende Annahme $\mathfrak{A}$ trifft bei vernünftigen Parametereinstellungen einer lexikonzentrierten Korrektur immer zu. Ich zeige an Hand eines Beispiels, wie mit extremen Parametereinstellungen $\mathfrak{A}$ auch innerhalb meiner Software ausgehebelt werden kann: Die OCR-Engine liest `nukleocytoplasmatischer` und dieses Wort steht auch so im Dokument. Im Crawl-Lexikon ist dieses Wort mit Frequenz eins enthalten. Allerdings steht `nukleozytoplasmatischer` mit Levenshtein-Abstand eins ebenfalls im Lexikon. Es hat einem entsprechend geringen längensensitiven Levenshtein-Abstand und eine deutlich höhere Frequenz. Betrachtet man die Gesamtfehlerzahl, ist eine extreme Gewichtung der Frequenzinformation weit von der optimalen Einstellung entfernt. Wählt man trotzdem diese Gewichtung, wird ein richtig gelesenes Wort, das zudem noch im Lexikon enthalten ist, verdrängt. Es entsteht an dieser Stelle ein Fehler, der schwer in meine Klassifikationsschema einzuordnen ist. Ohne $\mathfrak{A}$ kollabieren (2), (4) und (5) zu einer Fehlerklasse. Das Beispiel zeigt, dass das Klassifikationsschema für die Untersuchung einer lexikalischen Nachkorrektur maßgeschneidert ist; eine Korrektur, die andere Faktoren in den Mittelpunkt stellt, erfordert eine Adaption der Klassifikation.

6.3 Vergleich mit anderen Fehlerklassifikationen

In der Literatur zum Themenfeld OCR habe ich weitere, grobschichtigere Fehlerklassifikationen gefunden.
Im Bereich der Fehlererkennung werden häufig *real word errors* von *non-word errors* unterschieden. Real word errors entsprechen meiner Klasse der falschen Freunde und non-word errors gelten als leicht auffindbare Fehler, die durch eine erfolglose Lexikonanfrage entdeckt werden. Die Schwäche dieser Klassifikation liegt im ungerechtfertigten, impliziten Allgemeingültigkeitsanpruch des Lexikons. Daher halte ich auch die Namen der beiden Klassen für ungeeignet. Ich will den Kritikpunkt zeigen, indem ich ein typisches Beispiel zur Einführung dieser Klassifikation näher beleuchte. In [43] werden z. B. non-word errors mit dem Erkennungsfehler `Graffe` an Stelle von `Giraffe` eingeführt. Da aber Graffe ein deutscher Nachname ist und z. B. Google mehr als zweitausend Fundstellen dazu im Web angibt, ist die Bezeichnung non-word in meinen Augen problematisch.
In vielen Wissenschaftsdisziplinen werden erzielte Ergebnisse oft in einer Vierfeldertafel mit den Dimensionen Handeln (aktiv/passiv) und Resultat (positiv/negativ) ausgewertet. Innerhalb dieser Tabelle werden die beiden Fehlerklassen *false positive* und *false negative* unterschieden. Diese Fehlerklassifikation ist hauptsächlich aus der Medizin bekannt. In [24], einer Arbeit über OCR-

Korrektur auf einem Korpus medizinischer Publikationen, wird diese Klassifikation ebenfalls eingesetzt, aber auch ohne Bezug zur Medizin wird diese Klassifikation in der Fehlererkennung ebenso wie in der Fehlerkorrektur eingesetzt, z. B. in [42]. Übertragen auf eine OCR-Nachkorrektur sind also false positives Fehler im Endresultat, bei denen die Nachkorrektur aktiv wurde, und false negatives sind Fehler im Endresultat, bei denen die Nachkorrektur passiv blieb. Die Fehler meiner Klassen (4), (5) und (7) sind also false positives und Fehler meiner Klassen (1), (2), (3), und (6) sind false negatives. Bei dieser Klassifikation bleibt unklar, ob ein Fehler der Nachkorrekturkomponente anzulasten ist, wie z. B. bei (4), oder andere Ursachen hat, wie z. B. bei (7).
In der Fehlerklassifikation aus [31] (Seite 5f) wird eine korrekte Erkennung der Fehler vorausgesetzt. Es werden die Fehler der Nachkorrektur klassifiziert, wobei die Nachkorrektur – anders als in meinem Bearbeitungsprozess – Bestandteil der OCR-Engine selbst ist. Es werden drei Fälle unterschieden: (a) Das korrekte Resultat ist unter den anderen Kandidaten, (b) es gibt keinen weiteren Kandidaten oder (c) das korrekte Resultat ist nicht in der Kandidatenliste enthalten. Damit entspricht (a) meinen Klassen (3) und (4). Die Klassen (b) und (c) entsprechen meinen Klassen (6) und (7), aber i. Allg. in anderer Aufteilung.

Kapitel 7

Kombination von OCR-Engines

7.1 Motivation

Indem ich das Web als Nachkorrekturlexikon nutzbar mache, verbessere ich zwar die Gesamtfehlerbilanz, doch bei näherer Inspektion der Fehler sieht man einen deutlichen Anstieg der falschen Freunde. Mit einer Kombination von OCR-Engines ziele ich primär auf ein Absenken dieser Fehlerklasse; ausserdem erhoffe ich mir auch weniger Fehler der Klasse falscher Kandidaten, da sich auch die Kandidatenkür auf die Stimme mehrerer Engines stützt. Es kommt mir die überraschende, in [42] beschriebene Beobachtung zu Gute, dass die Spitzengruppe kommerzieller OCR-Engines zwar vergleichbare Genauigkeitsresultate erzielt, tendenziell die Lesefehler jedoch an verschiedenen Stellen begehen.

7.2 Scorebasierte Wiederverwendung der Einzel-OCR-Komponente

Mein Ansatz zur Kombination mehrerer OCR-Engines basiert auf der Wiederverwendung meiner Software zur optimalen Gewichtung verschiedener Scores für Kandidaten-Ranking und Bestimmung der Korrekturgrenze beim Einsatz *einer* OCR-Engine.

Die Parallelität der beiden Aufgaben lässt sich an zwei Punkten festmachen:

- **Es gibt zu jedem Token eine Kandidatenliste**. Im Fall einer OCR-Engine stammt diese Liste direkt aus dem Lexikon. Im Falle mehrerer OCR-Engines stammt sie aus den Kandidatenlisten der einzelnen Engines.

- **Jeder Kandidat wird bewertet**. Im Falle einer OCR-Engine dienen dazu Scores aus diversen Rohmaßen, die mit der Software optimal kom-

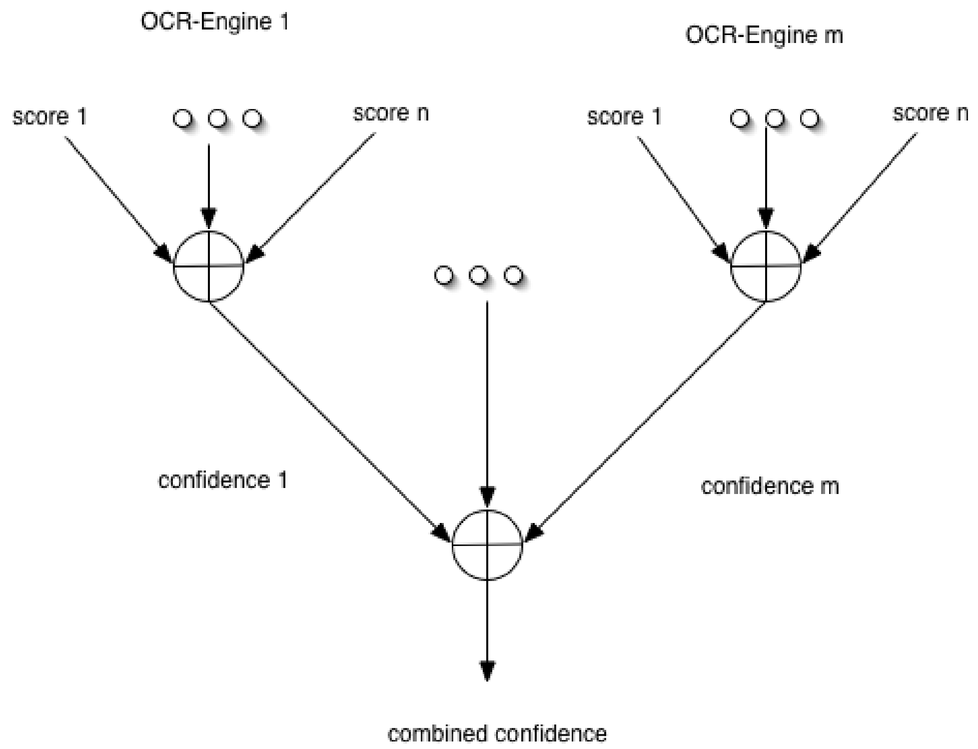


Abbildung 7.1: Prinzip der scorebasierten Software-Wiederverwendung.

biniert werden. Im Falle mehrerer OCR-Engines werden die Kandidqaten mit den Konfidenzwerten der einzelnen Engines bewertet.

Die Beschränkung meiner Software zur Parameteroptimierung auf zwei Scores überträgt sich auch auf die Anzahl kombinierbarer OCR-Engines. Die vorgestellte Vorgehensweise enthält jedoch keine fundamentalen Gründe, die eine Hinzunahme weiterer Engines a priori ausschließt.
Für eine Wiederverwendung der Software sind vier Vorbereitungsschritte notwendig:

1. Alignierung der Leseergebnisse mehrerer OCR-Engines.

2. Getrennte Konfidenzwertberechnung für jede einzelne Engine.

3. Normalisierung der Konfidenzwerte zu Scores.

4. Mischen der Korrekturkandidatenlisten.

Im ersten Schritt werden die Leseresultate der beiden Engines mit dem in 8.3.2 vorgestellten Algorithmus aligniert. In meinem Kombinationsansatz werden nicht alle OCR-Engines symmetrisch behandelt. Von den beiden Engines wird diejenige mit der besseren Erkennungs- und Wortsegmentierungsgüte zur Master-Engine erkoren, die verbleibenden ist eine Slave-Engine. Falls kein Korrekturkandidat zum Zuge kommt, wird im Resultat das Token der Master-OCR gewählt. Ausserdem gibt die Master-Engine die Liste aller normalen Tokens für den Kombinationsschritt vor. Die Alignierungsliste wird in Reihenfolge der Master-Tokens durchlaufen. Falls einem Master-Token zwei[1] Slave-Tokens zugeordnet sind, werden die Slave-Tokens entfernt. Nach diesem Schritt sind bzgl. der Kardinalität nur noch 1-1 und 1-0 Master-Slave-Zuordnungen möglich. Die Master-Tokens behalten ihre Adressierung, die einer Positionierung innerhalb des OCR-Ausgabefiles entspricht. Die Slave-Tokens erhalten die Adressierung des ihnen zugeordneten Master-Tokens.
Nach dieser Alignierungsvorbereitung durchlaufen die verschiedenen Leseresultate getrennt die Parameteroptimierung für Score-Gewichtung und Bestimmung der Korrekturgrenze.
Das Ergebnis ist jeweils eine Korrekturkandidatenliste mit Konfidenzwerten. Für die erneute Speisung des Tools mit diesen Ergebnissen, müssen zuerst die Konfidenzwerte mit Hilfe einer linearen Transformation auf das Intervall $[0; 1]$ normiert werden.
Unter Zuhilfenahme des Adressierungsmechanismus werden Master- und Slave-Resultat anschließend wieder zusammengeführt. Bei einer 1-0 Zuordnung wird die Master-Kandidatenliste direkt übernommen, der erste Score jeweils auf den normierten Konfidenzwert und der zweite Score jeweils auf Null gesetzt. Im Fall einer 1-1 Zuordnung werden die beiden Kandidatenlisten gemischt. Der normierte Master-Konfidenzwert wird als erster Score, der normierte Slave-Konfidenzwert als zweiter Score verwendet. Fehlt einer der beiden Scores, da

[1] Bei Einsatz meines Alignierungsalgorithmus ist zwei das Maximum; i. Allg. können es aber auch mehr Slave-Tokens sein.

der Kandidat nur in einer Liste auftaucht, wird er auf null gesetzt.

Anschließend wird eine Parameteroptimierung für die Gewichtung der beiden OCR-Engines und ein Bestimmung der Gesamtkorrekturgrenze vorgenommen. Die interne Datenstruktur – und damit auch die Ein- und Ausgabeschnittstellen – wurde nur dahingehend erweitert, dass zu jedem Master-OCR-Token ein weiteres Slave-OCR-Token vorhanden sein kann. Das Endresultat ist ein XML-File mit Konfidenzwerten für die Kombination von OCR-Engines. Die resultierende Gewichtung ließe sich auch in Abhängigkeit aller eingehenden Einzel-Scores ausdrücken, so dass alle Einflussfaktoren in einer Gesamtformel versammelt sind.

7.3 Erweiterte Fehlerklassifikation

Bei Einsatz mehrerer OCR-Engines erfordert das im Kapitel 6 vorgestellte Klassifikationsschema eine Erweiterung. Ebenso wie in der Referenzimplementierung wird dabei die Kombination auf zwei Engines beschränkt, ohne damit eine weitere Skalierung prinzipiell auszuschließen.

Stimmen die Leseresultate der beiden OCR-Engines überein $w^{ocr_1} = w^{ocr_2}$, kann das alte Schema weiterverwendet werden. Für differierende Leseresultate $w^{ocr_1} \neq w^{ocr_2}$ führe ich weitere Fehlerklassen ein:

8. **OCR-Engine$_2$ korrigiert OCR-Engine$_1$ unglücklich I**. OCR-Engine$_2$ alleine unterliegt einem Fehler der Klasse no chance und dominiert das korrekte Ergebnis von OCR-Engine$_1$.
9. **OCR-Engine$_2$ korrigiert OCR-Engine$_1$ unglücklich II**. OCR-Engine$_2$ alleine unterliegt einem Fehler der Klasse false friend und dominiert das korrekte Ergebnis von OCR-Engine$_1$.
10. **OCR-Engine$_2$ korrigiert OCR-Engine$_1$ unglücklich III**. OCR-Engine$_2$ alleine unterliegt einem Fehler der Klasse wrong candidate und dominiert das korrekte Ergebnis von OCR-Engine$_1$.
11. **OCR-Engine$_1$ korrigiert OCR-Engine$_2$ unglücklich I**. OCR-Engine$_1$ alleine unterliegt einem Fehler der Klasse no chance und dominiert das korrekte Ergebnis von OCR-Engine$_2$.
12. **OCR-Engine$_1$ korrigiert OCR-Engine$_2$ unglücklich II**. OCR-Engine$_1$ alleine unterliegt einem Fehler der Klasse false friend und dominiert das korrekte Ergebnis von OCR-Engine$_2$.
13. **OCR-Engine$_1$ korrigiert OCR-Engine$_2$ unglücklich III**. OCR-Engine$_1$ alleine unterliegt einem Fehler der Klasse wrong candidate und dominiert das korrekte Ergebnis von OCR-Engine$_2$.
14. **Unvermeidbar III**. Beide Lesefehler liegen ebenso wie das Original-Token nicht im Lexikon. In der Klassifikation einer OCR-Engine bzw.

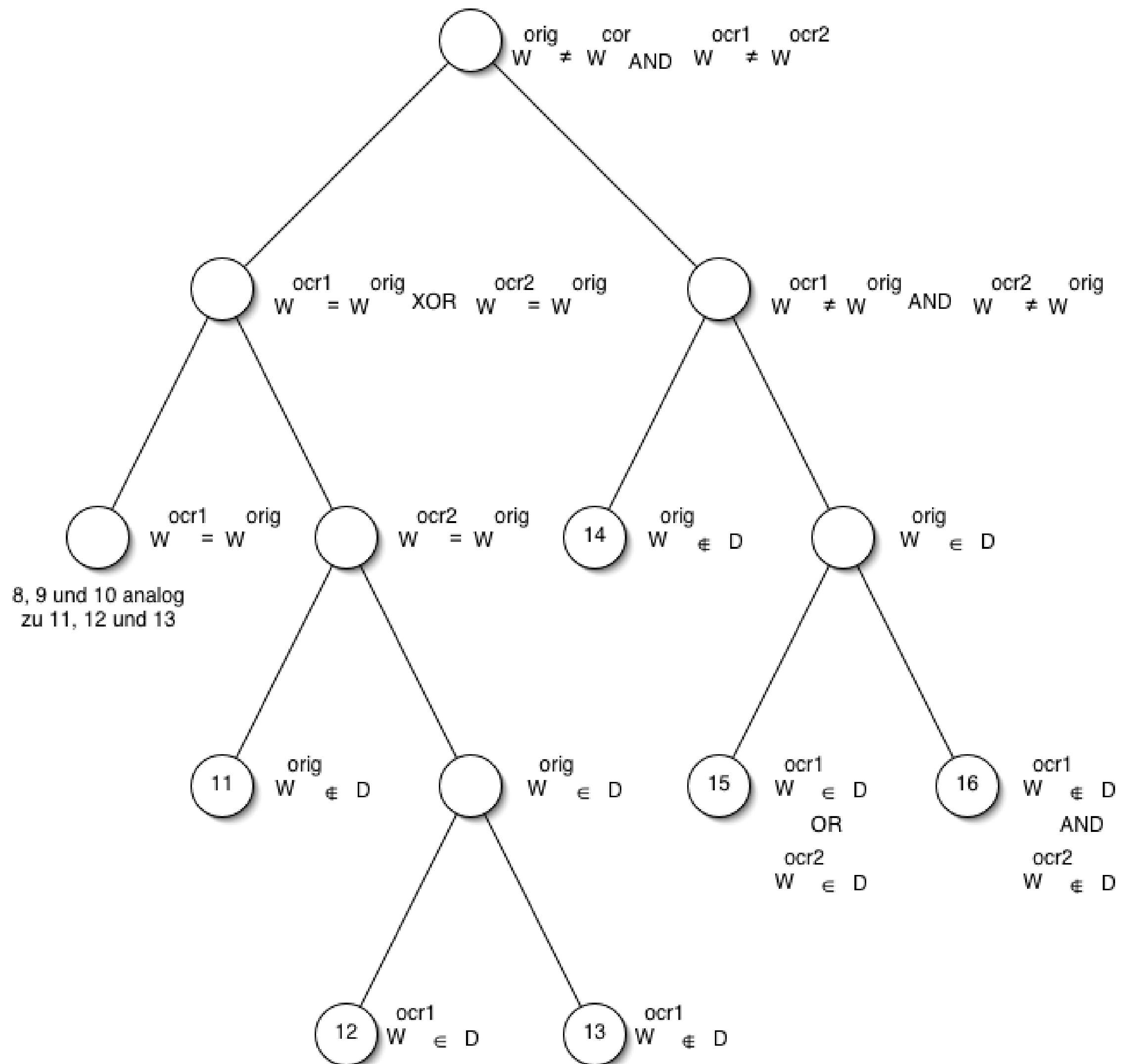


Abbildung 7.2: Erweiterte Fehlerklassifikation für zwei OCR-Engines mit unterschiedlichen Leseresultaten.

zweier, die mit einer Stimme sprechen, wird unterschieden, ob die Korrektur ausgeführt wird, oder nicht. Analog kann man auch die vorliegende Klasse nach der letztendlich bevorzugten OCR-Engine noch weiter unterteilen.

15. **Falscher Freund II**. Eine Korrektur wird verhindert, da eine der beiden OCR-Engines ein lexikalisches Token erkennt. Auch diese Klasse lässt sich weiter zergliedern nach der Verursacherin des falschen Freundes, OCR-Engine$_1$, OCR-Engine$_2$ oder beide. Im Falle eines Zusammentreffens zweier unterschiedlicher, falscher Freunde kann weiter nach der in der Ausgabe dominierenden OCR-Engine unterteilt werden.

16. **Falscher Kandidat II**. Die OCR-Engines liefern unterschiedliche Lesefehler. Der Fehler ist aber nicht unvermeidbar, da das Original-Token im Lexikon ist.

Der linker Ast der erweiterten Klassifikation – das sind die Klassen (8) bis (13) – umfasst Fehler, bei denen erst durch eine Kombination von OCR-Engines das Endresultat verschlechtert wird. Eine geeignete Vermeidungsstrategie ist ein verbessertes Voting der Engines.
Die Klassen (14) bis (16) umfassen Fehlerfälle, bei denen die Kombination zwar nicht zusätzlich schadet, aber auch nicht nutzt. Es gelten die gleichen Vermeidungsstrategien, wie im korrespondierenden Fall einer OCR-Engine: ein großes Lexikon senkt die unvermeidbaren Fehler (14), ein kleines Lexikon verhindert falsche Freunde (15) und eine verbessertes Ranking senkt die Fehler der Kandidatenwahl (16).

7.4 Alternative Ansätze

7.4.1 Lexikonbasierte Wiederverwendung der Einzel-OCR-Komponente

Ein einfacher Kombinationsansatz, der ebenfalls auf die Software zur Parameteroptimierung einzelner OCR-Engines zurückgreift, ist die Umwandlung zusätzlicher OCR-Leseresultate in jeweils ein Lexikon. Auch dieser Ansatz erfordert die Bestimmung einer Master-Engine. Dadurch ist der lexikonbasierte Ansatz sogar noch stärker asymmetrisch ausgerichtet als der scorebasierte. Da die Master-Engine die Tokenisierung vorgibt und ihr Leseergebnis bei Passivität der Korrekturkomponente als Default eingestellt ist, empfiehlt es sich, die Wahl an der Erkennungs- und Wortsegmentierungsgüte auszurichten.
Die Transformation aller weiteren OCR-Texte zu Lexika erfolgt analog zur Transformation von Web-Crawl zu Lexikon, inkl. aller Scores. Die Software, die für die Lexikongewinnung aus Web-Dokumenten erstellt wurde, kann ab dem Textextraktionsschritt direkt wiederverwendet werden. Ein weiterer Vorteil dieses Kombinationsansatzes ist die Möglichkeit zur Skalierung auf beliebig viele OCR-Engines ohne jede Modifikation der Software.

Aus der fehlenden Alignierung ergeben sich zwei Nachteile. Zum einen finden nur Wörter zueinander, die im Levenshtein-Abstand kleiner oder gleich zwei liegen. Dies ist durch die Grobfilterung der Levenshtein-Automaten bedingt. Dadurch steigt die Gefahr potentieller Fehler aus den Klassen (11), (12) und (13). Zum anderen entstehen durch Lesefehler neue Tokens im Lexikon, die Fehler der Klassen (8), (9), und (10) einschleppen können. Ein weiterer Nachteil gegenüber der scorebasierten Kombination ist die verminderte Kandidatenerzeugung. Zusätzliche OCR-Engines bringen nur selbst gelesene Tokens in die Kandidatenlisten ein, jedoch keine weiteren Tokens mit Levenshtein-Abstand eins oder zwei. Daher sind auch mehr Fehler der Klasse (15) zu erwarten. Fehler der Klasse (16) sind hingegen ausgeschlossen, da das Korrekturlexikon immer alle normalen Tokens der OCR-Engine$_2$ enthält.

7.4.2 Voting bei fehlender Übereinstimmung

In [42] wird eine weitere Kombinationsmöglichkeit für OCR-Engines vorgestellt. Der Ansatz ist auf beliebig viele OCR-Engines ausdehnbar, wird aber – wie die beiden Ansätze meiner Arbeit – der Einfachheit halber nur mit zwei OCR-Engines vorgeführt.
Die Kombination beginnt mit w Einlesen der Wort-Tupel (w^{ocr_1}, w^{ocr_2}). Leider wird dabei das Problem der korrekten Alignierung in Zusammenhang mit Wortsegmentierungsfehlern in der Veröffentlichung stiefmütterlich behandelt. Falls beide Wörter übereinstimmen $w^{ocr_1} = w^{ocr_2}$, wird das Resultat ohne weitere Prüfungen akzeptiert. Falls $w^{ocr_1} \neq w^{ocr_2}$, wird ein Voting-Verfahren gestartet. Zuerst wird eine Kandidatenliste aus den beiden Tokens unter Anwendung einer Konfusionsmatrix gebildet. Die Information, welche Kandidaten davon lexikalisch sind, wird in einen Score umgerechnet. Auch aus den Levenshtein-Abständen zu den OCR-Tokens, den Konfidenzwerten der Einzelsymbolerkennung sowie den Frequenz- und Kontextinformationen werden Scores berechnet, die alle in einem ausgeklügelten Verfahren geschachtelter Entscheidungen miteinander kombiniert werden. Schließlich wird ein Korrekturkandidat benannt.
Der prinzipieller Unterschied zu meinen Ansätzen besteht in der Vermeidung aller unglücklichen Korrekturen (Klasse 5) unter Inkaufnahme der maximalen Zahl von Fehlern aus zu großer Vorsicht (Klasse 2). Der Voting-Algorithmus für Scores ließe sich hingegen adaptieren und an Stelle meiner linearen Kombination setzen.
Ein direkter Vergleich der erzielten Nachkorrekturresultate ist nicht möglich, da sich die Experimente in zu vielen Punkten unterscheiden, wie z. B. die verwendeten OCR-Engines und Korpora oder die untersuchten Sprachen.

Kapitel 8

Alignierung

8.1 Aufgabenstellung

Die für diese Arbeit relevante, textuelle Repräsentation eines Dokuments ist eine Liste von Wörtern. Das ist entweder eine Liste von Wörtern im weiteren Sinne der OCR-Nachkorrektur (textuelle Tokens) oder eine Liste von Wörtern in meiner enger gefassten Definition (normale Tokens), d. h. die zurückgegebenen Datenstrukturen der beiden Methoden `getTextualTokens()` und `getQueryTokens()` des Interfaces `Tokenizer` (siehe 3.2.1). Die vorgestellten Algorithmen sind auch auf der Rückgabeliste der Methode `getPreTextualTokens()` und vergleichbaren linearen Textdatenstrukturen anwendbar. Aufgabe der Alignierung ist es, die Listenelemente zweier, i. Allg. verschiedener, textueller Repräsentationen eines Dokuments einander zuzuordnen. Dazu habe ich ein Java-Interface formuliert:

```
public interface Aligner {
List align(List list1, List list2);
}
```

Die Aufgabe der Alignierung stellt sich in meiner Arbeit an zwei verschiedenen Stellen:

- Alignierung einer OCR-Rückgabe mit der zugehörigen Groundtruth eines Dokuments.
- Alignierung der Leseresultate verschiedener OCR-Engines.

8.2 OCR-Fehler

Die Zuordnung wird durch folgende Typen von OCR-Fehlern erschwert:

- **Fehler auf Symbol-Ebene**, d. h. Substitution, Löschung, Einfügung, Trennung in zwei oder mehr Symbole, Verschmelzung von zwei oder mehr Symbolen sowie komplexere Symbolsegmentierungsfehler (z. B. `lm` $\mapsto$ `hii`)

- **Fehler auf Wort-Ebene**, d. h. Löschung, Einfügung, Trennung in zwei oder mehr Wörter, Verschmelzung von zwei oder mehr Wörtern sowie komplexere Wortsegmentierungsfehler, analog zu den komplexeren Symbolsegmentierungsfehlern
- **vertauschte Lesereihenfolge**

Im Appendix von [51] ist eine Statistik zur Häufigkeit der Fehler auf Symbolebene. Diesen Fehlertyp fasse ich zu einem Fehler auf Wortebene zusammen, Substitution eines Wortes durch ein anderes. Der letzte Fehlertyp spielt in dieser Arbeit eine untergeordnete Rolle, da das verwendete Korpus bewusst so gewählt wurde, dass dieser Fehlertyp nicht auftritt. Da aber in der Praxis diese Problematik eine wichtige Rolle spielt, werden am Ende des Kapitels diesbezüglich geeignete Alternativen und Erweiterungen vorgestellt. Weitere, in der Literatur oft genannte Fehlertypen sind Vertauschung zweier benachbarter Symbole oder Wörter. Doch sie entsprechen nicht der üblichen Fehlercharakteristik eines OCR-Laufs, sondern sind eher typisch für Abtippen eines Textes. Diese sog. Transpositionen lassen sich auf eine Kombination zweier Substitutionen zurückführen. Auch bei den oben aufgezählten OCR-Fehlertypen sind starke Redundanzen vorhanden. Aus den beiden Fehlertypen Symboleinfügung und -löschung ließen sich alle anderen Fehlertypen kombinieren. Mein Ziel ist es, die Betrachtungen auf eine sinnvolle Menge von Fehlertypen auf Wortebene zu konzentrieren.

8.3 Minimum-Edit-Distance-Algorithmus

Die zentrale Idee zur Lösung des Alignierungsproblems ist eine Anpassung des klassischen Minimum-Edit-Distance-Algorithmus an OCR-Fehlertypen auf Wortebene.

8.3.1 Bestimmung von Symboleditieroperationen

Ursprungsversion

In seiner einfachsten Form bestimmt der Minimum-Edit-Distance-Algorithmus den Levenshtein-Abstand ([50]) zwischen zwei Strings a und b, indem er die minimale Anzahl der Editieroperationen zählt, die notwendig sind, um a in b zu überführen. Alternative Bezeichnungen sind Editier-Abstand oder Levenshtein-Damerau-Abstand. Die Längen der Strings $|a| = n$ und $|b| = n'$ können dabei differieren. Die klassischen Editieroperationen sind Einfügung, Löschung oder Vertauschung eines Symbols. Um z. B. den String `zeitgeist` in den String `zimtgeiss` zu überführen, kann man das erste `e` löschen, zwischen `i` und `t` ein `m` einfügen und das zweite `t` durch ein `s` ersetzen. Alternativ hätte man auch drei Substitutionen `e` $\mapsto$ `i`, `i` $\mapsto$ `m` und `t` $\mapsto$ `s` durchführen können. Es gibt auch längerer Editieroperationsfolgen, die das gewünschte Ergebnis liefern, jedoch drei ist das Minimum. Dieses Minimum lässt sich mit einem Schema

Levenshtein Matrix

	_	z	i	m	t	g	e	i	s	s
_	0	1	2	3	4	5	6	7	8	9
z	1	0	1	2	3	4	5	6	7	8
e	2	1	1	2	3	4	4	5	6	7
i	3	2	1	2	3	4	5	4	5	6
t	4	3	2	2	2	3	4	5	5	6
g	5	4	3	3	3	2	3	4	5	6
e	6	5	4	4	4	3	2	3	4	5
i	7	6	5	5	5	4	3	2	3	4
s	8	7	6	6	6	5	4	3	2	3
t	9	8	7	7	6	6	5	4	3	3

Abbildung 8.1: Übergangsmatrix des Beispiels.

der dynamischen Programmierung berechnen. Dieses Schema wurde unabhängig voneinander mehrfach entdeckt, erstmals von [85]. Eine sog. Übergangsmatrix gibt in Position $m_{i-1,j-1}$ den minimalen Levenshtein-Abstand der Substrings $a[1]..a[i-1]$ und $b[1]..b[j-1]$ an. Der Abstand der Substrings $a[1]..a[i]$ und $b[1]..b[j]$, d. h. $m_{i,j}$ lässt sich rekursiv berechnen:

$$
\begin{aligned}
m_{0,0} &= 0 \\
m_{i,j} &= min \begin{cases} m_{i-1,j-1} & + \quad editOperations(a[i], b[j]) \quad \textit{Match oder Substitution} \\ m_{i,j-1} & + \quad 1 \quad \textit{Insertion} \\ m_{i-1,j} & + \quad 1 \quad \textit{Deletion} \end{cases}
\end{aligned}
$$

Die Funktion *editOperations* ist dabei folgendermaßen definiert:

$$
editOperations(a[i], b[j]) = \begin{cases} 0 & falls \quad a[i] = b[j] \quad Match \\ 1 & falls \quad a[i] \neq b[j] \quad Substitution \end{cases}
$$

Die Matrix wird mit einer quadratischen Zeitkomplexität $O(nn')$ von links oben nach rechts unten berechnet. Wäre man nur am minimalen Abstand interessiert (das entspricht dem Wert $m_{n,n'}$), könnte man den Speicherplatzbedarf auf lineare Komplexität drosseln, indem man immer nur die vorherige sowie die aktuelle Zeile im Hauptspeicher behält. Will man aber die Editieroperationen benennen, muss man die Elemente der gesamten Matrix mit der Information versehen, welche der drei Optionen minimal waren. Ist das Minimum nicht eindeutig, resultieren daraus alternative Editieroperationsfolgen. Mit der Herkunftsinformation kann am Ende der Matrix-Berechnung der Weg durch die Matrix (sog. Trace-Back-Spur) zurückverfolgt werden. Daraus resultiert eine quadratische Speicherplatzkomplexität $O(nn')$.
Da ich mehrfach in meiner Arbeit auf den Levenshtein-Abstand zurückgreife,

führe ich für dieses grundlegende String-Abstandsmaß eine Notation ein:

$$lev(a, b) := m_{n,n'}$$

Eine Variante, auf die ich ebenfalls an anderen Stellen zurückgreife, ist der längensensitive Levenshtein-Abstand:

$$lev_{length}(a, b) := \frac{2 \cdot lev(a, b)}{|a| + |b|}$$

Die Relativierung der Editieroperationen zur Länge erlaubt eine feingranulare-re Ordnung auf String-Paaren als der klassische Levenshtein-Abstand. So gilt bspw.

$$lev(\texttt{an}, \texttt{Rat}) = lev(\texttt{phosphatidylcholine}, \texttt{phosphatidyldioline}) = 2$$

In einer Alignierungsanwendung würde man intuitiv beim zweiten String-Paar die Korrektheit der Zuordnung viel höher einschätzen als beim ersten. Der längensensitive Levenshtein-Abstand hilft, diese Intuition zu quantifizieren. Es gilt:

$$lev_{length}(\texttt{an}, \texttt{Rat}) = 0.8 \gg 0.1 = lev_{length}(\texttt{phosphatidylcholine}, \texttt{phosphatidyldioline})$$

Verallgemeinerungen

Es gibt eine Reihe von Verallgemeinerungen des Levenshtein-Abstands, die zwei orthogonale Aspekte aufweisen:

- Kostenfunktion an Stelle von Abzählen der Editieroperationen.
- Einführung neuer Editieroperationen.

Der Name Editieroperation suggeriert den Arbeitsablauf, dass Texte abgetippt werden, dabei Fehler entstehen und deren Entstehung nachvollzogen wird. Der Levenshtein-Abstand lässt sich auch für eine OCR-Nachkorrektur einsetzen. Im OCR-Korpus wurde z. B. an Stelle des Worts `nahm` der String `nanm` erkannt. Dieser String ist in einem deutschen Standardlexikon nicht enthalten, jedoch drei Wörter mit Levenshtein-Abstand 1: `nahm`, `nanu` und `nano`, die von einem interaktiven Korrekturprogramm vorgeschlagen werden können. Dass hinter dem OCR-Lesefehler tatsächlich das Wort `nahm` steckt, ist viel naheliegender als die anderen beiden Optionen, da sich die Buchstaben des Paares (`n`, `h`) bzgl. ihrer Gestalt viel ähnlicher sind als die der Paare (`m`, `o`) und (`m`, `u`). Mit einer Verfeinerung des Levenshtein-Abstands[1] lässt sich erreichen, dass der Korrekturvorschlag `nahm` näher an `namn` liegt als die anderen beiden Vorschläge. Anstatt

[1] In OCR-Anwendungen sind die Editierkosten nicht immer symmetrisch, d. h. $editCosts(x, y) = editCosts(y, x)$ gilt i. Allg. nicht. Daher ist der Begriff Abstand im Sinne einer Metrik eigentlich nicht gerechtfertigt.

die Editieroperationen abzuzählen, wird eine Kostenfunktion (auch Editiergewichtsfunktion genannt) *editCosts* eingeführt.

$$m_{i,j} = min \begin{cases} m_{i-1,j-1} & + & editCosts(a[i], b[j]) & \textit{Match oder Substitution} \\ m_{i,j-1} & + & editCosts(\varepsilon, b[j]) & \textit{Insertion} \\ m_{i-1,j} & + & editCosts(a[i], \varepsilon) & \textit{Deletion} \end{cases}$$

Im Beispiel gilt $editCosts(\mathtt{n}, \mathtt{h}) < editCosts(\mathtt{m}, \mathtt{o})$ und $editCosts(\mathtt{n}, \mathtt{h}) < editCosts(\mathtt{m}, \mathtt{u})$. Die Editiergewichte der Übergangsmatrix werden mit Hilfe von Trainingsdaten bestimmt. In [67], [89] und [68] werden unterschiedliche Verfahren zur Berechnung dieser Werte untersucht.

Symbolzweiteilung und Verschmelzung zweier Symbole sind typische OCR-Erkennungsfehler, die sich leicht in das rekursive Schema integrieren lassen:

$$m_{i,j} = min \begin{cases} m_{i-1,j-1} & + & editCosts(a[i], b[j]) & \textit{Match oder Substitution} \\ m_{i,j-1} & + & editCosts(\varepsilon, b[j]) & \textit{Insertion} \\ m_{i-1,j} & + & editCosts(a[i], \varepsilon) & \textit{Deletion} \\ m_{i-1,j-2} & + & editCosts(a[i], b[j-1]b[j]) & \textit{Split} \\ m_{i-2,j-1} & + & editCosts(a[i-1]a[i], b[j]) & \textit{Merge} \end{cases}$$

Diese Adaption des Programmierschemas wird z. B. in [56] eingesetzt[2], in [73] wird zusätzlich die Doppelsymbolsubstitution verwendet[3]. In [15] werden alle diese Editieroperationen als Spezialfall von g:h Substitutionen betrachtet, 1:1 (Match oder Substitution), 0:1 (Insertion), 1:0 (Deletion), 1:2 (Split), 2:1 (Merge) und 2:2 (Doppelsubstitution).
Die Berechnung mit dem dynamischen Programmierschema wird beibehalten. Es werden zwei obere Schranken p und q für die Anzahl der beteiligten Symbole bei Quelle und Ziel eingeführt. Diese Parameter lassen sich mit Hilfe beobachteter Übergänge und gewünschtem Laufzeitverhalten adjustieren.

$$m_{i,j} = min\{m_{i-g,j-h} + editCosts(a[i-g+1] \dots a[i], b[j-h+1] \dots b[j])\}$$

$$\text{wobei} \qquad 0 \leq g \leq p \qquad \text{und} \qquad 0 \leq h \leq q$$

8.3.2 Algorithmus zur Bestimmung von Worteditieroperationen

In [95] werden Worteditieroperationen in das dynamische Programmschema auf Symbolebene mitaufgenommen, um alternative Umschreibungen gleicher Informationen aus verschiedenen Datenbanken abzugleichen. Im Sinne von [15] werden bestimmte g:h Multisubstitutionen zugelassen, wie z. B. Löschung eines Wortes. Mein Algorithmus zur Bestimmung von Worteditieroperationen setzt

[2]Einsatzziel in der Arbeit ist eine Query-Expansion für eine Volltextsuche in einer Dokumentensammlung, die mit OCR elektronisch verfügbar gemacht wurde.
[3]Die Arbeit fokussiert auf Erkennung handschriftlicher Dokumente.

hingegen Wörter an die Stelle von Symbolen im dynamischen Programmierschema. Die Überführung der Liste [`Fuchs`, `du`, `hast`, `die`, `Gans`, `gestohlen`] in die Liste [`Fuchs`, `dii`, `ha`, `st` ,`die` ,`gestohlen`] erfordert drei Worteditieroperationen: eine Substitution `du` $\mapsto$ `dii`, und ein Split `hast` $\mapsto$ `ha st` und eine Deletion `Gans` $\mapsto \varepsilon$. Für den Alignierungsalgorithmus verwende ich außerdem noch Insertions (einzelner Wörter) sowie Merges (zweier Wörter).
Den Algorithmus von Wagner-Fischer auf Wortebene anzuwenden ist nicht nur eine simple Adaption an die Eingabedatenstruktur, sondern bringt einen enormen Vorteil für den Hauptspeicherbedarf, da die Länge der Eingabe quadratisch zu Buche schlägt. Erst durch diese Problemgrößenreduktion wird das Alignierungsverfahren in der Praxis effizient anwendbar. Die maximal beobachtete Symbolanzahl pro OCR-Seite[4] ist 6 717, die korrespondierende Anzahl textueller Tokens hingegen nur 901. Der Hauptspeicherbedarf sinkt auf knapp 2% gegenüber dem Alignierungsansatz auf Symbolebene. Weiteres Zahlenmaterial folgt im Abschnitt zur Implementierung.
Die rekursive Definition der Funktion auf Wortebene sieht exakt so aus wie die Definition auf Symbolebene. Lediglich die Bedeutung der Variablen hat sich verschoben. Listen von Strings werden nun von a und b repräsentiert. Im Beispiel gilt etwa für die Länge $|a| = |b| = 6$. Außerdem gilt $a[1] = b[1] =$ `Fuchs` und $|a[1]| = 5$.

$$m_{i,j} = min \begin{cases} m_{i-1,j-1} & + & editCosts(a[i], b[j]) & \textit{Match oder Substitution} \\ m_{i,j-1} & + & editCosts(\varepsilon, b[j]) & \textit{Insertion} \\ m_{i-1,j} & + & editCosts(a[i], \varepsilon) & \textit{Deletion} \\ m_{i-1,j-2} & + & editCosts(a[i], b[j-1]b[j]) & \textit{Split} \\ m_{i-2,j-1} & + & editCosts(a[i-1]a[i], b[j]) & \textit{Merge} \end{cases}$$

Für die Editierkosten auf Wortebene wurde der klassische Levenshtein-Abstand auf Symbolebene herangezogen:

$$m_{i,j} = min \begin{cases} m_{i-1,j-1} & + & lev(a[i], b[j]) & \textit{Match oder Substitution} \\ m_{i,j-1} & + & \xi_1 + |b[j]| & \textit{Insertion} \\ m_{i-1,j} & + & \xi_2 + |a[i]| & \textit{Deletion} \\ m_{i-1,j-2} & + & \xi_3 + lev(a[i], b[j-1]b[j]) & \textit{Split} \\ m_{i-2,j-1} & + & \xi_4 + lev(a[i-1]a[i], b[j]) & \textit{Merge} \end{cases}$$

Ein Summand ξ_c, der bei allen Editiergewichten außer Match/Substitution eingerechnet wird, motiviert sich aus der Überlegung, dass bei diesen Operationen ein oder mehrere Whitespaces gelöscht oder eingefügt wurden. Allerdings sollte man $\xi_1 < \xi_3$ und $\xi_2 < \xi_4$ wählen. Bei $\xi_1 = \xi_3$ bzw. $\xi_2 = \xi_4$ hat man beim Trace-Back bei allen Insertions bzw. Deletions immer alternative Wege durch die Matrix, da ein Split bzw. Merge gleichteuer ist. Etwa im Beispiel ließe sich

[4] In der Groundtruth ist die maximal beobachtete Symbolanzahl sogar 11 796 bei nur 273 textuellen Tokens. Allerdings entstammt diese Diskrepanz einer Anomalie. Ein Großteil des Textexports besteht aus Leerzeichen, die eine rechtsbündige Spalte simulieren, d. h. die Leerzeichen werden als präsentationales Markup eingesetzt. Mein Ansatz zur Alignierung bleibt von dieser Anomalie unberührt, da Whitespaces vom Tokenisierer vorgefiltert werden.

Levenshtein Matrix

	_	Fuchs	dii	ha	st	die	gestohlen
_	0	6	10	13	16	20	30
Fuchs	6	0	5	4	5	5	8
du	9	4	2	5	6	7	11
hast	14	5	6	4	3	7	14
die	18	5	6	8	7	3	13
Gans	23	4	8	9	10	8	12
gestohlen	33	8	12	13	16	16	8

Abbildung 8.2: Übergangsmatrix des Alignierungsbeispiels.

nicht entscheiden, ob die Deletion `Gans` $\mapsto \varepsilon$ oder der Merge `die Gans` $\mapsto$ `die` die korrekte Alignierung ist. Bei $\xi_1 > \xi_3$ bzw. $\xi_2 > \xi_4$ kommen Insertions bzw. Deletions nie zum Zuge. In der Referenzimplementierung wurden verschiedene Wertebelegungen durchprobiert und mit $\xi_1 = \xi_2 = 1$ sowie $\xi_3 = \xi_4 = 2$ die besten Alignierungsergebnisse erzielt.
Bei Insertions und Deletions genügt die Länge, da für den klassischen Levenshtein-Abstand gilt:

$$lev(\varepsilon, wort) = lev(wort, \varepsilon) = |wort|$$

Die Wahl der Editieroperationen erfolgte in Hinblick auf die beobachteten OCR-Fehler einer Stichprobe des Korpus. Auf Grund der relativ einfachen Implementierbarkeit, wurde der klassische Levenshtein-Abstand für die Editierkosten auf Wortebene eingesetzt[5]. Es ist denkbar, die Editieroperationen auf Wortebene zu g:h Substitutionen im Sinne von [15] auszuweiten und auf Symbolebene trainierte, gewichtete Editierkosten zu verwenden. Ebenso ist denkbar ξ_c zu trainieren. Eine Matrix aller Gewichte auf Wortebene zu trainieren ist hingegen problematisch, da die Anzahl der Wörter theoretisch unendlich, praktisch lediglich durch Systemparameter (z. B. maximale Länge) beschränkt ist. Allerdings rechtfertigt sich die pragmatische Wahl der Editieroperationen und -kosten durch den Erfolg; die Alignierungsaufgaben im betrachteten Korpus konnten ohne nennenswerte Probleme bewerkstelligt werden.
Der Flaschenhals liegt an anderer Stelle: Speicherplatzbedarf und v. a. Laufzeit.

8.3.3 Implementierung

Naive Implementierung

Eine erste ad hoc Implementierung, mit der der Algorithmus untersucht wurde und auch die Graphiken erzeugt wurden, berechnet eine Matrix, die mit Objekten gefüllt ist.

[5]Im Korpus wurde gemessen, dass 80% der Fehler auf Worteben Levenshtein-Abstand 1 und 15% Abstand 2 haben.

LevenshteinMatrixEntry

+value: int
+isMatch: boolean
+isSplit: boolean
+isMerge: boolean
+isDeletion: boolean
+isInsertion: boolean

Abbildung 8.3: Element der Übergangsmatrix.

Da das Groundtruth-Korpus seitenweise vorliegt, wird der Algorithmus an Hand der Seite mit den meisten Wörtern überprüft. Es wurde folgende Verteilungsfunktion der Wortanzahl pro Seite im gesamten Korpus beobachtet.

Der Verlauf der Verteilungsfunktion zeigt deutlich, dass die Seite mit maximaler Wortanzahl (958 in der Groundtruth) ein guter Testkandidat für meine oberen Laufzeit- und Speicherbedarfabschätzung ist, da der Großteil der Seiten nur zwischen 200 und 500 Wörter enthält.
Getestet wurde auf meinem Arbeitsplatzrechner (Pentium III mit 2GHz Taktfrequenz und 1GB Hauptspeicher). In [16] werden Alignierungsverfahren, die auf dynamischer Programmierung basieren als extrem langsam und speicherintensiv beschrieben. Durch die seitenweise Alignierung auf Wortebene ergeben sich deutlich positivere Resultate: Die Hauptspeicherauslastung von 3% ist vollkommen unkritisch; lediglich die Laufzeit von 24 Sekunden motiviert noch einige Überlegungen zur Optimierung.

Optimierungen und Skalierungsverhalten

Da nur Dokumente im Korpus enthalten sind, die auf Grund ihrs Aufbaus weitgehend frei von Lesereihenfolgefehler sind und die OCR-Engine kaum Insertions und Deletions von Wörtern verursacht, verlaufen alle Trace-Back-Spuren sehr eng an der Diagonale der Matrix. Diese Beobachtung rechtfertigt die Optimierungsidee, nur einen engen Kanal ober- und unterhalb der Diagonale zu berechnen. Für die Alignierungen wurde die Kanalbreite schließlich empirisch auf 30 gesetzt.
Ein weiterer Optimierungsschritt ist das Ausschalten des Overheads für die Objektorientierung. An Stelle von Objekten der Klasse `LevenshteinmatrixEntry` werden Integer-Zahlenwerte gespeichert. Die obersten 5 Bits des primitiven Datentyps enthalten die 5 booleschen Informationen, die restlichen Bits enthalten den Abstandswert. Die Kombination der unterschiedlichen Informationen ist durch eine ODER-Operation auf Bit-Ebene realisiert. Ausgelesen werden die Informationen durch eine UND-Verknüpfung mit einer Bit-Maske.
Mit diesen beiden Optimierungen konnte die Laufzeit auf 0.8 Sekunden reduziert werden. Der Hauptspeicherbedarf hat sich dabei noch einmal halbiert.

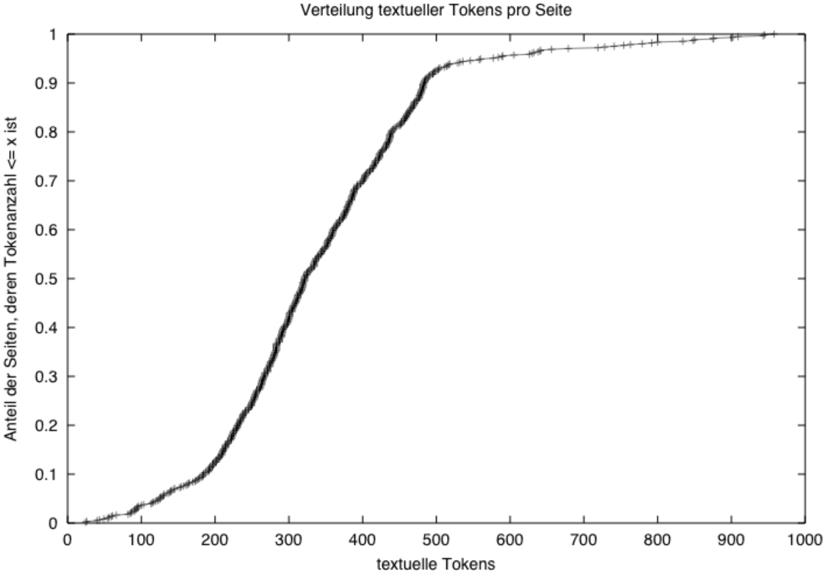


Abbildung 8.4: Verteilung textueller Tokens pro Seite.

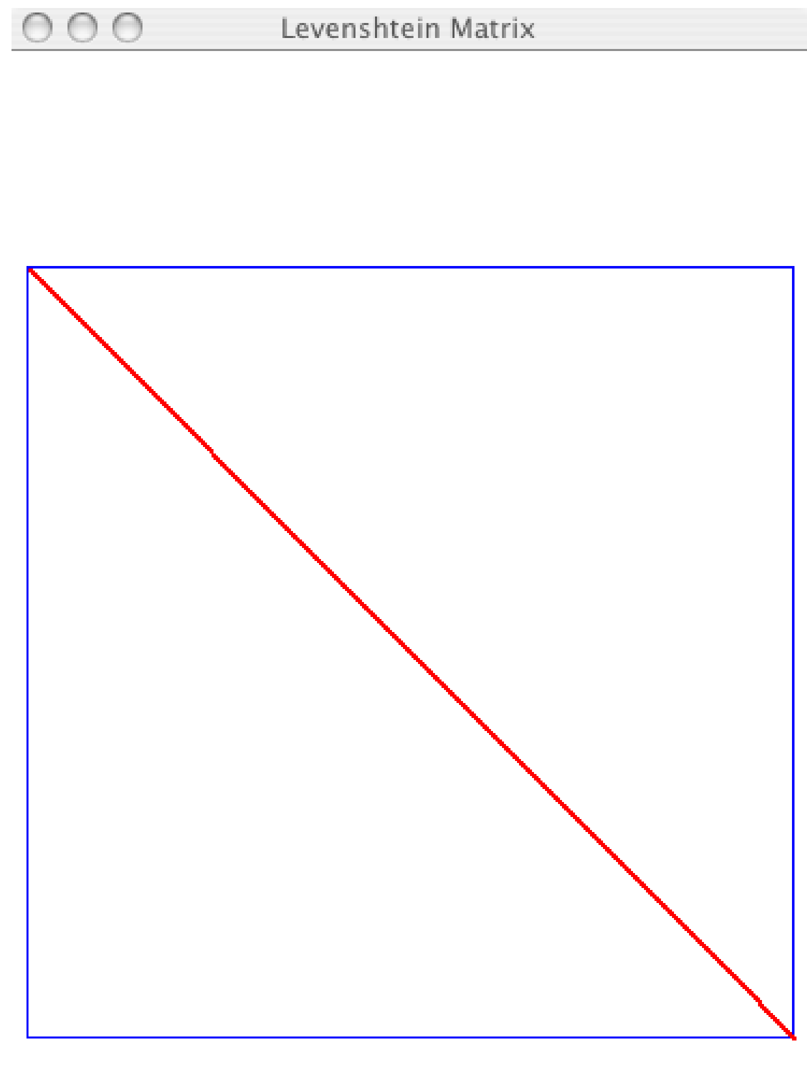


Abbildung 8.5: Trace-Back-Spur durch die Übergangsmatrix des längsten Dokuments im Korpus.

Seitenzahl	1	2	5	7	10	12	15	16	17
Memory [MB]	13	25	107	201	392	571	801	949	error
Laufzeit [sec.]	0.8	3.1	17.5	34.4	77.7	113.6	186.1	319.2	error
Kanalbreite	30	60	150	210	300	360	450	480	510

Tabelle 8.1: Skalierungsverhalten des Minimum-Edit-Distance-Algorithmus.

Da auch größere Seiten denkbar sind[6] und Aufgabenstellungen denkbar sind, bei denen eine Alignierung nicht seitenweise, sondern en bloc erforderlich ist, habe ich meine Implementierung auf Skalierbarkeit untersucht. Für den Skalierungstest habe ich die maximale Seite der Groundtruth (sowie den entsprechenden OCR-Text) mehrfach aneinander gekettet. In meiner Messreihe habe ich empirisch ermittelt, dass die Kanalbreite linear zur Wortanzahl mitwachsen muss.

Fazit ist, dass das Alignierungsverfahren bei Einzelseiten effizient einsetzbar ist und bei etwa $1,5 \cdot 10^4$ Wörtern an seine Grenzen stößt. Diese Hauptspeichergrenze ließe sich mit einigen Optimierungen noch nach oben schieben:

- Mit einer Portierung in die systemnahe Programmiersprache C kann Rechenzeit und Hauptspeicherbedarf eingespart werden, den die Virtual-Machine kostet. In C lassen sich außerdem die vielen Funktionsaufrufe durch Makros ersetzten.
- Anstelle eines zweidimensionalen Arrays bei dem man nur einen engen Kanal verwendet, kann man eine selbstverwaltete, lineare Datenstruktur verwenden.
- In meiner Implementierung verläuft der Kanal parallel zur Diagonalen der Matrix. Es genügt jedoch, am Anfang der Alignierung mit einer Breite von einem Wort zu beginnen und erst am Ende die volle Kanalbreite auszuschöpfen; das entspricht einem trichterförmigen Zuschnitt des Kanals.
- Anstelle der 27 Bit, die für den `value` verwendet werden, kann eine deutlich schlankere, maßgeschneiderte Zahldarstellung eingesetzt werden.
- Da beim rückwärtigen Durchlaufen der Matrix alternative Pfade nicht von Interesse sind, muss nicht für jede der fünf booleschen Informationen ein eigenes Bit codiert werden. Mit einer priorisierenden Ordnung auf den 5 Operationen kann die Information in 3 Bits abgespeichert werden.

Robustheit

Die Robustheit des Verfahrens wurde durch die Alignierung aller textuellen Tokens (Rückgabemenge der Methode `getTextualTokens()`) der Groundtruth

[6] In [16] wird bspw. im TREC-5 Korpus [59] die maximale Wortzahl einer Seite mit 2000 angegeben; die hohe Zahl rührt von der Verwendung sehr kleiner Fonts.

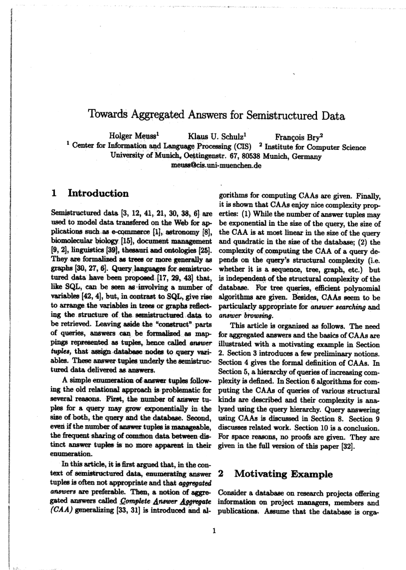
zu denen der OCR-Ausgabe auf dem vorhandenen Korpus untersucht (11 der 15 Teilkorpora). Da eine detaillierte, manuelle Überprüfung zu zeitintensiv wäre, wurden folgende Punkte geprüft:

- **Ordnungsgemäßer Programmablauf**. Das Alignierungsprogramm läuft bei allen untersuchten Dokumenten ohne Exceptions oder Errors und liefert eine alignierte Doppel-Liste.

- **Plausible Substitutionen**. Da alle Merges, Splits, Deletions und Insertions für die lexikalische Nachkorrektur ausgefiltert werden, liegt ein Hauptaugenmerk auf der korrekten Alignierung von Matches und Substitutionen. Da auch non-normale, textuelle Tokens für die lexikalische Nachkorrektur ausgefiltert werden, wurden nur normale, problematische Alignierungspaare beobachtet. Als problematisch wurden Substitutionspaare eingestuft, deren längensensitiver Levenshtein-Abstand den Schwellwert von 0.5 überschreitet, d. h. mehr als die Hälfte der Symbole der textuellen Tokens stimmen nicht überein. Die maximale, beobachtete Anzahl problematischer Alignierungspaare auf einer Seite liegt knapp über 2% (in absoluten Zahlen: 5 Paare). Nur 4 Dokumente enthalten mehr als 1% an problematischen Alignierungen. Eine Kontrolle in dieser Spitzengruppe hat gezeigt, dass es sich um stark von der OCR-Engine verunstaltete Textpassagen handelt, bei denen sich auch eine manuelle Token-Zuordnung schwierig gestaltet. Die restlichen, vereinzelten Alignierungsauffälligkeiten – in absoluten Zahlen sind es insgesamt 107 – wurden manuell durchsucht. Mehr als 80% davon sind bzgl. ihrer Gestalt als korrekte Alignierungspaare mehr oder minder zu erkennen; ihr längensensitiver Levenshtein-Abstand liegt jedoch noch über dem Schwellwert, wie z. B. (`may`, `nnay`) oder (`November`, `Nijvecjiber`). Von den verbleibenden Fehlalignierungen lassen sich noch einige durch folgenden Störeffekt bei der Erkennung erklären: Im Text steht z. B. `Bauernhofes`, die OCR-Engine erkennt aber nur den ersten Teil `Bauern`. Schließlich bleiben nur eine knappe Hand voll unerklärbarer Alignierungspaare, davon das ungewöhnlichste: (`Pflanzenreste`, `leer`).

- **Plausible Länge der Alignierungsliste**. Es wurde die Länge der Alignierungsliste mit der Länge der Groundtruth und der OCR-Ausgabe verglichen. Dabei ist zu beachten, dass auch Groundtruth und OCR-Ausgabe in ihrer Länge i. Allg. differieren. In 90% der Dokumente liegt diese Abweichung unter einem Prozent. Die maximale beobachtete Abweichung beträgt etwa 7%. In diesem Dokument hat die OCR-Engine mathematische Formeln als Bilder überlesen, die in der Groundtruth textuell repräsentiert sind. Andere Ursachen für größere Längendifferenzen sind meist von der OCR-Engine überlesene Passagen, die sehr undeutlich oder klein gedruckt sind. Für den Robustheitstest der Alignierung sind aber nur Längenabweichungen von Interesse, die über diese Differenz hinausgehen. Es wurden keine derartigen Abweichungen festgestellt.

Prinzipiell lässt sich der Algorithmus auch auf die Rückgabemenge der Methode `getQueryTokens()` anwenden, allerdings ist es vorzuziehen, zuerst alle



Towards Aggregated Answers for Semistructured Data

Holger Meuss[1] Klaus U. Schulz[1] François Bry[2]
[1] Center for Information and Language Processing (CIS) [2] Institute for Computer Science
University of Munich, Oettingenstr. 67, 80538 Munich, Germany
meuss@cis.uni-muenchen.de

1 Introduction

Semistructured data [3, 12, 41, 21, 30, 38, 6] are used to model data transfered on the Web for applications such as e-commerce [1], astronomy [8], biomolecular biology [15], document management [9, 2], linguistics [39], thesauri and ontologies [25]. They are formalized as trees or more generally as graphs [30, 27, 6]. Query languages for semistructured data have been proposed [17, 29, 43] that, like SQL, can be seen as involving a number of variables [42, 4], but, in contrast to SQL, give rise to arrange the variables in trees or graphs reflecting the structure of the semistructured data to be retrieved. Leaving aside the "construct" parts of queries, answers can be formalized as mappings represented as tuples, hence called *answer tuples*, that assign database nodes to query variables. These answer tuples underly the semistructured data delivered as answers.

A simple enumeration of answer tuples following the old relational approach is problematic for several reasons. First, the number of answer tuples for a query may grow exponentially in the size of both, the query and the database. Second, even if the number of answer tuples is manageable, the frequent sharing of common data between distinct answer tuples is no more apparent in their enumeration.

In this article, it is first argued that, in the context of semistructured data, enumerating answer tuples is often not appropriate and that *aggregated answers* are preferable. Then, a notion of aggregated answers called *Complete Answer Aggregate (CAA)* generalizing [33, 31] is introduced and algorithms for computing CAAs are given. Finally, it is shown that CAAs enjoy nice complexity properties: (1) While the number of answer tuples may be exponential in the size of the query, the size of the CAA is at most linear in the size of the query and quadratic in the size of the database; (2) the complexity of computing the CAA of a query depends on the query's structural complexity (i.e. whether it is a sequence, tree, graph, etc.) but is independent of the structural complexity of the database. For tree queries, efficient polynomial algorithms are given. Besides, CAAs seem to be particularly appropriate for *answer searching* and *answer browsing*.

This article is organized as follows. The need for aggregated answers and the basics of CAAs are illustrated with a motivating example in Section 2. Section 3 introduces a few preliminary notions. Section 4 gives the formal definition of CAAs. In Section 5, a hierarchy of queries of increasing complexity is defined. In Section 6 algorithms for computing the CAAs of queries of various structural kinds are described and their complexity is analyzed using the query hierarchy. Query answering using CAAs is discussed in Section 8. Section 9 discusses related work. Section 10 is a conclusion. For space reasons, no proofs are given. They are given in the full version of this paper [32].

2 Motivating Example

Consider a database on research projects offering information on project managers, members and publications. Assume that the database is orga-

1

Abbildung 8.6: Zweispaltiges Beispieldokument.

textuellen Tokens zu alignieren und anschließend aus der Ergebnisliste die normalen Tokens auszuwählen. Ansonsten kann es in seltenen Fällen zu Alignierungsfehlern wie diesem kommen: Im Text steht `aren't well` und die OCR-Engine erkennt `arertt w.ell`. Durch meine restriktive Definition normaler Tokens wird jeweils eines davon ausgesondert und es bleibt das Alignierungspaar (`well`, `arertt`).

8.4 Context-Match-Algorithmus

8.4.1 Grenzen

Solange die Lesereihenfolge der zu alignierenden Texte parallel läuft, eignet sich der Minimum-Edit-Distance-Algorithmus sehr gut. In OCR-Anwendungen können jedoch in mehrspaltigen Dokumenten oder Dokumenten mit komplexem Layout auch Lesereihenfolgevertauschungen auftauchen. Es wurde ein zweispaltiges Beispieldokument mit einer OCR-Engine eingelesen.

Fälschlicherweise hat die OCR-Engine das Dokument als einspaltigen Text von links oben nach rechts unten durchgelesen. Für eine Alignierung der OCR-Ausgabe zur Groundtruth lässt sich (ohne der optimierenden Einschränkung auf einen Kanal) eine Übergangsmatrix berechnen. Auffallend ist, dass die Trace-Back-Spur sehr unregelmäßig verläuft und ein enormer Gesamt-Levenshtein-Abstand errechnet wird. Bei näherer Inspektion der Alignierungspaare wird klar, dass der Minimum-Edit-Distance-Algorithmus an dieser Aufgabe komplett

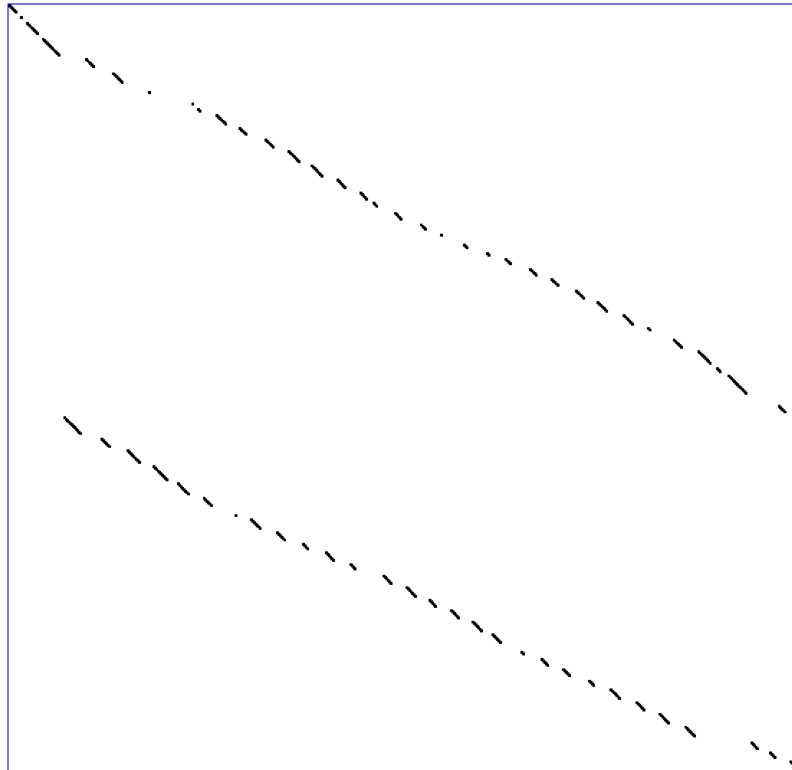


Abbildung 8.7: Korrekte Trace-Back-Spur durch zweispaltiges Beispieldokument.

scheitert. Die korrekte Trace-Back-Spur pendelt zwischen zwei Subspuren.

Der Algorithmus kann die Sprungstellen zwischen den Subspuren, die durch die Zweispaltigkeit bedingt sind, nicht verbinden, da beim Rücklauf in der Matrix immer nur fünf benachbarte Zellen in Betracht gezogen werden.
Der Context-Match-Algorithmus ist hingegen robust gegenüber solchen Sprungstellen, da er immer alle Tokens bei der Alignierungspartnersuche miteinbezieht.

8.5 Context-Match-Algorithmus

8.5.1 Algorithmusidee

Einen Kontext kann man sich als sichtbaren Ausschnitt auf einem fortlaufenden Text vorstellen. Er besteht aus einem Präkontext, gefolgt vom Zentralwort und einem Postkontext $c = [c^{pre}, w, c^{post}]$.

Die Prä- und Postkontextlänge kann durch Symbolanzahl oder durch Wort-

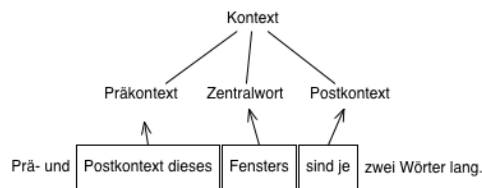


Abbildung 8.8: Aufbau eines Kontexts.

anzahl festgelegt sein, wobei die beiden Längen $|c^{pre}|, |c^{post}|$ auch differieren können.
Je ein Dokument wird zum Master- und zum Slave-Dokument gewählt. Da das Master-Dokument die resultierende Lesereihenfolge bestimmt, sollte bei der OCR-Groundtruth-Alignierung der Originaltext als Master gewählt werden und bei der Alignierung verschiedener OCR-Engines, die in der Block-Segmentierungstechnik überlegene Engine. Das Kontextfenster wird schrittweise über das Master-Dokument geschoben und aus dem Slave-Dokument wird dazu der jeweils ähnlichste Kontext gesucht.

8.5.2 Kontextabstandsmaß

Die Ähnlichkeitsbestimmung zweier Kontexte kann mit Hilfe diverser String-Abstandsmaße erfolgen. Ich habe mich für den Levenshtein-Abstand entschieden, da er auch an anderen Stellen der Arbeit eingesetzt wird und daher Implementierungen dazu vorhanden sind. Der Abstand zweier Kontexte setzt sich aus den Abständen der Präkontexte, der Zentralwörter und der Postkontexte zusammen:

$$d(c_1, c_2) = \omega_1 \cdot lev_{length}(c_1^{pre}, c_2^{pre}) + \omega_2 \cdot lev_{length}(w_1, w_2) + \omega_3 \cdot lev_{length}(c_1^{post}, c_2^{post})$$

Die drei Komponenten sind mit einer Gewichtung $\omega_i \in [0, 1]$ versehen. Sind Prä- bzw. Postkontexte auf Worteben definiert und deren Länge $\geqslant 2$ Wörter, wird die String-Konkatenation der beiden Prä- bzw. Postkontexte gebildet und deren Gesamtabstand berechnet. Der Abstand wird nicht paarweise zwischen den einzelnen Wörtern berechnet, da ansonsten Splits und Merges auf Wortebene zu große Abstände bewirken würden. Da verschiedene Kontextabstände auch untereinander vergleichbar bleiben sollen, ist der längensensitiven Levenshtein-Abstand dem klassischen überlegen. Die Vergleichbarkeit verschiedener Kontextabstände wird z. B. zur Bestimmung einer oberen Schranke für die Alignierung benötigt. Bedingt durch Wortlöschungen der OCR-Engine, o. ä. hat nicht jeder Kontext des Master-Dokuments eine Entsprechung im Slave-Dokument. Fehlalignierungen, die aus dieser Situation entstehen können, werden durch eine obere Schranke verhindert.

8.5.3 Naive Implementierung

Um die prinzipielle Machbarkeit zu prüfen, wurde der Algorithmus ad hoc in Java mit folgenden Parameterbelegungen implementiert:

- Die Kontextlänge wurde auf Wortebene gebildet. Es wurde $|c^{pre}| = |c^{post}| = 3$ gewählt.
- Die Gewichtung wurde mit $\omega_1 = \omega_2 = \omega_3 = 1$ gewählt.
- Zur Beschleunigung wurde ein einfacher Grobfilter eingebaut. Wenn die Zentralwörter schon zu stark differieren, wird die Kontextähnlichkeit nicht weiter geprüft. Ein Slave-Kontext c_{slave} wird nicht weiter betrachtet, falls $lev_{length}(w_{master}, w_{slave}) > 2/3$. Dieser Grobfilter ist gleichzeitig der einzige Test, ob überhaupt eine Zuordnung stattfindet. Allerdings können damit nur Löschungen von Wörtern erkannt werden, die nur einmal im Text auftreten und auch keinen zu ähnlichen Verwechslungspartner im Text besitzen.

Für die Dokumentenränder wurde eine Spezialbehandlung eingebaut, da die Kontexte am Anfang bzw. Ende des Dokuments keinen Prä- bzw. Postkontext haben. Ohne diese Vorkehrung treten Fehlalignierungen an diesen Stellen auf. Die Implementierung wurde wie in 8.3.3 mit dem längsten Dokument aus dem Korpus auf meinem Arbeitsplatzrechner getestet. Ohne Grobfilter benötigt die Alignierung etwas mehr als eine Minute, mit Grobfilter knapp unter 9 Sekunden. Der Ergebnisvergleich mit der Minimum-Edit-Distance-Alignierung hat vier Abweichungen aufgezeigt. Der naive kontextbasierte Algorithmus hat Probleme mit der korrekten Alignierung von Insertions, Merges und Splits.

8.5.4 Verbesserungspotential

Die Alignierungsprobleme, die OCR-Fehler auf Wortebene betreffen, können in der Regel durch partnerlose Kontexte erkannt werden. Master-Kontexte ohne Alignierungspartner ermittelt der Algorithmus ohne weitere Vorkehrungen. Markiert man bei jeder Zuordnung den Slave-Kontext, kann man am Ende unmarkierte und damit partnerlose Slave-Kontexte bestimmen. Die verschiedenen OCR-Fehlertypen auf Wortebene im Einzelnen:

- **Split**. Betroffen sind ein Master-Kontext und zwei Slave-Kontexte. Das Resultat des naiven Algorithmus kann vielfältig ausfallen, ist aber immer falsch: Zuordnung einer der beiden Slave-Kontexte zum Master-Kontext, keine Zuordnung oder gar Fehlzuordnung. Im ersten Fall ist eine Heilung durch eine Analyse des Prä- und Postkontexts im Slave-Kontext möglich. Es wird geprüft, ob die Zuordnung des Slave-Zentralworts konkateniert mit dem letzten Wort des Prä- bzw. mit dem ersten Wort des Postkontexts ähnlicher zum Master-Zentralwort ist als das Slave-Zentralwort alleine. Alternativ kann der gesamte Algorithmus um eine Phase erweitert werden, in der Slave-Kontexte mit zwei Zentralwörtern zu den Master-Kontexten

aligniert werden. Wird sowohl in der einfachen als auch in der erweiterten Phase zu einem Master-Kontext jeweils ein Slave-Kontext zugeordnet, ist eine Konfliktauflösung erforderlich.

- **Merge**. Betroffen sind zwei Master-Kontexte und ein Slave-Kontext. Der naive Algorithmus liefert wie im Split-Fall diverse falsche Alignierungen: Zuordnung des Slave-Kontexts zu einem der beiden Master-Kontexte wobei der zweite Master-Kontext fehl- oder leeraligniert ist, zwei Leeralignierungen, zwei Fehlalignierungen oder eine Leeralignierung und eine Fehlalignierung. Analog zum Split-Fall ist im ersten Fall eine Heilung durch eine Analyse des Prä- und Postkontextes des Master-Kontexts möglich. Zusätzlich muss noch der zweite Master-Kontext aus der Ergebnisliste entfernt werden. Alternativ kann ebenfalls analog zum Split-Fall eine weitere Phase eingeführt werden, in der Master-Kontexte mit zwei Zentralwörtern einfachen Slave-Kontexten zugeordnet werden. Auch für diese zusätzliche Phase ist i. Allg. eine Konfliktauflösung für Mehrfachzuordnungen erforderlich.

- **Deletion**. Betroffen ist ein Master-Kontext. Der naive Algorithmus behandelt diesen Fehler-Typ in der Regel richtig. Fehlalignierungen sind durch die Einstellung der Alignierungsschranke zu vermeiden.

- **Insertion**. Betroffen ist ein Slave-Kontext. Insertions lassen sich leicht auffinden, wenn man den naiven Algorithmus um eine Zuordnungsmarkierung erweitert. Am Ende bleibt nur noch das Problem der Positionierung der unmarkierten Slave-Kontexte. Dazu eignet sich am besten ein speziell adaptierter Context-Match-Algorithmus, der nur Prä- und Postkontext ohne Zentralwort aus dem Master-Dokument mit den verbleibenden Slave-Kontexten vergleicht.

Einfache Experimente auf OCR-Dokumenten mit Lesereihenfolgefehlern haben gezeigt, dass die ad hoc Implementierung des Context-Match-Algorithmus im Gegensatz zum Minimum-Edit-Distance-Algorithmus auch diese Art von Dokumenten einigermaßen alignieren kann (vgl. [44]). Allerdings ist die Zuordnung direkt vor und hinter jeder Sprungstelle der Lesereihenfolge nicht immer erfolgreich. Hintergrund ist, dass direkt vor der Sprungstelle nur die Präkontexte übereinstimmen und direkt nach der Sprungstelle nur die Postkontexte. Eine Verbesserungsidee ist der Einsatz halber Kontexte. Mit folgender Auswahl zwischen Prä- und Postkontext bei der Kontext-Ähnlichkeitsbestimmung werden deutlich bessere Ergebnisse erzielt:

$$d(c_1, c_2) = \omega_1 \cdot max(lev_{length}(c_1^{pre}, c_2^{pre}), lev_{length}(c_1^{post}, c_2^{post})) + \omega_2 \cdot lev_{length}(w_1, w_2)$$

Folgende Beobachtung motiviert eine Optimierungsidee: Aus einer Kontextzuordnung resultiert in den meisten Fällen, dass auch die beiden jeweils folgenden Kontexte zueinander korrespondieren. Dies gilt auch für Kontexte aus Dokumenten mit Lesereihenfolgevertauschungen. In den meisten Fällen kann daher eine Suche auf allen Slave-Kontexten vermieden werden. Erst wird der

Folgekontext als Alignierungspartner geprüft und nur wenn der Abstand zum Master-Kontext einen festgelegten Grenzwert übersteigt, wird die Suche auf das gesamte Slave-Dokument ausgeweitet.

8.5.5 Vergleich und Kombination mit dem Minimum-Edit-Distance-Algorithmus

Vereinfachend lasse ich bei den Komplexitätsbetrachtungen die Optimierungen außer Betracht. Zusätzlich wird vereinfachend angenommen, dass die Ähnlichkeitsberechnung zweier Kontexte mit konstantem Aufwand erfolgt. Durch den Vergleich jedes Kontexts des Master-Dokuments mit jedem Kontext des Slave-Dokuments resultiert dieselbe quadratische Zeitkomplexität $O(nn')$ wie beim Minimum-Edit-Distance-Algorithmus. Die Speicherplatzkomplexität ist hingegen linear, da im Speicher immer nur ein aktuell zu alignierender Master-Kontext und dazu alle Slave-Kontexte mit Ähnlichkeitswerten vorhanden sein müssen.
In meiner Arbeit wird der Minimum-Edit-Distance-Algorithmus für die Alignierungsaufgaben eingesetzt, da er für Splits, Merges, Deletions und Insertions eine saubere konzeptionelle Lösung ohne Fehlalignierungen bietet. Der Context-Match-Algorithmus hingegen muss für diese relativ häufigen OCR-Fehlertypen mit Heuristiken angereichert werden. Die Vorteile des Context-Match-Algorithmus gegenüber dem Minimum-Edit-Distance-Algorithmus spielen für meine Aufgabenstellung eine untergeordnete Rolle:

- **Robustheit gegenüber Lesereihenfolgefehler**. Mein Korpus enthält keine Dokumente, für die Lesereihenfolgefehler zu erwarten sind.
- **Weniger Hauptspeicherplatzbedarf**. In 8.3.3 wurde gezeigt, dass der Hauptspeicherbedarf für die Aufgabenstellung unkritisch ist.
- **Skalierbarkeit auf mehrere Dokumente**. Die Master-Slave-Anordnung ermöglicht eine einfache Erweiterung zur Alignierung von drei oder mehr Dokumenten, indem mehrere Slave-Dokumente eingesetzt werden. Einem Master-Kontext wird aus jedem Slave-Dokument ein Kontext zugeordnet. Bei der Groundtruth-Zuordnung werden nur zwei Texte aligniert. Die Kombination von OCR-Engines wurde nur mit zwei Engines getestet.

Für weitergehende Anwendungen ist eine Kombination der beiden Algorithmen sinnvoll, die die Vorteile der beiden vereint. Zwei Ansätze dazu:

- Als Basis wird der Minimum-Edit-Distance-Algorithmus mit seiner Übergangsmatrix verwendet. Wird beim Trace-Back ein vorgegebener Konfidenzwert für den Alignierungsabstand überschritten, wird angenommen, dass eine Sprungstelle in der Matrix vorliegt, die mit Hilfe eines Maßes bzgl. der Kontextabstände gesucht wird.
- Vorbereitend wird der Context-Match-Algorithmus verwendet und alle sicheren Kontextzuordnungen im Dokument gesucht. Zwischen diesen An-

kerpunkten erfolgt dann jeweils eine Alignierung mit Hilfe des Minimum-Edit-Distance-Algorithmus.

Kapitel 9

Automatisierte Anfragen an Suchmaschinen

9.1 Motivation und Aufgabenstellung

Das Web als riesiges, permanent aktualisiertes Textkorpus für die OCR-Nachkorrektur nutzbar zu machen, ist eine Hauptintension meiner Arbeit. Der klassische Weg, um dieses Korpus lokal verfügbar zu machen, ist das systematische Einsammeln von Web-Seiten. Mehr zum sog. *Crawling* findet man in 4.3. Moderne Verfahren dazu sind z. B. in [3], [57] oder [94] beschrieben. Trotz der einfachen Idee des Basisalgorithmus - rekursive Verfolgung aller Links - ist ein enormer Programmieraufwand für eine Reihe von Detailproblemen wie Dublettenerkennung, Lastbalancierung, robustes HTML-Parsing, etc. notwendig. Ausserdem benötigt der Aufbau einer (Teil-)Spiegelung des Webs riesige Ressourcen an Bandbreite und Speicher. Einen viel einfacheren Zugang zu mehreren Milliarden Web-Seiten erhält man via Suchmaschinen. Im diesem Kapitel beschreibe ich zwei Alternativen, um Anfragen an Suchmaschinen zu automatisieren. Es wird ein Anfrage-String an die Maschine geschickt und aus der Rückgabe die gewünschte Information extrahiert; alles weitere an Aufwand ist an den Dienstleister Suchmaschine weiterdelegiert. Mit einem Webservice (Anwendungsbeispiel Google) und einem Wrapper (Anwendungsbeispiel AllTheWeb) habe ich diese Funktionalität realisiert. Solche automatisierten Anfragen an Suchmaschinen verwende ich in meiner Arbeit an verschiedenen Stellen:

- Erzeugung dynamischer Lexika zur Nachkorrektur, d. h. themenspezifische Wortlisten mit Frequenzinformation (siehe 9). Mit der automatisierten Anfrage ermittle ich die URLs relevanter Dokumente, aus denen die Lexika gewonnen werden.

- Aufspüren von Kopien der Groundtruth-Dokumente (siehe 9). Durch automatisierte Anfragen längerer Phrasen (sog. Fingerabdrücke) ermittle ich

die URLs von Dokumenten, die für den Aufbau dynamischer Lexika ausgeschlossen werden müssen.

- Berechnung von Scores, die widerspiegeln wie gut sich Korrekturkandidaten in den Kontext einpassen (siehe 5.4). Dazu werden Wort-n-Gramme an eine Suchmaschine angefragt und damit deren Häufigkeit im indexierten Web ermittelt.

- Ich untersuche statistisch den Aufbau von Wörtern mit Hilfe ihrer Häufigkeit im Web (siehe 3.2.1).

Meine Anwendungen erfordern zwei unterschiedliche Informationstypen aus Suchanfragen: die rückgelieferte URL-Liste und deren Länge.

9.2 Juristische Aspekte

Beide Ansätze, die ich vorstelle, sind nur mit starken rechtlichen Einschränkungen anwendbar. In den Nutzungsbedingungen des Webservices zur Anfrage an Google ([23]) ist eine kommerzielle Nutzung ausgeschlossen. Automatisierte Anfragen an die Suchmaschine AllTheWeb sind gemäß ihrer Nutzungsbedingungen immer genehmigungspflichtig. Dank einer Kooperation des Instituts mit Overture, der Betreiberfirma von AllTheWeb, erfolgte die Zusage, dass ich für meinen akademischen Prototyp lastbalancierte automatisierte Anfragen stellen darf. Dazu habe ich mittels Reverse-Engineering der HTML-Ausgabe einen sog. Wrapper geschrieben, der mir die gewünschte Information extrahiert. Die Beschreibung des Wrapper-Ansatzes ist nicht als Anleitung zum Datendiebstahl zu verstehen, sondern stellt einen De-facto-Standard für den Datenaustausch vor (vgl. [76]). Der Wrapper-Ansatz ist einer mundgerechten Aufbereitung auf Seite der Datenquelle technisch unterlegen. Dieser Ansatz ist jedoch notwendig, wenn dem Datenlieferanten im Gegensatz zum Datenempfänger Kapazitäten für den Datentransfer fehlen, d. h. Mangel an Zeit, Personal, Know-how, Geld, o. ä.

9.3 Webservice

In [84] wird ein Webservice folgendermaßen definiert: ein Software-System, identifiziert durch eine URI, dessen öffentliche Schnittstellen und Anbindungen mit Hilfe von XML definiert und dokumentiert sind. Andere Software-Systeme haben die Möglichkeit, diese Definition aufzufinden und an Hand ihr, in Interaktion mit dem Webservice zu treten. Dazu werden XML-basierte Nachrichten mit Hilfe von Internet-Protokollen versandt.
Der von Google ([23]) seit Frühjahr 2002 angebotene Webservice stützt sich auf die weitverbreiteten Standards WSDL (Definition und Dokumentation), SOAP (XML-Nachrichtenformat) und HTTP (Transport). Zusätzlich sind für die Programmiersprachen Java, VisualBasic und C# fertige APIs vorhanden. Dadurch ließ sich der Webservice direkt in meinen Java-Prototypen einbauen.

Einem Objekt der Klasse `GoogleSearch` werden mit Setter-Methoden Berechtigungsschlüssel sowie eine Anfrage übergeben. Der Anfrage-String gehorcht dabei der gleichen Syntax, die auch für direkte Eingaben auf der Web-Seite der Suchmaschine verwendet wird. Anschließend wird mit der Methode `doSearch()` wird ein Ergebnis-Objekt vom Typ `GoogleSearchResult` erzeugt. Daraus kann man direkt die (teilweise geschätzte) Trefferzahl auslesen und auch eine Iteration durch das Array der einzelnen Resultate anstoßen. Von denen lassen sich wiederum die URLs auslesen. Pro Rückgabeobjekt sind maximal 10 URLs enthalten. Mit der Methode `setStartResult()` kann man jedoch dem Anfrageobjekt mitteilen, welchen Ausschnitt der Ergebnisliste man zu Gesicht bekommen möchte. Ich habe eine durchschnittliche Antwortzeit von 0.83 sec. gemessen.
Der Ansatz ist derzeit nicht auf andere Suchmaschinen ausdehnbar, da der Webservice von Google der einzige dieser Art ist, den ich gefunden habe. Neben dem Ausschluss kommerzieller Anwendungen weist dieser Ansatz weitere Beschränkungen auf: maximal 1000 Anfragen innerhalb eines Tages und fehlende Planungssicherheit, d. h. es wird explizit darauf hingewiesen, dass der Dienst sich im Beta-Status befindet und jederzeit, ohne Vorankündigung abgeändert oder sogar eingestellt werden kann.

9.4 Wrapper

Im Web gibt es eine Vielzahl an Sites, die unterschiedliche Informationen in immer das gleiche Layout-Template verpacken. Das sind zum einen regelmäßig aufgefrischte Seiten wie Wetterberichte oder Börsenkurse und zum anderen dynamisch zu einer Anfrage generierte Webseiten, z. B. Rückgabemengen von E-Commerce-Angeboten oder auch Suchmaschinen. Möchte man zu einer solchen Datenquellen eine eigene Schnittstelle schaffen, muss man die Informationen aus dem prozeduralen Layout-Gewand auspacken und in sein eigenes, deskriptives Strukturgewand einpacken. Im Fachjargon heißen Programme, die für dieses Umpacken zuständig sind Wrapper. Mit dieser Wrapper-Technik lassen sich auch verschiedene Datenquellen vereinen. Systeme, die solche Transformationen zur Laufzeit ausführen, nennt man Mediatorensysteme. Neben diesen virtuell integrierten Systemen gibt es auch Systeme, die mehrere Quellen so vereinen, dass alle Anfragen innerhalb einer materialisierten Datenbank verarbeitet werden können. Diese Systeme werden universelle Datenbankmanagementsysteme genannt. Es gibt eine Reihe von Werkzeugen, die eine Wrapper-Entwicklung erleichtern, d. h. spezialisierte Hochsprachen, selbstlernende Algorithmen, Repositories und Entwicklungsumgebungen für Wrapper. In [76] gebe ich einen kleinen Überblick mit weiterführenden Verweisen.
Ein Test mit dem HTML-Validator des W3C (`http://validator.w3.org/`) hat gezeigt, dass die zu verarbeitende HTML-Seite eine Reihe kleiner Syntax-Fehler enthält. Daher ist zu erwarten, dass die HTML-Parser einiger Wrapper-Tools nicht robust genug für diese Aufgabe sind. WebL[1] [41] ist über längere Zeit

[1] Compaq hat WebL vorsorglich in Compaq's Web Language umgetauft, um einer Namensrechtstreitigkeit vorzubeugen; für bessere Lesbarkeit verwende ich jedoch weiter den ursprüng-

unter industrieller Obhut (Compaq) entstanden und daher eines der ausgereifteren und stabileren Wrapper-Tools. Mit einem WebL-Skript ließen sich die Ergebnis-URLs aus der Rückgabeseite der Suchmaschine erfolgreich extrahieren. Java-Code kann problemlos in ein WebL-Skript eingebunden werden, da der WebL-Interpreter selbst in Java implementiert ist. Die umgekehrte Einbettung erfordert hingegen einen unverhältnismäßig großen Aufwand, so dass eine unmittelbare Einbindung in meinen Java-Prototypen verhindert wird.
Da für das Auslesen der Ergebnisseite kein komplettes Parsing des DOM-Baumes notwendig ist, sondern einfaches Pattern-Matching ausreicht, habe ich den Wrapper in Java recodiert. Ein weiteres Mal habe ich die gleiche Idee auf UNIX-Skriptebene mit zwei Befehlsaufrufen realisiert: das Regionen-Algebra-Werkzeug sgrep [34] für die Informationsextraktion und wget [21] für die Anfrage. Die Anfrage an die Suchmaschine erfolgte in allen vorgestellten Wrapper-Programmen nach dem gleichen Schema. Es wird ein String zusammengebaut, der die Anfrage enthält und an die URL angehängt. Im Bereich der Suchmaschinen ist die Verwendung von HTML-Forms zusammen mit der get-Methode eine Standardtechnik. Die Syntax einer Anfrage lässt sich leicht aus Beispielen erschließen. Technisch lässt sich daher der Wrapper-Ansatz auf nahezu jede Suchmaschine ausdehnen. Mit der Anfragesyntax der Suchmaschine lassen sich noch eine Reihe von Feineinstellungen vornehmen wie etwa Bedingungen an die Seitengröße zu knüpfen oder die Auswahl einer Sprache, eines Dokumenttyps oder einer geographischen Region. Die Rückgabe einer Suchanfrage an AllTheWeb enthält direkt eine Anzahl der Gesamttreffer und eine Liste von bis zu 100 Treffer-URLs. Ist die Rückgabemenge größer als 100 URLs, lässt sich mit einem Parameter in der Anfrage steuern, welchen Ausschnitt der Ergebnisliste (à 100 URLs) man geliefert haben will.
Ich habe eine durchschnittliche Antwortzeit von 0.85 sec. gemessen (vgl. dazu auch die Ergebnisse aus [61] S.59).
Ein schwerwiegender Nachteil des Wrapper-Ansatzes sind die Abhängigkeiten von Layout und diversen technischen Details. Im Entstehungszeitraum dieser Arbeit mussten die Wrapper mehrfach angepasst werden, da sich das Erscheinungsbild der Web-Seite und das Session-Management für gezielte Werbeeinblendungen, das in die URLs der Trefferliste eingebettet wurde, geändert haben.

lichen Namen.

Kapitel 10

Evaluation

10.1 Ablauf

Die Evaluation ist in fünf Experimente gegliedert. Zu Beginn wird die Eignung der Crawl-Lexika zur OCR-Nachkorrektur untersucht, indem maximal erreichbare Korrekturresultate auf Trainingsmaterial für verschiedene statische und dynamische Lexikonkombinationen gegenübergestellt werden. Im zweiten Experiment wird mit einer Trainings- und Testphase die Verbesserung der automatischen Nachkorrektur einzelner OCR-Engines durch eine Optimierung der Einflussparameter überprüft. Dieses Experiment wird mit zwei unterschiedlichen OCR-Engines durchgeführt, und im Anschlussexperiment wird das Potential einer scorebasierte Kombination der beiden Engines ausgeleuchtet. Dieses ebenfalls in Trainings- und Testphase organisierte Experiment bündelt die vermeintlich besten automatischen Nachkorrekturtechniken meiner Arbeit. Im vierten Experiment wird auf Trainingsmaterial untersucht, ob sich der für die scorebasierte Kombination nötige Aufwand zur Alignierung der OCR-Resultate wirklich lohnt, oder ob eine einfachere, lexikonbasierte Kombination vorzuziehen ist. Mit einer prototypischen Adaption des UNIX-Tools ispell zeige ich im letzten Experiment, dass meine parameteroptimierten, OCR-kombinierten Ergebnis-Files auch für eine interaktive Nachkorrektur geeignet sind.

10.2 Eignung der Crawl-Lexika

10.2.1 Experiment

Die Eignung der Crawl-Lexika für die lexikalische Nachkorrektur wurde mit einem Experiment auf den Teilkorpora der Themengebiete Mykologie, Fische, Holocaust, Rom, Speisepilze und Neurologie sowohl in Englisch als auch in Deutsch überprüft. Dieses Experiment wurde bereits auf der ICDAR2003 in [74] vorgestellt. Es wurden fünf verschiedene Lexika bzw. Lexikonkombinationen pro Sprache getestet:

- Ein kleines, statisches, sprachspezifisches Lexikon mit den 10^5 häufigsten Wörtern ($D^E_\downarrow$ bzw. $D^G_\downarrow$).
- Eine sprachspezifische Kombination aller statischen Lexika (D^{E+} bzw. D^{G+}); siehe 4.2 für die Größenangaben.
- Ein sprach- und themenspezifisches Crawl-Lexikon (D^{EC} bzw. D^{GC}); siehe 4.3.3 für die Größenangaben.
- Eine sprachspezifisch Kombination aller statischen Lexika zusammen mit dem jeweiligen Crawl-Lexikon (D^{EC}_{E+} bzw. D^{GC}_{G+}).
- Das perfekte Lexikon je Teilkorpus und Sprache (D^{perf}) als obere Schranke.

Für alle (Teil-)Lexika außer dem perfekten und dem Abkürzungslexikon wurde das Flexions- und Orthographievarianten-Tool hinzugenommen. Für alle fünf Lexikonkombinationen gepaart mit allen fünf Teilkorpora in beiden Sprachen wurde jeweils die optimale Korrekturgrenze ermittelt. Für die Korrekturentscheidung wurden der längensensitive Levenshtein-Abstand und die Frequenz eingesetzt. Die Kombination erfolgte in einfachster Weise. Primär wurde der Levenshtein-Abstand miteinander verglichen und nur im Zweifelsfall auch die Frequenz herangezogen. Ausserdem wurden die Abdeckungen der verschiedenen Lexikonkombinationen miteinander verglichen. Für diesen Abdeckungsvergleich bedarf es keines OCR-Laufs. Die Abdeckung wird mit dem Anteil normaler, lexikalischer Tokens der Groundtruth im Verhältnis zu allen normalen Tokens der Groundtruth angegeben. Mehrfach auftretende Tokens sind dabei in diesem Quotienten auch mehrfach enthalten.
Die Güte bzgl. der automatischen Nachkorrektur wurde direkt mit der optimal erreichbaren Korrekturgenauigkeit gemessen.
Für eine Gütemessung bzgl. interaktiver Korrektur wurde ein Maß eingeführt, mit dem sich der manuelle Inspektionsaufwand aller im Lexikon fehlenden Tokens abschätzen lässt. Diese sog. Inspektionsrate wird als der Anteil nicht lexikalischer, normaler Tokens unter allen normalen Tokens definiert. Es ist zu bedenken, dass die Gütemessung mit der Inspektionsrate die Fehlerklasse der falschen Freunde nicht berücksichtigt. Die Definition normaler Tokens bezieht sich bei der Inspektionsrate auf Tokens der OCR-Erkennung; diese Definition erklärt, dass die Inspektionsrate nicht direkt aus der Abdeckung berechnet werden kann, da die Abdeckung unabhängig von OCR-Läufen definiert ist.
Zusätzlich wurden die Fehlerklassen in absoluten Zahlen gegenübergestellt, die sich bei der Lexikonauswahl antagonistisch verhalten, falsche Freunde (Klasse 1) und unvermeidbare Fehler (Klasse 6 und 7).

10.2.2 Resultate

Abdeckung

Die Abdeckung der kleinen Lexika $D^E_\downarrow$ und $D^G_\downarrow$ ist in den meisten Fällen sehr schwach. Im Deutschen fällt der Abstand zu den großen Lexika auf Grund der

Kompositabildungen erwartungsgemäß noch deutlicher aus, als im Englischen. Bemerkenswert ist, dass die Abdeckung der Crawl-Lexika D^{EC} und D^{GC} in allen Fällen die Abdeckung der maximalen, statischen Lexika D^{E+} und D^{G+} übertrifft. Die besten Abdeckungsergebnisse im Experiment werden selbstverständlich durch Einsatz der maximalen Lexikonkombinationen D^{GC}_{G+} und D^{GC}_{G+} erreicht. Im Englischen liegt diese Abdeckung bei allen Teilkorpora bis auf Mykologie, das ausgeprägt viele lateinische Fachtermini enthält, über 99%. Diese hohe Abdeckung wird nahezu auch alleine von den Crawl-Lexika erreicht. Im Deutschen steigt die Abdeckung durch Hinzunahme der statischen Lexika in allen Fällen ca. um einen Prozentpunkt, d. h. im Übergang von D^{GC} zu D^{GC}_{G+}. Um dem Leser und mir einen Eindruck von den fehlenden Wörtern zu verschaffen, habe ich zwei Mini-Stichprobe innerhalb der deutschen Teilkorpora gezogen: s_1 mit 10 Wörter, die in D^{GC}_{G+}, nicht aber in D^{GC} enthalten sind und s_2 mit 10 Wörter, die nicht im maximalen Lexikon D^{GC}_{G+} enthalten sind. Zu jedem Begriff habe ich eine automatisierte Google-Suchanfrage gestellt und damit die Anzahl der indexierten Web-Dokumente ermittelt, die diesen Begriff enthalten.

s_1 = [(`bezahntem`, 39), (`hogräfe`, 43), (`insekten`, 521000), (`landkammeri`, 28), (`nierenförmiger`, 620), (`polibios`, 47), (`reservepolizeibataillon`, 36), (`schieflagen`, 6070), (`soziodemographisch`, 490), (`überlang`, 2320)]
s_2 = [(`bittmann`, 10800), (`chimeras`, 117000), (`coleoptile`, 8670), (`knynoskephalai`, 1), (`mühlkoppe`, 741), (`olivgelbe`, 11), (`opekarova`, 199), (`staatssekretärsbesprechung`, 94), (`zahnleisten`, 642), (`zandernester`, 18)]

Die Stichproben zeigen deutlich, dass mit einem Ausbau der Crawl-Techniken die Abdeckung noch gesteigert werden kann, da alle Wörter im Web vorhanden sind. Da die beiden Stichproben weder hinsichtlich enthaltener Worttypen noch Webfrequenzen zu unterscheiden sind, ist zu vermuten, dass bei einem Ausbau der Crawl-Lexika – ähnlich wie derzeit schon im Englischen – eine Abdeckungsannäherung zwischen D^{GC} und D^{GC}_{G+} erreicht wird.

Nachkorrektur

Die besten automatischen Nachkorrekturergebnisse werden teils mit den Crawl-Lexika D^{EC} bzw. D^{GC}, teils mit den maximalen Lexikonkombinationen D^{EC}_{E+} bzw. D^{GC}_{G+} erzielt, wobei die beiden Ergebnisse immer sehr eng beieinander liegen. Da die reine OCR-Erkennung schon ein hohes Niveau vorgibt, liegen allgemein alle Verbesserungen innerhalb eines relativ eng abgesteckten Rahmens. Daher definiere ich die Differenz zwischen der puren OCR-Erkennungsrate und der Korrekturgenauigkeit mit dem perfekten Lexikon D^{perf} als die maximal erreichbare Verbesserung. Rechnet man jetzt jede erreichte Verbesserung in den Anteil der möglichen Verbesserung um, erhält man anschaulichere Werte. Diese realen Verbesserungen betragen z. B. auf den englischen Teilkorpora mit der maximalen Lexikonkombination D^{EC}_{E+} 25%, 22%, 16%, 78%, 60% und 58%; auf den deutschen Teilkorpora D^{GC}_{G+} 13%, 2%, 50%, 3%, 4% und 18%. Mit einem Balkendiagramm habe ich den Korrekturspielraum weiter veranschaulicht.

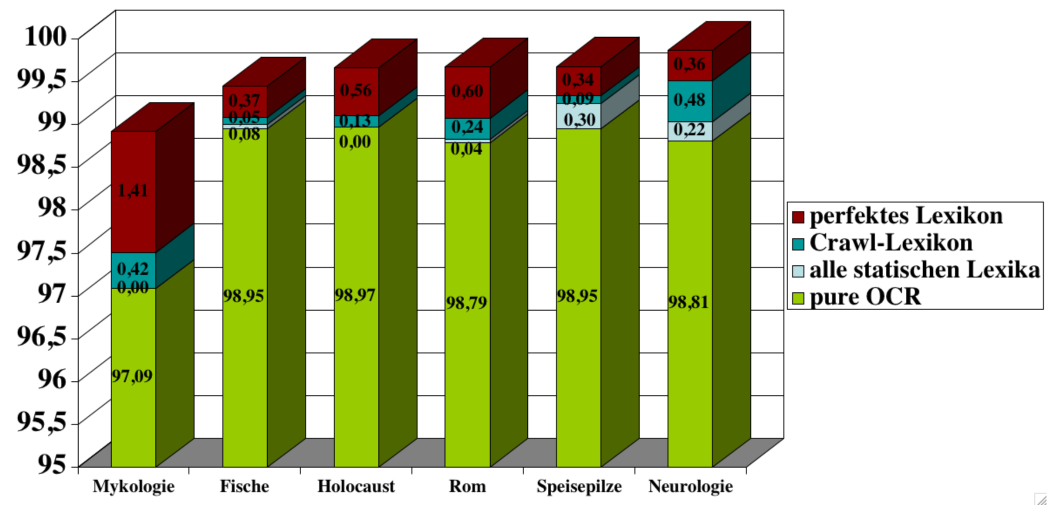


Abbildung 10.1: Korrekturspielraum der englischen Teilkorpora.

Bei der Hälfte der deutschen Teilkorpora liegt das Nachkorrekturergebnis kaum oberhalb der Ausgangsgenauigkeit der OCR-Engine. Betrachtet man dazu parallel deren Abdeckung, sieht man, dass sie auch dort mit weniger als 95% das Schlusslicht bilden. In genau der Hälfte der 48 Teilexperimente liegt die Abdeckung unter 95%. Innerhalb dieser schlechteren Hälfte liegen alle 5 Teilexperimente, bei denen überhaupt keine Verbesserung erreicht wurde; und die maximale, reale Verbesserung der schlechter abgedeckten Teilkorpora liegt bei 18% (Neurologie/deutsch mit D^{GC}).
Da ich bei Skalierung der Crawl-Menge eine bessere Abdeckung für das Deutsche erwarte, erwarte ich dadurch auch eine direkte Anhebung der Nachkorrekturqualität.
Die Anzahl falscher Freunde (respektive unvermeidbarer Fehler) steigt (fällt) erwartungsgemäß mit der Größe des Lexikons. Vergleicht man die falschen Freunde der Crawl-Lexika mit den falschen Freunden der maximalen Lexikonkombination, fällt auf, dass immer mehr als drei Viertel davon schon durch die Crawl-Lexika erklärt werden können. Im Vergleich der Crawl-Lexika mit den maximalen, statischen Lexikonkombinationen ist in fast allen Fällen die Anzahl falscher Freunde bei den Crawl-Lexika höher, obwohl sie allesamt kleiner sind, als die statischen Lexikonkombinationen. Drei Ursachen des Magnet-Phänomens falscher Freunde bei Crawl-Lexika konnte ich an Hand der Fehlerlisten ausmachen, wobei selbst die Zuordnung der genannten Beispiele nicht unbedingt eindeutig ist:

- **Akronyme**, z. B. (`vpn`, `von`) oder (`fir`, `für`).
- **Umlautproblematik**, z. B. (`staatsburger`, `staatsbürger`) oder (`fur`, `für`).

- **Fehlerhafte Tokens**, z. B. (`undt`, `und`) oder (`hclocaust`, `holocaust`).

Im Bereich kurzer Tokens ist die Gefahr falscher Freunde ohnehin recht groß, da sie bzgl. des Levenshtein-Abstands dicht geclustert liegen. Akronyme steigern diese Verdichtung zusätzlich und im Web tauchen überproportional viele technische Akronyme auf (vgl. dazu die Verteilungsfunktionen am Ende des Abschnitts 3.2.1). Analog zur Definition normaler Tokens stellt sich die Frage, ob eine Nachkorrektur besser auf längere Tokens beschränkt werden soll, wenn keine weiteren Desambiguierungsmöglichkeiten wie z. B. weitere OCR-Engines oder Kontextinformationen zur Verfügung stehen.
Für die anderen beiden Ursachen ist die geeignete Vermeidungsstrategie, qualitativ hochwertigere Seiten aus dem Web zu holen. Im Abschnitt 4.3.4 ist die Umlautproblematik und die Problematik anderer fehlerbehafteter Seiten inklusive Lösungsskizzen dargestellt.
Der Einsatz großer Lexika führt zu einer signifikanten Reduktion der Inspektionsrate. Die besten Ergebnisse werden durchwegs mit den maximalen Kombinationslexika D_{E+}^{EC} und D_{G+}^{GC} erreicht. Bedingt durch die höhere Abdeckung, ist im Englischen ist der Aufwand einer interaktiven Nachkorrektur geringer als im Deutschen. Im Englischen liegt deshalb auch die Inspektionsrate der Crawl-Lexika viel dichter bei der Inspektionsrate der maximalen Kombinationslexika, als im Deutschen.
Im Englischen fällt auf, dass mit kleinen Lexika $D_{\downarrow}^{E}$ gegenüber maximalen, statischen Lexikonkombinationen D^{E+} mindestens gleich gute, z. T. auch etwas bessere Nachkorrekturergebnisse erzielt werden. Das mag erklären, warum selbst heute noch manche Forscher den Einsatz kleiner Lexika für Nachkorrekturaufgaben anraten. So wurde bspw. im Anschluss an Sabine Carbonnels Vortrag über eine lexikalische Nachkorrektur handschriftlicher Dokumente auf der ICDAR2003 ([4]) der Einwand geäußert, dass das verwendete Lexikon mit 25 000 (!) Einträgen deutlich zu groß sei; woraufhin eine kontroverse Diskussion folgte.

10.3 Automatische Korrektur

10.3.1 Einsatz einzelner OCR-Engines

Experiment

Die Qualität der automatischen Nachkorrektur eines OCR-Ergebnisses wurde mit einem Experiment auf den Teilkorpora der Themengebiete Holocaust, Rom, Speisepilze und Neurologie sowohl in Englisch als auch in Deutsch überprüft. Da mein Evaluationsexperiment für die Kombination von OCR-Engines auf diesem Experiment aufbaut, wurde schon das Einzel-OCR-Experiment mit je zwei verschiedenen Engines durchgeführt und damit auf eine breitere empirische Basis gestellt.
Alle Teilkorpora wurden im Verhältnis 3 zu 7 in Trainings- und Testdaten aufgespaltet. Als Korrekturlexikon wurde das jeweilige, themenspezifische Crawl-

Teilkorpus\ Lexikon	$D^{E}_{\downarrow}$	D^{E+}	D^{EC}	D^{EC}_{E+}	D^{perf}
Mykologie (E)	OCR-Grundgenauigkeit: 97.09				
Abdeckung	80.93	88.74	97.09	97.14	100
Korrekturgenauigkeit	97.09	97.09	97.51	97.54	98.92
falsche Freunde	13	16	20	20	0
unvermeidbar	70	43	9	6	0
Inspektionsrate	19.73	12.59	4.90	4.80	3.15
Fische (E)	OCR-Grundgenauigkeit: 98.95				
Abdeckung	98.07	98.55	99.10	99.40	100
Korrekturgenauigkeit	99.01	99.00	99.08	99.06	99.45
falsche Freunde	30	30	36	36	10
unvermeidbar	10	10	7	7	8
Inspektionsrate	2.48	2.02	1.35	1.10	1.02
Holocaust (E)	OCR-Grundgenauigkeit: 98.97				
Abdeckung	96.27	97.83	99.19	99.37	100
Korrekturgenauigkeit	99.01	98.97	99.10	99.07	99.66
falsche Freunde	16	20	23	27	0
unvermeidbar	7	4	1	1	3
Inspektionsrate	4.38	2.77	1,38	1.11	1.14
Rom (E)	OCR-Grundgenauigkeit: 98.79				
Abdeckung	96.81	97.64	99.45	99.48	100
Korrekturgenauigkeit	98.83	98.83	99.07	99.06	99.67
falsche Freunde	20	22	38	39	6
unvermeidbar	21	19	3	3	0
Inspektionsrate	3.97	2.96	1.16	1.12	1.38
Speisepilze (E)	OCR-Grundgenauigkeit: 98.95				
Abdeckung	98.52	98.94	99.64	99.68	100
Korrekturgenauigkeit	99.25	99.25	99.34	99.39	99.68
falsche Freunde	12	14	23	24	0
unvermeidbar	5	4	2	1	4
Inspektionsrate	2.08	1.83	0.95	0.91	1.13
Neurologie (E)	OCR-Grundgenauigkeit: 98.81				
Abdeckung	98.31	99.06	99.83	99.83	100
Korrekturgenauigkeit	99.06	99.03	99.51	99.42	99.87
falsche Freunde	13	17	19	23	0
unvermeidbar	5	2	1	1	1
Inspektionsrate	2.59	1.84	1.05	0.99	1.22

Tabelle 10.1: Ergebnisse der englischen Teilkorpora.

Teilkorpus\ Lexikon	$D^G_\downarrow$	D^{G+}	D^{GC}	D^{GC}_{G+}	D^{perf}
Mykologie (G)	OCR-Grundgenauigkeit: 95.45				
Abdeckung	80.06	90.12	91.32	94.47	100
Korrekturgenauigkeit	95.49	95.53	95.59	95.78	97.98
falsche Freunde	40	55	53	69	11
unvermeidbar	93	57	46	39	14
Inspektionsrate	21.19	11.50	10.13	7.18	5.07
Fische (G)	OCR-Grundgenauigkeit: 98.43				
Abdeckung	77.56	89.72	92.28	93.24	100
Korrekturgenauigkeit	98.43	98.43	98.45	98.46	99.65
falsche Freunde	22	30	35	37	3
unvermeidbar	51	29	27	24	2
Inspektionsrate	22.08	9.99	7.64	6.59	2.03
Holocaust (G)	OCR-Grundgenauigkeit: 97.92				
Abdeckung	91.83	96.62	97.85	98.42	100
Korrekturgenauigkeit	98.03	98.26	98.53	98.69	99.47
falsche Freunde	15	20	20	24	2
unvermeidbar	29	13	15	11	2
Inspektionsrate	8.35	4.37	3.39	2.86	2.52
Rom (G)	OCR-Grundgenauigkeit: 98.55				
Abdeckung	84.29	90.92	96.51	96.89	100
Korrekturgenauigkeit	98.55	98.56	98.59	98.58	99.62
falsche Freunde	14	22	20	26	4
unvermeidbar	43	29	19	13	0
Inspektionsrate	15.24	9.07	4.00	3.52	1.45
Speisepilze (G)	OCR-Grundgenauigkeit: 97.45				
Abdeckung	77.66	86.09	92.11	93.07	100
Korrekturgenauigkeit	97.45	97.47	97.55	97.51	99.14
falsche Freunde	26	30	30	33	14
unvermeidbar	64	52	38	31	3
Inspektionsrate	21.80	13.73	7.99	7.03	2.36
Neurologie (G)	OCR-Grundgenauigkeit: 97.32				
Abdeckung	81.48	90.33	94.53	95.43	100
Korrekturgenauigkeit	97.42	97.38	97.70	97.70	99.47
falsche Freunde	28	35	38	43	4
unvermeidbar	56	38	20	17	6
Inspektionsrate	18.43	10.18	6.38	5.39	3.05

Tabelle 10.2: Ergebnisse der deutschen Teilkorpora.

Lexikon mit aktiviertem Flexions- und Orthographievarianten-Tool eingesetzt. Auf den Trainingsdaten wurde eine optimale Gewichtung α zwischen dem längensensitiven Levenshtein-Abstand und der Crawl-Frequenz durch eine Iteration in 20 Schritten ermittelt. Dazu wurde eine ebenfalls optimale Korrekturgrenze $score_{border}$ bestimmt. Auf den Trainingsdaten wurde die Grundgenauigkeit der OCR, die aus den optimalen Parametereinstellungen resultierende Genauigkeit der Nachkorrektur und die erzielte Verbesserung (Differenz der Prozentpunkte) ermittelt.
Die Parametereinstellungen wurden in der Testphase übernommen, und es wurde ebenfalls die Genauigkeit der Nachkorrektur und die Differenz zur Grundgenauigkeit der OCR in Prozentpunkten bestimmt.

Resultate

Die Tatsache, dass in der Trainingsphase auch alle Teilexperimente mit der zweiten OCR-Engine eine Verbesserung bis zu einem Prozentpunkt verzeichnen, bekräftigt den Eignungstest der Crawl-Lexika des vorherigen Abschnitts.
Die Verbesserung im Test fällt in der Regel etwas schlechter aus als im Training. Bemerkenswert ist, dass in allen Testfällen eine Verbesserung verzeichnet werden kann, denn durch zufälliges Raten der Parameter α und $score_{border}$ erhält man in den meisten Fällen entweder keine Verbesserung, wenn die Nachkorrektur zu vorsichtig agiert, oder sogar ein negatives Ergebnis, wenn die unglücklichen Korrekturen Überhand nehmen.
Die Werte von α sind so zu deuten, dass die Frequenzinformation stärker als der längensensitive Levenshtein-Abstand in die Kombinationsformel einfließt. Da aber alle Kandidaten durch eine Levenshtein-Automatenanfrage an die Lexika ohnehin mit einer Vorfilterung bzgl. des Levenshtein-Abstands ($\leqslant 2$) ausgewählt wurden, kann man aus den α-Werten nicht folgern, dass die Frequenz generell wichtiger als der Abstand ist.
Tendenziell rücken die Qualitätsresultate der beiden OCR-Engines durch die Nachkorrektur zusammen. In einem Fall überholt die schlechtere Engine sogar die bessere.
Im Vergleich der beiden Sprachen fallen lediglich die unterschiedlichen Werte für α auf. Es liegen die Vermutungen nahe, dass im Deutschen der Levenshtein-Abstand ein schwerere Gewichtung als im Englischen erfordert, und dass Trainingszeit eingespart werden kann, indem die Gewichtung nur sprachabhängig und nicht domainabhängig trainiert wird, da α innerhalb der Sprachgruppen nur schwach variiert.

10.3.2 Evaluation der Kombination von OCR-Engines

Experiment I

Die Kombination von zwei OCR-Engines wurde auf den Teilkorpora des Einzel-OCR-Experiments – das sind die Themengebiete Holocaust, Rom, Speisepilze und Neurologie in Englisch und Deutsch – getestet. Alle weiteren Einstellungen

	$\text{Verbesserung}_{Training}$		$\text{Verbesserung}_{Test}$		α	$score_{border}$
Neurologie (G)						
OCR-Engine$_1$	96.78%→97.52%	+0.74	97.78%→97.84%	+0.06	0.20	0.19783
OCR-Engine$_2$	96.39%→96.72%	+0.33	96.94%→97.18%	+0.24	0.10	0.10854
Holocaust (G)						
OCR-Engine$_1$	98.07%→98.60%	+0.53	97.99%→98.22%	+0.23	0.20	0.19739
OCR-Engine$_2$	97.48%→98.24%	+0.76	97.29%→97.97%	+0.68	0.15	0.15190
Rom (G)						
OCR-Engine$_1$	98.56%→98.70%	+0.14	98.71%→98.73%	+0.02	0.20	0.20009
OCR-Engine$_2$	98.47%→98.51%	+0.04	98.21%→98.29%	+0.08	0.20	0.20467
Speisepilze (G)						
OCR-Engine$_1$	98.40%→98.85%	+0.45	97.52%→97.97%	+0.45	0.10	0.10542
OCR-Engine$_2$	98.02%→98.63%	+0.61	96.35%→96.94%	+0.59	0.10	0.10340

Tabelle 10.3: Einzel-OCR-Ergebnisse der deutschen Teilkorpora.

	$\text{Verbesserung}_{Training}$		$\text{Verbesserung}_{Test}$		α	$score_{border}$
Neurologie (E)						
OCR-Engine$_1$	98.81%→99.69%	+0.88	98.71%→98.95%	+0.24	0.10	0.102920
OCR-Engine$_2$	97.84%→98.82%	+0.98	98.64%→99.10%	+0.46	0.10	0.100483
Holocaust (E)						
OCR-Engine$_1$	98.50%→98.87%	+0.37	99.01%→99.16%	+0.15	0.10	0.10771
OCR-Engine$_2$	98.04%→98.46%	+0.42	98.65%→98.71%	+0.06	0.10	0.106948
Rom (E)						
OCR-Engine$_1$	98.89%→99.53%	+0.64	98.80%→98.98%	+0.18	0.05	0.061064
OCR-Engine$_2$	99.19%→99.62%	+0.43	98.93%→99.14%	+0.21	0.10	0.103108
Speisepilze (E)						
OCR-Engine$_1$	99.08%→99.54%	+0.46	98.98%→99.38%	+0.40	0.10	0.10520
OCR-Engine$_2$	98.76%→98.90%	+0.14	98.76%→98.88%	+0.12	0.10	0.10912

Tabelle 10.4: Einzel-OCR-Ergebnisse der englischen Teilkorpora.

	$\text{Verbesserung}_{Training}$		$\text{Verbesserung}_{Test}$		α	$score_{border}$
Neurologie (G)	96.78%→98.52%	+1.74	97.78%→98.22%	+0.44	0.00	0.78493
Holocaust (G)	98.07%→99.18%	+1.11	97.99%→98.91%	+0.92	0.15	0.75443
Rom (G)	98.56%→99.28%	+0.72	98.71%→99.09%	+0.38	0.00	0.85427
Speisepilze (G)	98.40%→98.93%	+0.53	97.52%→98.03%	+0.51	0.15	0.729523

Tabelle 10.5: Kombinations-Ergebnisse I der deutschen Teilkorpora.

des Einzel-OCR-Experiments wurden beibehalten, d. h. Trainings-/Testaufteilung 3 zu 7, D^{GC} bzw. D^{EC} als Korrekturlexikon und angeschaltetes Varianten-Tool. Auch der Experimentablauf und das Erfassungsschema wurden übernommen. Auf den Trainingsdaten wurde eine optimale Gewichtung α zwischen den Konfidenzwerten der beiden OCR-Engines durch das gleiche Iterationsprinzip in 20 Schritten ermittelt zusammen mit einer ebenfalls optimalen Korrekturgrenze $score_{border}$. Auf den Trainingsdaten wurde neben der bereits vorhandenen Grundgenauigkeit der Master-OCR, die aus den optimalen Parametereinstellungen resultierende Genauigkeit der Kombination und die somit erzielte Verbesserung in Prozentpunkten gemessen.
Die optimalen Trainingsparameter wurden in der Testphase übernommen, und es wurde ebenfalls die Genauigkeit der Nachkorrektur und die Differenz gegenüber der Ausgangsgenauigkeit in Prozentpunkten erfasst. Da OCR-Engine_1 in 7 von 8 Teilexperimenten die präziseren Leseresultate erzielte, wurde sie insgesamt zur Master-Engine bestimmt.

Resultate I

Der Vergleich zwischen Training und Test liefert die gleichen Ergebnisse wie das Einzel-OCR-Experiment: die Zugewinne in der Testphase fallen erwartungsgemäß, in der Regel schwächer als in der Trainingsphase aus, und es wird in allen Fällen eine echte Verbesserung durch die Nachkorrektur erzielt.
Die Daten erlauben auch einen direkten Vergleich zwischen den Nachkorrekturergebnissen der Master-OCR-Engine alleine und denen der Kombination. Mit Ausnahme eines Falles ist die Kombination der einzelnen OCR-Engine überlegen.
Es fällt auf, dass die Gewichtung der Master-OCR α immer relativ niedrig eintrainiert wird und in drei Fällen sogar bei 0.00 liegt. Da aber die Master-OCR-Engine im Falle einer leeren Kandidatenliste immer den Korrekturvorschlag vorgibt, ist sie implizit immer stark involviert; im Deutschen noch mehr als im Englischen auf Grund der geringeren Abdeckungsrate, wodurch sich die im Deutschen niedrigeren α-Werte erklären.

	$\text{Verbesserung}_{Training}$		$\text{Verbesserung}_{Test}$		α	$score_{border}$
Neurologie (E)	98.81%→99.59%	+0.78	98.71%→99.51%	+0.80	0.10	0.70959
Holocaust (E)	98.50%→99.25%	+0.75	99.01%→99.22%	+0.21	0.00	0.889053
Rom (E)	98.89%→99.96%	+1.07	98.80%→99.29%	+0.49	0.40	0.600000
Speisepilze (E)	99.08%→99.86%	+0.78	98.98%→99.30%	+0.32	0.55	0.672101

Tabelle 10.6: Kombinations-Ergebnisse I der englischen Teilkorpora.

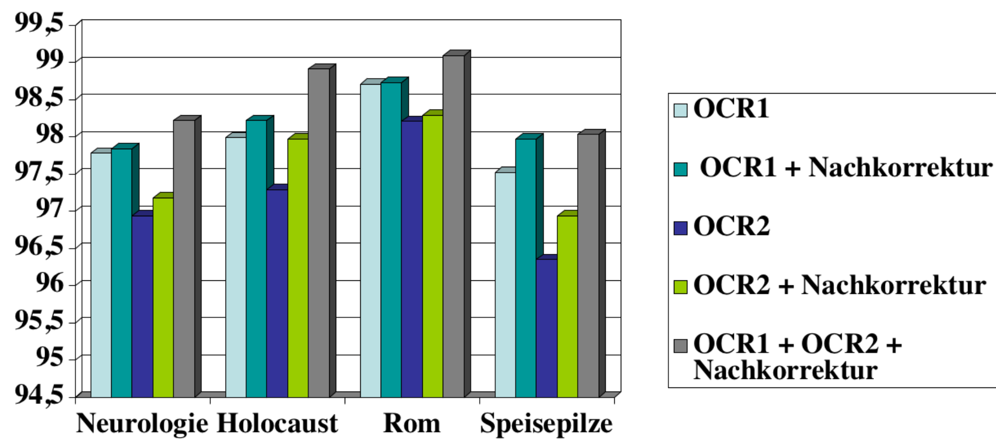


Abbildung 10.2: Nutzen der scorebasierten Kombination im Deutschen.

	D^{OCR_2}	Verbesserung$_{Tr}$	$D_{EC}^{OCR_2}$	Vergleich zu 10.5
Neurologie (G)	96.78%→98.26%	+1.48	96.78%→98.39%	-0.13
Holocaust (G)	98.07%→98.37%	+0.30	98.07%→98.77%	-0.41
Rom (G)	98.56%→98.80%	+0.24	98.56%→99.04%	-0.24
Speisepilze (G)	98.40%→98.93%	+0.53	98.40%→99.31%	+0.38

Tabelle 10.7: Kombinations-Ergebnisse II der deutschen Teilkorpora.

Experiment II

Die vereinfachte OCR-Engine-Kombination durch Umwandlung des zweiten OCR-Engine-Outputs in ein Lexikon, wurde ebenfalls auf den Teilkorpora Holocaust, Rom, Speisepilze und Neurologie in Englisch und Deutsch durchgeführt. Im ersten Teilexperiment wird nur mit dem Leseresultat der OCR-Engine$_2$ als Lexikon D^{OCR_2} korrigiert. Diese ansonsten lexikonfreie Variante zeigt den puren Effekt einer Kombination von OCR-Engines und erlaubt einen Direktvergleich mit Lexikonkorrekturen. Im zweiten Teilexperiment wurde mit der Kombination von D^{OCR_2} und dem jeweiligen Crawl-Lexikon D^{GC} bzw. D^{EC} korrigiert, um das lexikonbasierte Kombinationsverfahren mit dem scorebasierten zu vergleichen. Vereinfachend wurde dabei jeweils auf eine Testphase verzichtet. Es wurde auf den 30% Trainingsdaten eine optimale Gewichtung zwischen dem längensensitiven Levenshtein-Abstand und der Frequenz (Iteration in 20 Schritten) sowie eine optimalen Korrekturgrenze ermittelt. Bei der Lexikonauswahl D^{OCR_2} wird der direkte Genauigkeitszuwachs durch OCR-Engine-Kombination gegenüber dem reinen OCR-Resultat in Prozentpunkten angegeben. Die Hinzunahme der Crawl-Lexika erlaubt eine Quantifizierung des Nachteils dieses Kombinationsverfahrens gegenüber dem Score-Ansatz. Es wird die Differenz gegenüber dem Score-Ansatz in Prozentpunkten angegeben.

Resultate II

In erster Linie ist festzustellen, dass in allen Teilexperimenten eine echte Verbesserung gegenüber der Ausgangsgenauigkeit von OCR-Engine$_1$ erzielt wird. Eine Gegenüberstellung der reinen Nachkorrektur mit der zweiten OCR-Engine und den Trainingsverbesserungen aus den Tabellen 10.3 und 10.4, d. h. Nachkorrektur mit Crawl-Lexika, zeigt, dass beide Einzelmaßnahmen einen positiven Effekt gleicher Größenordnung mit sich bringen. Die Korrektur mit D^{OCR_2} ist der Korrektur mit D^{GC} bzw. D^{EC} leicht überlegen.
Eine Kombination der beiden Einzelmaßnahmen steigert in der Regel die Nachkorrekturqualität. Ein direkter Vergleich mit der scorebasierten Kombination belegt in 6 von 8 Fällen die Unterlegenheit des vereinfachten Kombinationsansatzes.
Es sind keine sprachspezifischen Auffälligkeiten auszumachen.

	D^{OCR_2}	Verbesserung$_{Tr}$	$D_{EC}^{OCR_2}$	Vergleich zu 10.6
Neurologie (E)	98.81%→99.38%	+0.57	98.81%→99.74%	+0.15
Holocaust (E)	98.50%→99.11%	+0.61	98.50%→99.01%	-0.24
Rom (E)	98.89%→99.62%	+0.73	98.89%→99.53%	-0.43
Speisepilze (E)	99.08%→99.72%	+0.64	99.08%→99.54%	-0.32

Tabelle 10.8: Kombinations-Ergebnisse II der englischen Teilkorpora.

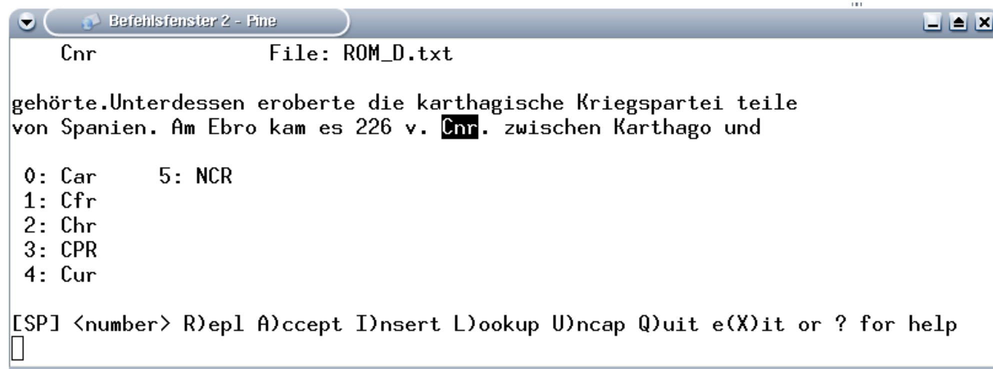


Abbildung 10.3: Korrekturinteraktion mit Auswahlliste.

10.4 Interaktive Korrektur

10.4.1 Adaption des UNIX-Tools ispell

Mit einer Adaption des UNIX-Tools ispell [90] demonstriere ich eine prototypische Umsetzung der ersten beiden Varianten interaktiver Korrektur, die in 3.2.2 vorgestellt werden.
Ispell präsentiert dem Korrektor alle Wörter, die nicht im Lexikon des Tools enthalten sind. Ich adaptiere das Tool für meine Zwecke, indem ich das Standardlexikons mit einer Wortliste austausche, die alle Wörter des OCR-Texts enthält, außer denen, die inspiziert werden sollen. Ispell bietet dem Korrektor die Möglichkeit, ein Wort zu akzeptieren, manuell zu korrigieren oder aus einer Korrekturkandidatenliste das korrekte Substitut zu wählen.

Da das Tool die Kandidaten aus seinem Lexikon auswählt, füge ich zu jedem Wort, das inspiziert werden soll, die Korrekturkandidaten meines XML-Korrekturfiles in die Austauschwortliste für ispell ein. Da diese Korrekturvorschläge identisch zu Inspektionswörtern sein können, die in der Wortliste fehlen müssen, um rückgefragt zu werden, entsteht die Gefahr falscher Freunde bzgl. der interaktiven Korrektur. Um diese Fehlerquelle näher zu quantifizieren habe ich eine Messreihe vorgenommen. Für vier deutsche Ergebnis-Dateien der OCR-Kombination wurde eine interaktive ispell-Nachkorrektur vorbereitet.

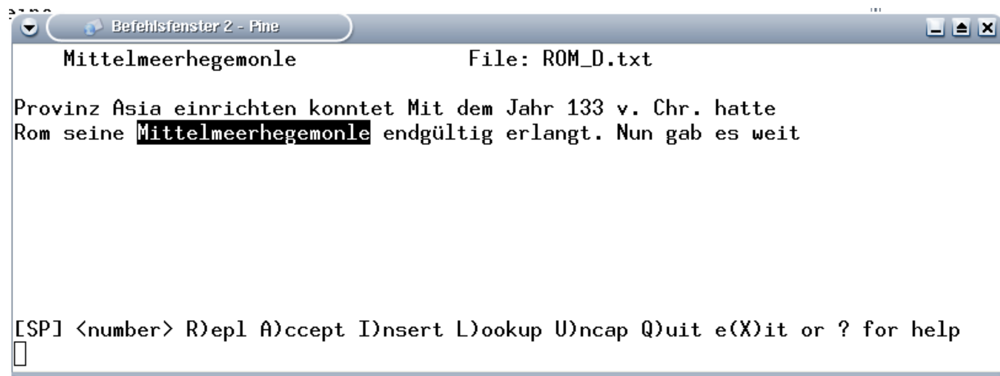


Abbildung 10.4: Korrekturinteraktion ohne Auswahlliste.

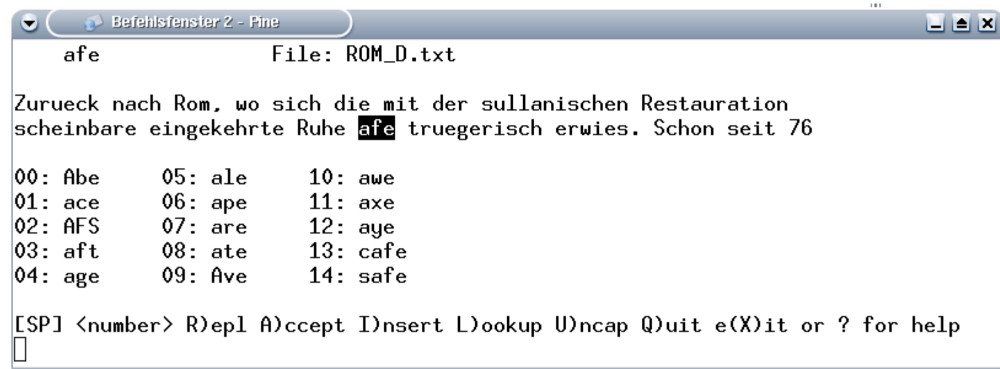


Abbildung 10.5: Der richtige Kandidat fehlt, da der Levenshtein-Abstand zu groß ist.

Die Ergebnis-Dateien enthalten pro Wort bis zu fünf Korrekturvorschläge und Korrekturstrategie ist, alle Wörter zu inspizieren, denen ein Korrekturkandidat mit Konfidenzwert größer eins fehlt. Die Messergebnisse (Holocaust, 18.75%), (Neurologie, 8.12%), (Rom, 20.69%) und (Speisepilze, 11.26%) zeigen, dass die Fehlerquelle nicht zu vernachlässigen ist.
Weitere Grenzen der ispell-Adaption liegen in der Bestimmung von Korrekturkandidaten. Es werden nur Kandidaten mit Levenshtein-Abstand gleich eins angezeigt[1]. Dadurch fällt ein Teil der von mir bereits ermittelten Korrekturkandidaten unter den Tisch.

Die Algorithmen zur Wortzerlegung und Bestimmung normaler Tokens unterscheiden sich geringfügig von meiner Implementierung. Damit wird aber das

[1]Transpositionen werden von ispell ebenfalls als unäre Operationen gewertet.

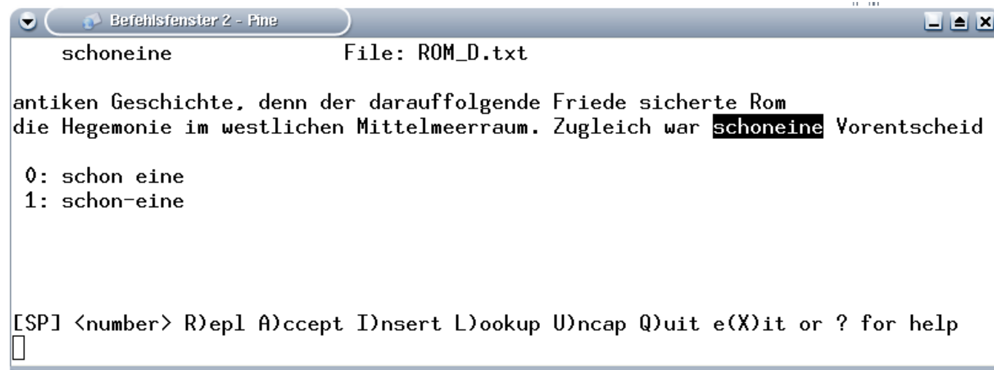


Abbildung 10.6: Mit ispell können z. T. auch Merges erkannt werden.

Rückmeldeverhalten in Richtung von ispell gelenkt. So wird bspw. das non-normale Token `Hi/er` nicht zusammen, sondern getrennt in `Hi` und `er` nachgefragt. Sofern sie nicht als falsche Freunde auftreten, werden auch Splits von ispell in zwei Schritten rückgefragt. Vermeintliche Merges hingegen löst eine in ispell integrierte Heuristik auf.

Auch ispell-Lexika lassen sich – ähnlich zu meinen eingesetzten Lexika – in eine für die Suche effiziente Datenstruktur vorkompilieren; es wird ein Hashing-Verfahren an Stelle eines Tries verwendet. Da jedoch die Länge einer Austauschwortliste nur in der Größenordnung der Teilkorpuslänge liegt, bleiben die Suchzeiten in einer unkompilierten, gar unsortierten Liste unmerklich gering.

10.5 Fazit

10.5.1 Status

Es konnte gezeigt werden, dass die Crawl-Lexika – sowohl alleine als auch in Kombination mit statischen Lexika – für eine OCR-Nachkorrektur geeignet sind. Obwohl die Anzahl falscher Freunde erwartungsgemäß steigt, fällt die Gesamtbilanz positiv aus. Nach einer Parameteroptimierung auf Trainingsmaterial konnte auf allen getesteten Teilkorpora eine echte Verbesserung des OCR-Resultats demonstriert werden. Mit einer scorebasierten Kombination zweier OCR-Engines ließ sich das Ergebnis noch weiter anheben, abgesehen von einer Ausnahme. Im Englischen wurden alle Endresultate deutlich über 99% gehievt, im Deutschen alle über 98%. Das einfachere, lexikonbasierte Kombinationsmodell von OCR-Engines ist dem scorebasierten Ansatz unterlegen. Mit der ispell-Adaption wurde gezeigt, dass meine Methoden zur Parameteroptimierung und OCR-Kombination auch für eine Interaktive Nachkorrektur eingesetzt werden können.

10.5.2 Verbesserungspotential

Die Ergebnisse legen die Vermutung nahe, dass eine Skalierung der Crawls zu besseren Abdeckungen und damit auch zu besseren Nachkorrekturresultaten führt, wobei für das Deutsche ein noch größerer Nutzen zu erwarten ist, da die Abdeckungen noch deutlich unter denen der englischen Teilkorpora liegen. Techniken zur Filterung fehlerbehafteter Web-Dokumente versprechen ein Senkung des Anteils falscher Freunde und damit eine Verbesserung der Nachkorrekturqualität. Eine Ausweitung der Score-Kombination meiner Software ist empfehlenswert, da damit einerseits zusätzliche Korrekturhilfen wie bspw. Web-Kookkurenzen eingebunden werden können und andererseits auch die Anzahl der OCR-Engines noch erhöht werden kann.

curriculum vitæ

Persönliche Daten:

Name:	Christian M. Strohmaier
Geburtstag:	6.10.1971
Geburtsort:	Burghausen an der Salzach
Staatsangehörigkeit:	deutsch
Familienstand:	ledig, keine Kinder

Ausbildung:

WiSe 01/02 - WiSe 04/05	Promotion in Computerlinguistik mit den Nebenfächern Informatik und Statistik an der Ludwig-Maximilians-Universität München unter Betreuung von Prof. Klaus U. Schulz zum Thema: Methoden der lexikalischen Nachkorrektur OCR-erfasster Dokumente.
WiSe 98/99 - SoSe 01	Promotionsvorhaben in Informatik an der Ludwig-Maximilians-Universität München
WiSe 91/92 - SoSe 98	Studium der Informatik mit Nebenfach Statistik an der Ludwig-Maximilians-Universität München. Diplomarbeit unter Betreuung von Prof. Rudolf Bayer: UNICO, eine OMNIS-Datenbank zur Hochschulkooperation mit der Industrie.
10/82 - 07/91	Willi-Graf-Gymnasium München.
10/78 - 07/82	Burmester-Grundschule München.

Berufserfahrung:

ab 10/04	Senior Projektleiter, CreaLog München.
10/00 - 09/04	wissenschaftlicher Assistent, Eberhard-Karls-Universität Tübingen.
08/00 - 09/04	freiberuflicher IT-Trainer (Java, XML, Solaris und UML), Sun Microsystems Deutschland (Educational Services).
05/98 - 04/02	Förderung meines Promotionsvorhabens sowie Projektarbeit, Siemens AG.
08-09/96, 03-04/96 08-10/94, 03-04/94 08-10/92, 03-04/92	Werkstudent, Siemens Nixdorf Informationssysteme AG.

Weiterbildung:

01/00 - 12/01	diverse einwöchige Fach- und and Präsentationskurse inklusive Zertifizierung zum: Sun Certified Programmer for the Java^{TM}2 Platform Sun Microsystems Deutschland (Educational Services).
03/92 - 08/94	diverse einwöchige Informatik- und Rhetorikkurse, Siemens Nixdorf Informationssysteme AG.

Sprachen:

Deutsch:	Muttersprache
Englisch:	fließend
Französisch:	Grundkenntnisse
Spanisch:	Grundkenntnisse

Publikationen:

- mit Stoyan Mihov, Svetla Koeva, Christoph Ringlstetter und Klaus U. Schulz: *Precise and Efficient Text Correction using Levenshtein Automata, Dynamic Web Dictionaries and Optimized Correction Models.* Proceedings of the Workshop on International Proofing Tools and Language Technologies, Patras, 2004.
- mit Christoph Ringlstetter, Klaus U. Schulz und Stoyan Mihov: *A visual and interactive tool for optimizing lexical postcorrection of OCR-results.* Proceedings of the Workshop on Document Image Analysis and Retrieval DIAR'03.
- mit Christoph Ringlstetter, Klaus U. Schulz und Stoyan Mihov: *Lexical postcorrection of OCR-results: The web as a dynamic secondary dictionary?* Proceedings of the 7th International Conference on Document Analysis and Recognition ICDAR'03.
- *Aktuelles Schlagwort XML-Strukturakquisition.* Informatik Spektrum, Aug./Sept. 2002.
- mit Holger Meuss: *A Filter for Structured Document Retrieval.* CIS Bericht 99-123.
- mit Holger Meuss: *Improving Index Structures for Structured Document Retrieval.* 21st Annual Colloquium on IR Research (IRSG'99).

Mitgliedschaften:

- Gesellschaft für Informatik
- assoziiertes Mitglied im DFG-Graduiertenkolleg SIL (Sprache, Information, Logik)

Literaturverzeichnis

[1] Apache Software Foundation. Tomcat. `jakarta.apache.org/tomcat/`.

[2] Stephan Baumann, Michael H. Malburg, Hans-Günther Hein, Rainer Hoch, Thomas Kieninger, and Norbert Kuhn. Document analysis at DFKI part 2: Information extraction. Technical Report RR-95-03, Deutsches Forschungszentrum für Künstliche Intelligenz GmbH, 1995.

[3] Sergey Brin and Lawrence Page. The anatomy of a large-scale hypertextual Web search engine. *Computer Networks and ISDN Systems*, 30(1–7):107–117, 1998.

[4] Sabine Carbonnel and Eric Anquetil. Lexical post-processing optimization for handwritten word recognition. In *Proc. Int. Conference on Document Analysis and Recognition (ICDAR), Edinburgh*, pages 477–481, 2003.

[5] G. Chang, M. J. Healey, J. A. M. McHugh, and J. T. L. Wang. *Mining the World Wide Web: An Information Search Approach.* Kluwer Academic Publishers, Norwell, Massachusetts, 2001.

[6] Allen S. Condit. Autotag: A tool for creating structured document collections from printed materials. Master's thesis, University of Nevada, Las Vegas, 1995.

[7] Daniel S. Connelly, Beth Paddock, and Rebecca Rice. *XDOC Data Format, Technical Specification Version 3.0.* Xerox Corporation, may 1997.

[8] James H. Coombs, Allen H. Renear, and Steven J. DeRose. Markup systems and the future of scholarly text processing. *Communications of the ACM 30(11)*, pages 933–947, November 1987.

[9] Jan Daciuk, Stoyan Mihov, and and Richard Watson Bruce Watson. Incremental construction of minimal acyclic finite state automata. *Computational Linguistics*, 26(1):3–16, 2000.

[10] Fred J. Damerau. Evaluating computer-generated domain-oriented vocabularies. *Information Processing and Management*, 26(6):791–801, 1990.

[11] Fred .J. Damerau. Generating and evaluating domain-oriented multi-word terms from texts. *Information Processing and Management*, 29(4):433–448, 1993.

[12] Fred .J. Damerau and E. Mays. An examination of undetected typing errors. *Information Processing and Management*, 25(6):659–664, 1989.

[13] A. Dengel, R. Hoch, F. Hönes, M. Malburg, and A. Weigel. *Handbook on Optical Character Recognition and Document Analysis*, chapter Techniques for Improving OCR Results, pages 227–258. World Scientific Publication Company, 1997.

[14] Ted Dunning. Accurate methods for the statistics of surprise and coincidence. *Computational Linguistics*, 19(1):61–74, 1994.

[15] Jeffrey Esakov, Daniel P. Lopresti, Jonathan S. Sandberg, and Jiangying Zhou. Issues in automatic OCR error classification. In *Proceedings of Third Annual Symposium on Document Analysis and Information Retrieval, Las Vegas, Nevada*, 1994.

[16] Michael D. Garris, Stanlay A. Janet, and William W. Klein. Federal register document image database, nist special database 25 (vol. 1). Technical Report NISTIR 6245, National Institute of Standards and Technology (NIST), 1998.

[17] Mathias Géry and Hatem Haddad. Web as huge information source for noun phrases: Integration in the information retrieval process. In *Proc. of the 2002 International Conferencen Information and Knowledge Engineering, Nevada, USA*, June 2002.

[18] David Gibson, Jon Kleinberg, and Prabhakar Raghavan. Inferring web communities from link topology. In *Proc. 9th ACM Conference on Hypertext and Hypermedia*, 1998.

[19] Clemens Gleich. Lesen und lesen lassen. Die großen OCR-Programme im Test. Pruefstand Texterkennung: Abbyy FineReader, Scansoft Omnipage, Iris ReadIris. *c't*, 1(24/2):142, 2002.

[20] GNU. Gnu ocrad. `www.gnu.org/software/ocrad/ocrad.html`.

[21] GNU. Gnu wget. `wget.sunsite.dk`.

[22] D. Goldsmith and M. Davis. A mail-safe transformation format of unicode, RFC2152, 1994.

[23] Google. Google Web APIs (beta). `www.google.de/apis/`.

[24] Susan E. Hauser, Jonathan Schlaifer, Theseen F. Sabir, Dina Demner-Fushman, and George R. Thoma. Correcting OCR text by association with historic datasets. In *Proc. of SPIE Conference on Electronic Imaging*, 2003.

[25] Klaus Heller. *IDS Sprachreport Extraausgabe. Informationen und Meinungen zur deutschen Sprache*, chapter Rechtschreibreform. Eine Zusammenfassung von Dr. Klaus Heller. Institut für deutsche Sprache, Mannheim, 1996.

[26] Oliver Hitz and Rolf Ingold. Visualization of document recognition results using XML technology. In *Colloque International sur le Document Electronique (CIDE)*, Lyon, France, July 2000.

[27] Oliver Hitz, Lyse Robadey, and Rolf Ingold. Using XML in document recognition. In *Document Layout Interpretation and its Applications (DLIA99)*, 1999.

[28] Rainer Hoch and Thomas Kieninger. On virtual partitioning of large dictionaries for contextual postprocessing to improve character recognition. *International Journal of Pattern Recognition and Artificial Intelligence*, 4(10):273–289, 1996.

[29] P. Hoffman and F. Yergeau. UTF-16, a transformation format of ISO 10646, RFC2781, 2000.

[30] T. Hong and J. J. Hull. Visual inter-word relations and their use for OCR postprocessing. In *Proc. Int. Conference on Document Analysis and Recognition (ICDAR), Montral*, 1995.

[31] Tao Hong. *Degraded Text Recognition using Visual and Linguistic Context*. PhD thesis, University of New York at Buffalo, September 1995.

[32] J. Hu, R. Kashi, D. Lopresti, G. Nagy, and G. Wilfong. Why table groundtruthing is hard. In *Proc. Int. Conference on Document Analysis and Recognition (ICDAR), Seattle*, pages 129–133, 2001.

[33] ISRI Staff. OCR Accuracy Produced by the current DOE Document Conversion System. Technical Report 2002-06, Information Science Research Institute(ISRI), University of Nevada, Las Vegas, 2002.

[34] Jani Jaakkola and Pekka Kilpeläinen. Using sgrep for querying structured text files. Technical Report C-1996-83, Department of Computer Science, University of Helsinki, 1996.

[35] Jani Jaakkola and Pekka Kilpeläinen. Nested text-region algebra. Technical Report C-1999-2, Department of Computer Science, University of Helsinki, 1999.

[36] Rong Jin, Alex G. Haupmann, and ChengXiang Zhai. A content-based probabilistic correction model for OCR document retrieval. In *Proc. of the SIGIR 2002, Workshop on Information Retrieval and OCR: From Converting Content to Grasping Meaning, Tampere, Finland*, 2002.

[37] Paul B. Kantor and Ellen M. Voorhees. The TREC-5 confusion track: Comparing retrieval methods for scanned text. *Information Retrieval*, 2(2/3):165–176, 2000.

[38] T. Kanungo, G. A. Marton, and O. Bulbul. Omnipage vs. sakhr: Paired model evaluation of two arabic ocr products. In *Proceedings of SPIE Conference on Document Recognition and Retrieval VI*, 1999.

[39] Tapas Kanungo and Robert M. Haralick. An automatic closed-loop methodology for generating character groundtruth for scanned documents. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 21(2):179–183, 1998.

[40] Tapas Kanungo, Chang H. Lee, Jeff Czorapinski, and Ivan Bella. TRUEVIZ: a groundtruth/metadata editing and visualizing toolkit for OCR. In *Proceedings of SPIE Conference on Document Recognition and Retrieval*, 2001.

[41] Thomas Kistler and Hannes Marais. Webl - a programming language for the web. Technical report, DIGITAL Systems Research Center, Dezember 1997.

[42] S. T. Klein and M. Kopel. A voting system for automatic OCR correction. In *Proc. of the SIGIR 2002, Workshop on Information Retrieval and OCR: From Converting Content to Grasping Meaning, Tampere, Finland*, 2002.

[43] André Kramer. Rechtschreibkorrektursysteme im Vergleich. DITECT versus Microsoft Word. `www.mediensprache.net/de/networx/`, 2004.

[44] Christian Kreibich. Seminararbeit: Kombination von OCR-Engines zur Fehlerkorrektur, 2002.

[45] Karen Kukich. Techniques for automatically correcting words in texts. *ACM Computing Surveys*, pages 377–439, 1992.

[46] D. Landau, R. Feldman, O. Zamir, Y. Aumann, M. Fresko, Y. Lindell, and O. Lipshtat. TextVis: An Integrated Visual Environment for Text Mining. In *Proceedings of the 2nd European Symposium on Principles of Data Mining and Knowledge Discovery*, pages 135–148, September 1998.

[47] Thomas A. Lasko and Susan E. Hauser. Approximate string matching algorithms for limited-vocabulary OCR output correction. In *Proceedings of SPIE, Vol. 4307, Document Recognition and Retrieval VIII*, pages 232–240, 2001.

[48] Philippe Lefèvre and François Reynaud. ODIL : an SGML Description Language of the Layout of Documents. In *Proc. Int. Conference on Document Analysis and Recognition (ICDAR), Montral*, pages 480–488, 1995.

[49] Lothar Lemnitzer. *Sprache zwischen Theorie und Technologie. Festschrift fuer Wolf Paprotté zum 60. Geburtstag*, chapter Ist das nicht doch alles das Gleiche? Regeln und Distanzmaße zur Berücksichtigung orthographischer Idiosynkrasien bei der Abbildung von Textsegmenten auf lexikalische Einheiten, pages 135–148. Wiesbaden, 2003.

[50] V.I. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. *Sov. Phys. Dokl.*, 1966.

[51] Yanhong Li, Daniel P. Lopresti, George Nagy, and Andrew Tomkins. Validation of image defect models for optical character recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 18 (2):99–108, 1996.

[52] Christopher D. Manning and Hinrich. Schütze. *Foundations of Statistical Natural Language Processing*. MIT Press, Cambridge MA, 1999.

[53] Tony McEnery and Andrew Wilson. *Corpus Linguistics*. Edinburgh University Press, 1996.

[54] Robert C. Miller and Krishna Bharat. SPHINX: A framework for creating personal, site-specific web crawlers. *Computer Networks and ISDN Systems*, 30(1–7):119–130, 1998.

[55] Andreas Myka and Ulrich Güntzer. Automatic hypertext conversion of paper document collections. In *Advances in Digital Libraries, number 916 in Lecture Notes in Computer Science*, pages 65–90, 1995.

[56] Andreas Myka and Ulrich Güntzer. Fuzzy Full-Text Searches in OCR Databases. In *Proceedings of the ADL '95 Forum, McLean, Virginia, USA*, pages 131–45, 1995.

[57] M. Najork and A. Heydon. High-performance web crawling (chapter 2). In J. Abello, P. Pardalos, and M. Resende, editors, *Handbook of Massive Data Sets*. Kluwer Academic Publishers, 2001.

[58] NEC. ResearchIndex CiteSeer. `citeseer.nj.nec.com/cs`.

[59] NIST. TREC-5 Confusion Track. `trec.nist.gov/data/confusion/`, 1994.

[60] Oleg G. Okun and Ari Vesanen. Experimental tool for generating ground truths for skewed page. In *Proceedings of SPIE Conference on Document Recognition and Retrieval*, 2001.

[61] Yuliya Palchaninava. Kontextuelle Verfahren bei der OCR-Korrektur. Master's thesis, Ludwig-Maximilians-Universität München, 2003.

[62] James L. Peterson. Computer programs for detecting and correcting spelling errors. *Communications of the ACM*, 23(12):676–687, 1980.

[63] James L. Peterson. A note on undetected typing errors. *Communications of the ACM*, 29(7):633–637, 1986.

[64] F. Pinard. Free recode. `www.iro.umontreal.ca/contrib/recode/HTML/`.

[65] Christian Raum. Der Info 21-Überblick: Hersteller von Erkennungssoftware und -lösungen. *info 21*, 1:20–21, 2003.

[66] S. V. Rice, F. R. Jenkins, and T. A. Nartker. The Fifth Annual Test of OCR Accuracy. Technical Report TR-96-01, University of Nevada, Las Vegas, 1996.

[67] Christoph Ringlstetter. OCR-Korrektur und Bestimmung von Levenshtein-Gewichten. Master's thesis, Ludwig-Maximilians-Universität München, 2003.

[68] Eric Sven Ristad and Peter N. Yianilos. Learning string-edit distance. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 20(5):522–532, 1998.

[69] Geoffrey Sampson. Book review: Analyzing language in restricted domains. *Computional Linguistics*, 16(2):113–116, 1990.

[70] J. Sauvola and H. Kauniskangas. Mediateam document database ii, a cd-rom collection of document images, university of oulu, finland, 1999.

[71] Klaus U. Schulz. Vorlesungsskript: Korrekturverfahren bei der optischen Charaktererkennung und Dokumentenanalyse. Technical report, Ludwig-Maximilians-Universität München, 2003.

[72] Klaus U. Schulz and Stoyan Mihov. Fast string correction with levenshtein-automata. *International Journal of Document Analysis and Recognition*, 5(1):67–85, 2002.

[73] Giovanni Seni, V. Kripasundar, and Rohini K. Srihari. Generalizing edit distance to incorporate domain information: Handwritten text recognition as a case study. *Pattern Recognition*, 29(3), 1996.

[74] Christian Strohmaier, Christoph Ringlstetter, Klaus U. Schulz, and Stoyan Mihov. Lexical postcorrection of OCR-results: The web as a dynamic secondary dictionary? In *Proc. Int. Conference on Document Analysis and Recognition (ICDAR), Edinburgh*, pages 1133–1137, 2003.

[75] Christian Strohmaier, Christoph Ringlstetter, Klaus U. Schulz, and Stoyan Mihov. A visual and interactive tool for optimizing lexical postcorrection of OCR results. In *Proceedings of the Workshop on Document Image Analysis and Retrieval (DIAR'03), Madison Wisconsin*, 2003.

[76] Christian M. Strohmaier. Aktuelles Schlagwort: XML-Strukturakquisition. *Informatik Spektrum*, 25(4):262–265, 2002.

[77] Sun Microsystems. The source for java technology. `java.sun.com/`.

[78] The Unicode Consortium. *The Unicode Standard, Version 4.0.* Addison-Wesley, 2003.

[79] Arno Unkrig. html2text. `www.userpage.fu-berlin.de/~mbayer/tools/`.

[80] U.S. National Library of Medicine. History of medicine, 2001.

[81] Christopher C. Vogt and Garrison W. Cottrell. Fusion via a linear combination of scores. *Information Retrieval*, 1(3):151–173, 1999.

[82] W3C. HTML 4.01 Specification, W3C Recommendation. www.w3.org/TR/html401/, December 1999.

[83] W3C. XML Path Language (XPath) Version 1.0, W3C Recommendation. www.w3.org/TR/xpath, November 1999.

[84] World Wide Web Consortium: Web Services Architecture Requirements. `www.w3.org/TR/wsa-reqs/`, November 2002. Editors Daniel Austin, Abbie Barbir, Christopher Ferris and Sharad Garg.

[85] R. A. Wagner and M. Fischer. The string to string correction problem. *ACM Journal*, 21(1):168–173, 1974.

[86] Donald E. Walker and Robert A. Amsler. *Analyzing Language in Restricted Domains*, chapter The Use of Machine-Readable Dictionaries in Sublanguage Analysis. LEA, Hillsdale, NJ, 1986.

[87] Yalin Wang, Ihsin T. Phillips, and Robert Haralick. Automatic table ground truth generation and a background-analysis-based table structure extraction method. In *Proc. Int. Conference on Document Analysis and Recognition (ICDAR), Seattle*, pages 528–532, 2001.

[88] A. Weigel, S. Baumann, and J. Rohrschneider. Lexical postprocessing by heuristic search and automatic determination of the edit costs. In *Proc. of the third International Conference on Document Analysis and Recognition (ICDAR)*, pages 857–860, 1995.

[89] A. Weigel, T. Jäger, and A. Pies. Estimation of probabilities for edit operations. In *Proc. of the International Conference on Pattern Recognition (ICPR'00), Barcelona, Spain*, pages 781–784, 2000.

[90] Pace Willisson and Geoff Kuenning. International Ispell – A spell-checking program for Unix. `fmg-www.cs.ucla.edu/geoff/ispell.html`.

[91] B.A. Yanikoglu and L. Vincent. Ground-truthing and benchmarking document page segmentation. In *Proc. Int. Conference on Document Analysis and Recognition, Montreal*, 1995.

[92] F. Yergeau. UTF-8, a transformation format of ISO 10646, RFC2279, 1998.

[93] L. Zabala. ocre. `lem.eui.upm.es/ocre.html`.

[94] D. Zeinalipour-Yazti and M. Dikaiakos. Design and implementation of a distributed crawler and filtering processor. In *The Fifth Workshop on Next Generation Information Technologies and Systems (NGITS'2002) , Caesarea, Israel*, 2002.

[95] J. J. Zhu and L. H. Ungar. String edit analysis for merging databases. In *Proc. of the 6th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Workshop on Text Mining, Boston, MA, USA*, 2000.

[96] M. Zierl. Entwicklung und Implementierung eines Datenbanksystems zur Speicherung und Verarbeitung von Textkorpora. Master's thesis, Friedrich-Alexander-Universität Erlangen-Nürnberg, 1997.

[97] G. K. Zipf. The psycho-biology of language. In *Houghton Mifflin, Boston*, 1935.