Freie Universität Berlin Institut für Informatik

Diplomarbeit

Suche von ähnlichen Datensätzen unter Echtzeitbedingungen

Johannes Krugel
krugel@inf.fu-berlin.de

30. Mai 2008

vorgelegt bei
Prof. Dr. Heinz Schweppe

Vorwort

Ich danke Prof. Schweppe und Jürgen Broß für die Denkanstöße und kritischen Nachfragen, den Mitarbeitern der Firma PostCon für die Unterstützung und angenehme Atmosphäre sowie meiner Familie und Freunden für das Korrekturlesen und die moralische Unterstützung.

Erklärung der Selbstständigkeit

Hiermit versichere ich, dass ich diese Diplomarbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Berlin, den 30. Mai 2008

Zusammenfassung

Diese Diplomarbeit ist aus einer Kooperation zwischen der Arbeitsgruppe Datenbanken und Informationssysteme am Institut für Informatik der Freien Universität Berlin und der PostCon Deutschland GmbH entstanden. Ziel der Arbeit ist die Entwicklung eines Systems zur Korrektur mittels optischer Zeichenerkennung (OCR) gelesenen Adressen anhand einer Datenbank der korrekten Adressen. Die Korrektur einer Adresse muss dabei in Echtzeit (100 ms) geschehen. Dieses Problem der Adresskorrektur wird verallgemeinert zur Suche von ähnlichen Datensätzen in einer Datenbank. Zunächst wird dieses allgemeinere Problem gelöst und darauf aufbauend ein System zur Adresskorrektur entwickelt. Es werden verschiedene Ähnlichkeitsmaße sowohl für Zeichenketten als auch für Datensätze auf ihre Eignung untersucht. Dabei stellt sich die gewichtete Editierdistanz (Levenshtein-Distanz) als besonders geeignet heraus, um die Fehler der optischen Zeichenerkennung zu modellieren. Die Gewichte der Editieroperationen werden anhand einer Trainingsmenge erlernt. Das gewählte Ähnlichkeitsmaß für Datensätze basiert auf einer mit Hilfe einer Support Vector Machine gelernten Klassifikationsfunktion. Weil bisherige Verfahren zur Indizierung von Datensätzen (metrische Bäume, k-d-Bäume, R-Bäume) eine Suche nach ähnlichen Datensätzen bezüglich der gewählten Ähnlichkeitsfunktion nicht effizient durchführen können, wird eine neue Indexstruktur für Datensätze vorgeschlagen. Sie heißt Fri (Fast Record Index) und basiert auf mehreren Tries als Indizes für Zeichenketten. Es wird ein Algorithmus entwickelt, um in dieser Datenstruktur effizient die ähnlichsten Datensätze zu einem Anfragedatensatz zu ermitteln. Die Effizienz dieser Datenstruktur wird im Vergleich mit einer einfacheren Indexstruktur für Datensätze evaluiert und erreicht eine deutliche Verringerung der Anfragezeit. Bei der Evaluation des Systems zur Adresskorrektur mit Hilfe einer Testmenge stellt sich heraus, dass die korrekte Adresse in 97,5 % der Fälle korrekt bestimmt werden konnte.

Abstract

This thesis originates from a cooperation between the working group Databases and Information Systems at the department of computer science at the Freie Universität Berlin and the PostCon Deutschland GmbH. This work aims at developing a system for correcting addresses read by optical character recognition (OCR) on the basis of a database of correct addresses. The process of correction has to be accomplished in real-time (100 ms). This problem generalized to searching for similar records in a database. This second, more general problem is solved first, followed by the development of a method for correcting addresses. Various similarity measures for both strings and records are considered and studied for applicability. The weighted edit distance (Levenshtein Distance) turns out to be particularly suitable for modelling the errors of optical character recognition. The weights are machine learned with the aid of a training set. The similarity measure for records is based on a classification function learned by a support vector machine. Traditional indices for records (metric trees, kd-trees, R-trees) are not able to support efficient searching for records based on the chosen similarity measure. Therefore a new index for records is proposed. It is called Fri (Fast Record Index) and is based on tries for indexing strings. An algorithm is developed to facilitate efficient retrieval of records similiar to a given query record. The efficency of the data structure is experimentally compared to a simpler trie-based index and achieves a considerable decrease in runtime. The system for address correction is evaluated with a test set and is able to find the correct address in 97.5 %.

Inhaltsverzeichnis

1. Einleitung

Ziel dieser Diplomarbeit ist die Entwicklung eines Systems zur Korrektur von Adressen, die mit einer optischen Zeichenerkennung (OCR, engl.: Optical Character Recognition) von Briefen gelesen werden. Für diese Korrektur kann eine Datenbank aller korrekten Adressen in Deutschland verwendet werden. Das Problem der Adresskorrektur wird in dieser Arbeit verallgemeinert zur Suche von ähnlichen Datensätzen in einer Datenbank.

In den folgenden Abschnitten wird zunächst das Umfeld erläutert und die Motivation zur Lösung dieses Problems gegeben. Nachfolgend werden die technischen Anforderungen beschrieben und die Zielsetzung dieser Arbeit als sechs zu lösende Fragestellungen beschrieben.

1.1. Umfeld

Die Firma PostCon [Pos08] ist ein *Briefkonsolidierer*. Sie holt Briefe bei ihren Kunden ab, sortiert diese in einem eigenen Sortierzentrum nach der Postleitzahl, und liefert sie dann gebündelt bei der Deutschen Post zur anschließenden Zustellung ein. Für die Leistung der Vorsortierung gewährt die Deutsche Post Rabatte auf das Porto, die dann teilweise an die Kunden weitergegeben werden.

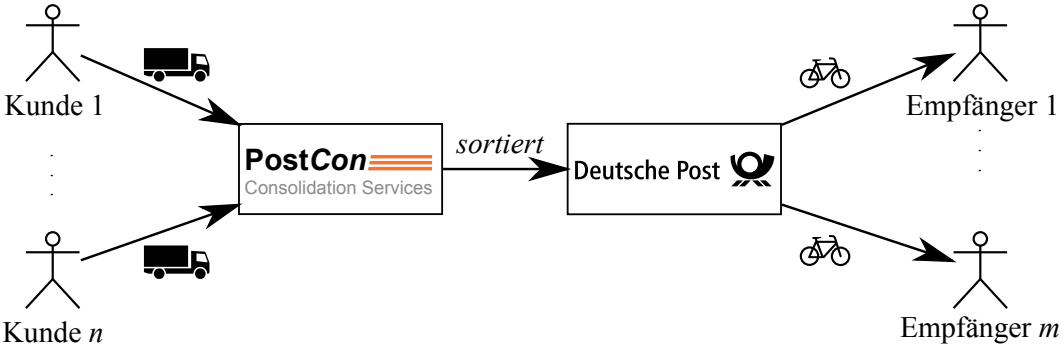

Abbildung 1.1.: Umfeld

Gegründet wurde PostCon im Jahre 2002, arbeitete zunächst als Briefgenossenschaft und nach einer Liberalisierung des Briefmarktes als Aktiengesellschaft. Im Jahr 2007 wurde PostCon von dem niederländischen Unternehmen TNT gekauft und anschließend in eine Gesellschaft mit beschränkter Haftung umgewandelt. Zu Beginn des Jahres 2008 verfügt PostCon über 13 Standorte in Deutschland und beschäftigt insgesamt ungefähr 300 Mitarbeiter. Täglich werden bundesweit bis zu 1,5 Mio Briefe bei ungefähr 1.000 mittleren und großen Versendern eingesammelt, sortiert und weitergeleitet (siehe Abbildung 1.1). Für die Zukunft ist außerdem geplant, nicht alle Briefe bei der Deutschen Post einzuliefern, sondern teilweise auch selbst mit eigenen Zustellern auszuliefern.

Dieser Absatz beschreibt den Prozess der Briefsortierung genauer. Die bei den Kunden abgeholten Briefe werden unsortiert auf die Sortiermaschine gelegt und dann automatisch einzeln nacheinander eingezogen. Sie werden von Förderbändern transportiert und passieren nacheinander zuerst eine Zeilenkamera[1] und wenige Meter weiter eine Weiche, an der die Briefe in die verschiedenen Fächer sortiert werden. Dieser Prozess ist in Abbildung 1.2 vereinfacht dargestellt.

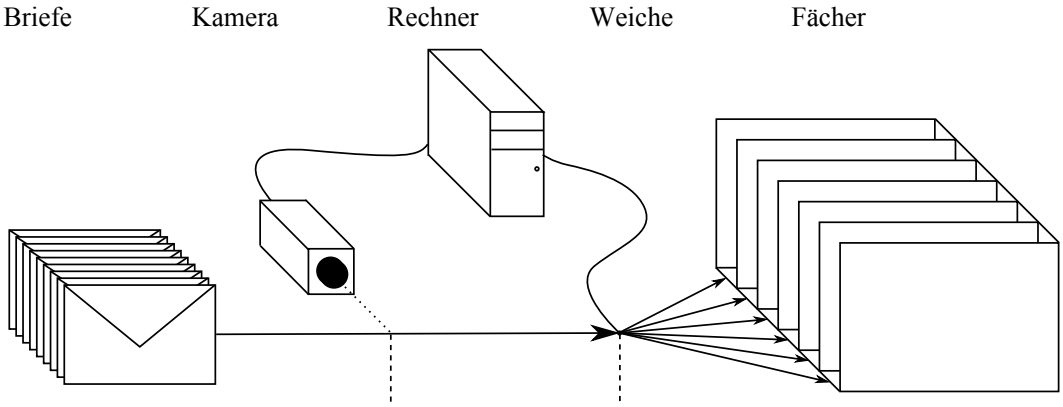

Abbildung 1.2.: Briefsortiermaschine (schematisch)

Die Zeilenenkamera nimmt von jedem Brief ein Bild auf, das anschließend von einer optischen Zeichenerkennung (OCR, engl.: Optical Character Recognition) verarbeitet wird. Diese analysiert das Bild und versucht die Zeichenkette zu extrahieren, die der aufgedruckten Adresse des Empfängers entspricht. Je nachdem welche Postleitzahl erkannt wurde, wird der Brief dann an der Weiche in eines der Fächer geleitet. Es gibt dabei jeweils ein Fach für jede Leitregion.[2] Abbildung 1.3 stellt dieses Verfahren und die beteiligten Komponenten dar.

1.2. Motivation

Einige der Briefe werden bei dem beschriebenen Prozess jedoch nicht in das korrekte Fach einsortiert, was verschiedene Gründe haben kann. Einerseits hat sich der Sender des Briefes möglicherweise vertippt, so dass eine falsche Postleitzahl auf dem Brief geschrieben ist. Andererseits geschehen auch bei der optischen Zeichenerkennung regelmäßig Fehler, wodurch die Postleitzahl nicht korrekt erkannt wird. Diese Fehler führen dazu, dass Briefe in ein falsches Fach einsortiert und damit zunächst zu einem falschen Bestimmungsort transportiert werden. Erst bei der Zustellung wird dieser Fehler bemerkt und die Briefe können in den korrekten Ort weitergeleitet werden. Dies ist in der Regel mit einer Verzögerung der Auslieferung um mindestens einen Tag verbunden. Die Kunden erwarten eine Zustellung der Briefe einen Tag nach dem Einlieferungsdatum.[3] Eine rechtzeitige Zustellung ist für PostCon also essentiell

[1] Eine *Zeilenkamera* ist eine Kamera, die nicht das ganze Bild auf einmal, sondern zeilenweise aufnimmt, während das Objekt die Kamera passiert.
[2] Eine Leitregion wird durch die zwei ersten Ziffern einer Postleitzahl repräsentiert. Die Leitregion der Postleitzahl „10963“ ist beispielsweise „10“.
[3] Im Fachjargon wird dies mit *E+1* bezeichnet.

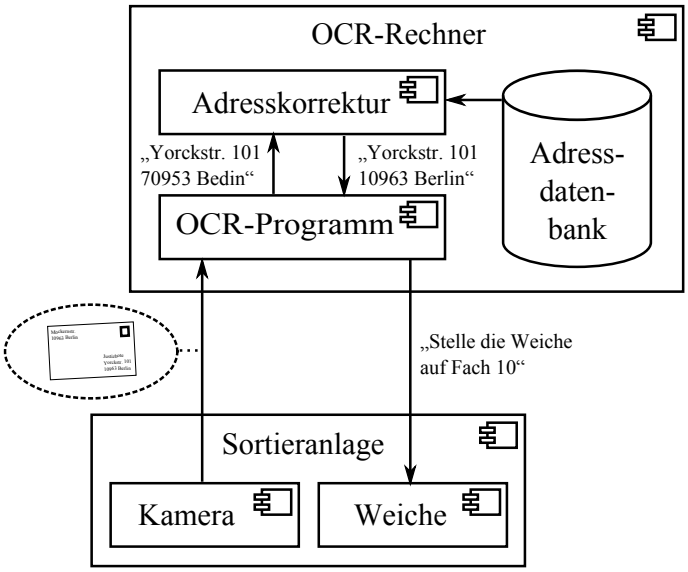

Abbildung 1.3.: Komponenten

für den Geschäftserfolg, weshalb eine korrekte Sortierung der Briefe in möglichst vielen Fällen gewährleistet sein sollte.

Ein Weg, um eine möglichst hohe Rate an korrekt sortierten Briefen zu erreichen, ist die Verbesserung der optischen Zeichenerkennung selbst. Komplett ausschalten lassen sich Fehler bei diesem Verfahren heutzutage jedoch nicht. Als Ergänzung wird deshalb eine Nachkorrektur der erkannten Adresse vorgeschlagen. Mit einer gegebenen Datenbank aller Adressen in Deutschland kann ermittelt werden, ob eine gelesene Adresse korrekt ist oder nicht. Über die Fehlererkennung hinaus ist außerdem auch eine automatische Korrektur der falsch erkannten Adressen möglich. Dabei kann die Redundanz der Informationen in den Adressbestandteilen Ort, Postleitzahl und Straße ausgenutzt werden.

1.3. Technische Anforderungen

In diesem Abschnitt werden die technischen Rahmenbedingungen erläutert. Diese umfassen die vorhandene Briefsortiermaschine, die OCR-Software und die Anforderungen an das zu erstellende Modul zur Adresskorrektur.

Die Briefsortiermaschine wurde von PostCon selbst entwickelt und beinhaltet einerseits die Hardware (Förderbänder, Motoren, Weichen, Fächer, Sensoren, Steuerungsrechner) und andererseits die Steuersoftware. Um einen hohen Durchsatz an Briefen zu erreichen, werden die Briefe möglichst schnell transportiert. Die Bandlaufgeschwindigkeit ist 2 m/s, der Abstand zwischen der Kamera und der Weiche beträgt 2,7 m. Damit bleibt für die Ermittlung der korrekten Adresse eine Zeit von 1350 ms. In dieser Zeit muss sowohl die optische Zeichenerkennung als auch die Adresskorrektur mit Hilfe der Datenbank abgeschlossen sein. Die Briefe haben einen Abstand von 40 cm, so dass jeweils die Daten von ungefähr 7 Briefen gleichzeitig berechnet werden. Durch eine Vergrößerung des Abstandes zwischen Kamera und Weiche würde mehr Zeit zur Verfügung stehen, jedoch müssten dann auch mehr Briefe gleichzeitig

verarbeitet werden.

Die OCR-Software wurde von einer externen Firma entwickelt und enthält mehrere verschiedene OCR-Engines, die in einem Abstimmungssystem (engl.: voting system) verbunden sind, um eine bessere Erkennungsrate zu erreichen. Vereinfacht besteht die optische Zeichenerkennung aus folgenden drei Phasen [Rin03]:

1. Vorverarbeitung,
2. Merkmalsextraktion und
3. Klassifikation.

Auf die Details der Zeichenerkennung wird hier nicht weiter eingegangen, weil sich diese Arbeit nicht mit der eigentlichen OCR, sondern mit der Nachkorrektur der Ergebnisse beschäftigt, die als vierte Phase angesehen werden kann. Die OCR-Software ist auf die spezifischen Gegebenheiten bei der Erkennung einer Adresse auf einem Brief angepasst. Unter anderem sind beispielsweise die zugelassenen Zeichen für die Postleitzahl nur die Ziffern von 0 bis 9.

Für jeden Brief, der die Kamera passiert, wird ein neuer Thread für die optische Zeichenerkennung gestartet, so dass die Analyse mehrerer Briefe parallel stattfindet.[4] Die optische Zeichenerkennung für einen Brief benötigt insgesamt 800–1000 ms der insgesamt zur Verfügung stehenden 1350 ms. Das Starten der Software-Prozesse und das Stellen der Weiche benötigt weitere Zeit, so dass für die Adresskorrektur eine Zeitspanne von ungefähr 100 ms verbleibt. In dieser Zeit soll die erkannte Adresse mit der Datenbank abgeglichen werden, dieser Abgleich muss also in Echtzeit geschehen. Die Datenbank der korrekten Adressen umfasst über 1 Mio Adressen (genaueres zur verwendeten Datenbank in Abschnitt 6.1). Weil Festplattenzugriffe relativ langsam sind, sollte das Modul zur Adresskorrektur inklusive Daten in den Arbeitsspeicher (2 Gigabyte) passen.

1.4. Zielsetzung

Ein Ziel der Diplomarbeit ist der Aufbau eines Systems zur OCR-Nachkorrektur von Adressen, das die Korrektur von möglichst vielen Adressen ermöglicht. Außerdem wäre es wünschenswert, wenn die entwickelten Konzepte auch auf andere allgemeinere Bereiche übertragbar wären. Das Problem der Korrektur von Adressen wird hier deshalb verallgemeinert zur Suche nach einem ähnlichen Datensatz in einer Datenbank.

In dieser Arbeit sollen daher insbesondere folgende Fragestellungen bearbeitet werden:

1. Wie kann die Ähnlichkeit von Zeichenketten definiert werden, so dass die Fehler (hier unter anderem die Fehler der optischen Zeichenerkennung) gut abgebildet werden?
2. Wie kann die Ähnlichkeit von Datensätzen definiert werden, so dass die Fehler in den einzelnen Attributen der Datensätze gut abgebildet werden?
3. Mit welcher Datenstruktur kann effizient nach ähnlichen Zeichenketten gesucht werden?
4. Mit welcher Datenstruktur kann effizient nach ähnlichen Datensätzen gesucht werden?
5. Wie kann die Ähnlichkeitsfunktion für Zeichenketten mit Hilfe von Trainingsdaten verbessert werden?

[4] Der OCR-Rechner ist mit mehreren Prozessoren ausgestattet, so dass diese Berechnungen parallelisiert ausgeführt werden können.

6. Wie kann die Ähnlichkeitsfunktion für Datensätze mit Hilfe von Trainingsdaten verbessert werden?

1.5. Zusammenfassung

In dieser Arbeit soll ein System zur Korrektur einer mittels optischer Zeichenerkennung gelesenen Adresse entwickelt werden. Dabei soll die vorhandene OCR-Software nicht verändert werden, sondern eine Komponente zur Nachkorrektur der OCR-Ergebnisse mit Hilfe einer Datenbank von vorhandenen, korrekter Adressen erstellt werden. Dem System steht zur Beantwortung einer Anfrage nur begrenzte Zeit (limitiert durch die Bandlaufgeschwindigkeit) und begrenzter Speicherplatz (limitiert durch die Größe des Arbeitsspeichers) zur Verfügung.

Das Problem der Adresskorrektur wird dabei zunächst verallgemeinert zur Suche von ähnlichen Datensätzen in einer Datenbank. Zur Lösung dieses Problems werden für Zeichenketten einerseits und Datensätze andererseits Verfahren zur Bestimmung der Ähnlichkeit, zur Indizierung und zum Erlernen der Ähnlichkeitsfunktion entwickelt und evaluiert.

Die Arbeit ist in acht Kapitel gegliedert. Kapitel 2 beinhaltet mit dem *Theoretischen Modell* eine formale Beschreibung des Problems, grundlegende Definitionen und ein Fehlermodell für die zu erwartenden Fehler bei der Adresskorrektur. In Kapitel 3 werden *Verwandte Arbeiten* aus verschiedenen Bereichen der Informatik vorgestellt. In Kapitel 4 werden *Ähnlichkeitsmaße* für Zeichenketten und Datensätze erläutert und auf die Eignung für das Problem der Adresskorrektur untersucht. In Kapitel 5 wird eine *Indexstruktur* für Zeichenketten genauer erläutert und außerdem eine Indexstruktur für Datensätze vorgeschlagen. Das System zur der Adresskorrektur wird in Kapitel 6 dargestellt. In Kapitel 7 werden die *Lernverfahren* der Ähnlichkeitsfunktion für Zeichenketten und Datensätze erläutert. Kapitel 8 beinhaltet eine *Evaluation* der Effizienz des vorgeschlagenen Verfahrens im Vergleich zu einem naiven Ansatz und außerdem eine Analyse der Qualität der Ergebnisse bei der Adresskorrektur. Zum *Schluss* wird in Kapitel 9 die Arbeit zusammengefasst und ein Ausblick auf mögliche zukünftige Arbeiten gegeben. Der Anhang A beinhaltet eine Erläuterung der verwendeten Notationen und Abkürzungen sowie die detaillierteren Ergebnisse der Evaluation.

Bemerkung Die in dieser Arbeit angeführten Beispiele stammen aus dem Bereich der Adresskorrektur, auch wenn viele der Konzepte auch auf allgemeinere Bereiche anwendbar sind.

2. Theoretisches Modell

In diesem Kapitel werden zunächst einige grundlegende Definitionen aufgeführt und die Begriffe *Ähnlichkeitsmaß* sowie *Abstandsmaß* formalisiert. Anschließend werden zwei theoretische Sichtweisen auf das Problem dargestellt. Zum einen wird das Problem als Anfrage an eine relationale Datenbank und zum anderen als Suche in einem metrischen Raum betrachtet. Beide Ansätze bieten jeweils einen eigenen Blickwinkel und somit auch mögliche Lösungsansätze für das gegebene Problem.

Der Versuch, eine fehlerhaft gelesene Adresse zu korrigieren, wird hier verallgemeinert als die Suche eines ähnlichen Datensatzes in einer Menge von Datensätzen. In jedem der folgenden Abschnitte wird zunächst theoretisch das verallgemeinerte Problem und danach die konkrete Situation bei der Suche nach ähnlichen Adressen erläutert.

2.1. Definitionen

Dieser Abschnitt beinhaltet die wichtigsten Definitionen, die in dieser Arbeit durchgängig verwendet werden. Dabei werden gebräuchliche Notationen benutzt, die in Anhang A.1 gesammelt aufgeführt sind.

2.1.1. Allgemein

- $U = A_1 \times A_2 \times \ldots \times A_m$ ist die Menge aller möglichen Datensätze.
- A_i ist die Wertemenge des i-ten Attributs.
- m ist die Anzahl der Attribute.
- $r = (a_1, a_2, \ldots, a_m) \in U$ bezeichnet einen Datensatz.
- $a_i \in A_i$ ist der Wert des i-ten Attributs.
- $D \subset U$ ist eine Datenbankrelation[1].
- $n = |D|$ ist die Anzahl der Datensätze in D.

Bei dem Problem der Suche nach ähnlichen Datensätzen ist zu einem Anfragedatensatz $q \in U$ derjenige Datensatz $r \in D$ gesucht, der am „ähnlichsten“ zu q ist. Das Konzept der Ähnlichkeit von Datensätzen wird in Abschnitt 2.2 erläutert.

[1] Weil im Folgenden ausschließlich Datenbanken betrachtet werden, die aus nur einer Relation bestehen, wird D auch *Datenbank* genannt.

2.1.2. Adresskorrektur

Das Problem der Adresskorrektur ist eine Instanz des oben genannten allgemeinen Problems: Zu einer gegebenen Adresse ist die ähnlichste korrekte Adresse gesucht.

- $U = A_1 \times A_2 \times A_3$ ist die Menge aller syntaktisch möglichen Adressdatensätze.
- $A_1 = \Sigma^*$ die Menge aller möglichen Ortsnamen.
- $A_2 = \{0, \ldots, 9\}^5$ ist die Menge der möglichen Postleitzahlen.
- $A_3 = \Sigma^*$ die Menge aller möglichen Straßennamen.
- $r = (ort, plz, str) \in U$ bezeichnet einen Adressdatensatz.
- $D \subset U$ ist die Datenbank der gegebenen Datensätze und basiert hier auf einer Datenbank der Deutschen Post, die alle in Deutschland vorhandenen Orte mit ihren Postleitzahlen und Straßen enthält. Details zur verwendeten Datenbank finden sich in Abschnitt 6.1.
- $n = |D| \approx 1{,}2\,\text{Mio}$ ist die Anzahl der Datensätze in D.

Gesucht ist also zu einer Anfrageadresse $q \in U$ diejenige Adresse $r \in D$, die am „ähnlichsten“ zu q ist. Im nächsten Abschnitt wird das Konzept der Ähnlichkeit von Datensätzen formalisiert.

2.2. Ähnlichkeits- und Abstandsmaße

2.2.1. Allgemein

Zu einer Anfrage q ist der Datensatz aus D gesucht, dessen *Ähnlichkeit* am größten beziehungsweise dessen *Abstand* am geringsten ist. In diesem Abschnitt werden die Konzepte der Ähnlichkeit und des Abstandes von Datensätzen formalisiert, und es wird erläutert, wie sie ineinander umgewandelt werden können.

Definition 2.1 (Abstandsmaß)**.** *Eine Funktion* $\delta\colon U \times U \to \mathbb{R}_0^+$ *heißt* Abstandsmaß*, wenn für alle* $x, y \in U$ *gilt:*

$$\begin{aligned} \delta(x,x) &= 0 \\ \delta(x,y) &> 0 \text{ für } x \neq y \end{aligned}$$

Ein Abstandsmaß heißt normiert*, wenn außerdem gilt:* $\delta(x,y) \leq 1$.

Definition 2.2 (Ähnlichkeitsmaß)**.** *Eine Funktion* $\sigma\colon U \times U \to \mathbb{R}_0^+$ *heißt* Ähnlichkeitsmaß*, wenn für alle* $x, y \in U$ *gilt:*

$$\begin{aligned} \sigma(x,x) &= 1 \\ \sigma(x,y) &< 1 \text{ für } x \neq y \end{aligned}$$

Ein Ähnlichkeitsmaß σ (beziehungsweise Abstandsmaß δ) ordnet zwei Datensätzen eine reelle Zahl zu, die umso größer (beziehungsweise kleiner) ist, je ähnlicher sich die Datensätze sind.

Ähnlichkeits- und Abstandsmaß sind ineinander umwandelbar. Zu einer gegebenen Ähnlichkeitsfunktion σ kann eine Distanzfunktion δ folgendermaßen konstruiert werden:

$$\delta(x, y) := 1 - \sigma(x, y)$$

Trivialerweise erfüllt δ die Eigenschaften einer Abstandsfunktion. □

Umgekehrt kann auch zu einer gegebenen Abstandsfunktion δ eine Ähnlichkeitsfunktion σ konstruiert werden:

$$\sigma(x, y) := 1 - \frac{\delta(x, y)}{\max\{\delta(v, w) \mid v, w \in U\}}$$

Trivialerweise erfüllt σ die Eigenschaften einer Ähnlichkeitsfunktion. □

Weil die beiden Maße ineinander umwandelbar sind, werden im Folgenden die Begriffe *Ähnlichkeitsmaß* und *Abstandsmaß* der Einfachheit halber gleichberechtigt verwendet.

(Ähnlichkeitsmaß und Abstandsmaß müssen nach dieser Definition beispielsweise noch nicht die Eigenschaft der Symmetrie erfüllen. Dies wird erst bei einer *Metrik* gefordert, siehe Abschnitt 2.4.)

2.2.2. Adresskorrektur

Ein Ähnlichkeitsmaß für Adressen basiert auf Ähnlichkeitsmaßen für die Attribute *plz*, *ort* und *str*. Verschiedene Alternativen für solche Ähnlichkeitsmaße bei Zeichenketten werden in Abschnitt 4.2 im Hinblick auf ihre Verwendbarkeit hin untersucht. In Abschnitt 4.3 wird erläutert, wie aus diesen attributweisen Ähnlichkeiten eine Ähnlichkeitsfunktion für Adressdatensätze konstruiert werden kann.

2.3. Datenbank

2.3.1. Allgemein

Das Problem der Suche eines ähnlichen Datensatzes kann folgendermaßen als Anfrage an eine relationale Datenbank beschrieben werden. Gegeben ist eine Datenbanktabelle D mit m Spalten A_1, A_2, ..., A_m. Gesucht sind zu einem Anfragetupel q alle Zeilen r aus D, die ähnlich zu q sind.

Algorithmus 2.1: Datenbankanfrage (Allgemein)

```
1 SELECT a_1, a_2, ..., a_m, σ(q,r) AS similarity
2 FROM D AS r
3 WHERE r.a_1 ≈_1 ⟨q.a_1⟩ AND r.a_2 ≈_2 ⟨q.a_2⟩ AND ... AND r.a_m ≈_m ⟨q.a_m⟩
4 ORDER BY similarity DESC
5 LIMIT 1
```

Dabei ist $\approx_i$ jeweils eine beliebige Ähnlichkeitsrelation. Die Ergebnisse sollen dabei absteigend nach Relevanz, also nach Ähnlichkeit der Datensätze sortiert werden.[2] Von Interesse ist hier nur der ähnlichste Datensatz.

Die WHERE-Klausel bewirkt, dass nur Datensätze in Erwägung gezogen werden, deren Attribute in gewisser Weise „ähnlich“ zu denen der Anfrage sind. Dies ist auch für die effiziente Auswertung der Anfrage sehr wichtig, weil die Ähnlichkeitsfunktion σ sonst für

[2] Dies ist verwandt mit der Situation im Information Retrieval, bei dem ebenfalls eine ungenaue Suche erwünscht ist und die Ergebnisse nach Relevanz sortiert sein sollen.

alle Datensätze berechnet werden müsste. So ergibt sich die Möglichkeit durch geeignete Indizierung der Attribute die Menge der zu überprüfenden Datensätze einzuschränken.

Die Attribute einer Datenbanktabelle sind in der Regel nicht unabhängig voneinander, sondern häufig eng korreliert. In nicht vollständig normalisierten Relationen gibt es in der Praxis oftmals sogar funktionale Abhängigkeiten zwischen Nicht-Schlüssel-Attributen. Im nächsten Abschnitt werden Korrelationen am Beispiel der Adressen genauer erläutert.

2.3.2. Adresskorrektur

Für das konkrete Problem des Auffindens der ähnlichsten Adresse hat die relationale Datenbanktabelle folgende Gestalt. Die Zeilen repräsentieren alle in Deutschland vorkommenden Straßen, zu denen neben dem Straßennamen auch der Ort und die Postleitzahl gespeichert sind. Zu einer Adresse, die mittels optischer Zeichenerkennung möglicherweise fehlerhaft gelesen wurde, soll der korrekte Eintrag in der Datenbank gefunden werden. Gegeben ist also ein Anfragetupel $q = (ort, plz, str)$, gesucht ist der ähnlichste Datensatz r in der Datenbank. Die Anfrage sieht in Pseudo-SQL damit folgendermaßen aus:

Algorithmus 2.2: Datenbankanfrage (Adressen)

```
1 SELECT plz, ort, str, σ(q,r) AS similarity
2 FROM D  AS r
3 WHERE r.plz ≈₁ ⟨q.plz⟩ AND r.ort ≈₂ ⟨q.ort⟩ AND r.str ≈₃ ⟨q.str⟩
4 ORDER BY similarity DESC
5 LIMIT 1
```

Dabei können $\approx_1$, $\approx_2$ und $\approx_3$ verschiedene Ähnlichkeitsrelationen sein. Es ist in diesem Kontext wahrscheinlich auch sinnvoll, die Ähnlichkeit von Postleitzahlen anders zu definieren als die von Orts- oder Straßennamen.

Die Attribute *ort*, *plz*, und *str* sind hier jedoch nicht unabhängig voneinander, sondern stark korreliert. In den meisten (jedoch nicht in allen) Fällen ist der Ortsname schon durch die Postleitzahl bestimmt und auch umgekehrt gibt es in einem Ort meist nur wenige verschiedene Postleitzahlen. Es gibt zwar keine echten funktionalen Abhängigkeiten, aber sehr enge Korrelationen. Statistiken zu den Korrelationen der Attribute in der gegebenen Adressdatenbank werden in Abschnitt 8.1 erläutert.

2.4. Metrischer Raum

2.4.1. Allgemein

Die Menge U aller möglichen Datensätze kann auch als Raum aufgefasst werden, in dem die Datensätze durch die Abstandsfunktion δ zueinander in Beziehung gesetzt werden. Wenn die Abstandsfunktion die Eigenschaften einer Metrik erfüllt, ist (U, δ) ein metrischer Raum.

Definition 2.3 (Metrik)**.** *Eine Funktion* $m\colon U \times U \to \mathbb{R}$ *heißt* Metrik*, wenn sie die folgenden Eigenschaften für alle Elemente* x*,* y *und* z *aus* U *erfüllt:*

$$
\begin{aligned}
&m(x,y) \geq 0 && \textit{(Nicht-Negativität)}\\
&m(x,y) = 0 \Leftrightarrow x = y && \textit{(Identität von nicht Unterscheidbaren)}\\
&m(x,y) = m(y,x) && \textit{(Symmetrie)}\\
&m(x,y) \leq m(x,z) + m(z,y) && \textit{(Dreiecksungleichung)}
\end{aligned}
$$

Ein metrischer Raum ist eine Verallgemeinerung des euklidischen Raumes. Dabei haben die Elemente nicht notwendigerweise absolute Koordinaten, sondern sind nur durch ihren relativen Abstand zueinander bestimmt (siehe Abbildung 2.1). Zu einem gegebenen Element q ist das Element $r \in D$ gesucht, für das $\delta(q, r)$ minimal ist. Diese Anfragen können mit Hilfe von generischen Indexstrukturen für metrische Räume (sog. *metrische Bäume*) relativ effizient durchgeführt werden. In Abschnitt 3.2.1 werden diese Indexstrukturen genauer erläutert. Diese Verfahren funktionieren aber nur, wenn die Abstandsfunktion in der Tat eine Metrik ist, also insbesondere die Dreiecksungleichung erfüllt.

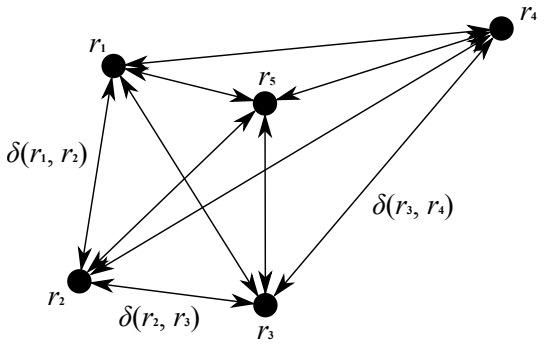

Abbildung 2.1.: Metrischer Raum

2.4.2. Adresskorrektur

Eine Abstandsfunktion δ für Datensätze kann gegebenenfalls Symmetrie und die Dreiecksungleichung erfüllen, so dass sie eine Metrik ist. In dem so gebildeten metrischen Raum (U, δ) ist zu einer Anfrageadresse q die Adresse r aus D gesucht, die den geringsten Abstand aufweist. Eine Indexstruktur in diesem metrischen Raum kann die Suche nach ähnlichen Adressen unterstützen. Auch Für die Zeichenketten-Abstandsfunktionen in Abschnitt 4.2 wird jeweils untersucht, ob es sich um eine Metrik handelt.

2.5. Fehlermodell

Zu einem Anfrage-Datensatz ist ein passender Datensatz in der Datenbank gesucht. Dabei ist keine exakte Suche möglich, weil der Datensatz möglicherweise durch verschiedene Fehler verändert wurde. In diesem Abschnitt werden diese Fehler zunächst theoretisch und für den allgemeinen Fall modelliert und anschließend für das konkrete Problem der Adresssuche untersucht. Der Begriff *Fehler* ist hier weit gefasst und beinhaltet alle Ursachen, die dazu führen, dass ein Datensatz nicht exakt in der Datenbank gefunden werden kann. Als Fehler zählen damit beispielsweise auch ungenaue Messwerte oder Abkürzungen von Wörtern, welche eigentlich keine Fehler im engeren Sinne darstellen.

Fehlerursachen bei numerischen Werten sind beispielsweise Mess- und Rundungsungenauigkeiten oder Schätzungen von Werten. Zeichenketten können von Rechtschreib- und Tippfehlern betroffen sein. Weitere Möglichkeiten sind das Fehlen von Werten (falls z. B. keine Angabe gemacht wurde), Bitfehler bei der Übertragung sowie Fehler der optischen Zeichenerkennung.

Die Formalisierung der Fehler ist eine wichtige Voraussetzung, um die Fehler erkennen und korrigieren zu können. Ein Ähnlichkeitsmaß ist gut geeignet, wenn es zwei Datensätze beziehungsweise Attributwerte in solchen Fällen als ähnlich bewertet, in denen der eine Wert aus dem anderen durch Fehler entstanden ist.

Bei einigen Fehlerarten kann eine Vorhersage darüber getroffen werden, wie hoch die Wahrscheinlichkeit eines bestimmten Fehlers ist. Bei Bit-Übertragungsfehlern kann beispielsweise in der Regel die Störung des Kanals quantifiziert werden, wodurch eine Aussage über die Wahrscheinlichkeit des Kippens eines Bits möglich ist. Einige OCR-Systeme sind in der Lage einen Konfidenzwert anzugeben, der für Zeichenketten oder Buchstabe eine Aussage über die Sicherheit der korrekten Erkennung macht. Auch bei Messfehlern kann in der Regel ungefähr abschätzt werden, wie groß der erwartete Fehler ist. Dieses Maß über die Unsicherheit eines Attributes kann in der Ähnlichkeitsfunktion für Datensätze ausnutzt werden, indem verschiedene Gewichtungen für die Attribute verwendet werden.

2.5.1. Allgemein

Die Fehler können sich im Allgemeinen sowohl auf die Datenbank erstrecken (z. B. ungenaue Messwerte, mittels OCR fehlerbehaftet eingelesene Dokumente), als auch auf die Anfrage (z. B. Rechtschreibfehler). Die Fehler werden hier mit Hilfe von Wahrscheinlichkeiten modelliert. $\mathrm{P}(q \mid r)$ bezeichnet die Wahrscheinlichkeit, dass der fehlerhafte Datensatz q anstatt r auftritt. Diese Wahrscheinlichkeit basiert auf Fehlerwahrscheinlichkeiten der Attribute. $\mathrm{P}(q.a_i \mid r.a_i)$ ist die Wahrscheinlichkeit, dass ein Fehler das Attribut vom Wert $r.a_i$ in den Wert $q.a_i$ verändert.

Bei numerischen Daten kann diese Wahrscheinlichkeit für die Veränderung eines Attributwertes beispielsweise durch eine Normalverteilung gegeben sein, wenn die Werte durch Messungenauigkeiten entstanden sind. Bei Zeichenketten mit Tippfehlern kann der Wert aus empirisch gewonnenen Wahrscheinlichkeiten für das Tippen einer falschen Taste bestimmt werden.

Die Wahrscheinlichkeit, dass ein Datensatz q statt einem anderen Datensatz r auftritt ist:

$$\begin{aligned} &\mathrm{P}\left(q \mid r\right) \\ =&\mathrm{P}\left((q.a_1, \ldots, q.a_m) \mid (r.a_1, \ldots, r.a_m)\right) \\ =&f\left(\mathrm{P}\left(q.a_1 \mid r.a_1\right), \ldots, \mathrm{P}\left(q.a_m \mid r.a_m\right)\right) \end{aligned}$$

Falls vorausgesetzt werden kann, dass die Fehler der Attribute unabhängig voneinander sind, ist f das Produkt der einzelnen Werte. Diese Unabhängigkeit gilt aber beispielsweise nicht, wenn sich ein Fehler auch auf mehrere Attribute erstrecken kann, wie es bei OCR-Fehlern der Fall ist.

2.5.2. Adresskorrektur

In diesem Abschnitt werden die Fehler beim konkreten Problem der Sortierung von Briefen untersucht. Hier sind die Daten in der Datenbank korrekt und nur der Anfragedatensatz enthält möglicherweise Fehler. Es gibt mehrere verschiedenartige Gründe, warum die vom Brief extrahierte Adresse so nicht in der Adressdatenbank enthalten ist. Einerseits verwenden die Absender nicht immer den im postalischen Sinne korrekten Orts- und Straßennamen sondern davon abweichende, alternative Bezeichnungen. Außerdem können ihnen Rechtschreib- und Tippfehler beim Schreiben der Adresse unterlaufen. Des Weiteren wird die aufgedruckte Adresse möglicherweise von der optischen Zeichenerkennung fehlerhaft gelesen. All diese

Fehler können und werden natürlich auch kombiniert auftreten, jedoch ist auch in diesem Fall ein erfolgreicher Abgleich mit der Adressdatenbank wünschenswert. Im Folgenden werden die möglichen Arten von Fehlern genauer betrachtet und anschließend in zwei Kategorien eingeteilt.

Für jede Fehlerart werden exemplarisch einige fehlerhafte Ausprägungen der Attribute angegeben, die in der Praxis aller Voraussicht nach eine Wahrscheinlichkeit > 0 haben. Die Wahrscheinlichkeit $\mathrm{P}_{\mathrm{Synonym},ort}(\text{„Dahlem“}, \text{„Berlin“})$ bezeichnet beispielsweise die Wahrscheinlichkeit, dass das Attribut *ort* durch Verwendung eines Synonyms von „Berlin“ in „Dahlem“ geändert wird. Alle drei Attribute *plz*, *ort* und *str* werden hier als Zeichenketten behandelt.[3]

Synonyme

Sehr häufig werden beim Versenden von Briefen alternative Bezeichnungen (Synonyme) für Orte und Straßen verwendet, die nicht mit den im postalischen Sinne korrekten Namen übereinstimmen. Bei Orten werden beispielsweise veraltete Ortsnamen, Namen von Ortsteilen oder auch Ortszusätze verwendet, z. B. „Dahlem“ statt „Berlin“ und „Hansestadt Hamburg“ statt „Hamburg“.

$$\begin{aligned}&\mathrm{P}_{\mathrm{Synonym},ort}(\text{„Dahlem“} \mid \text{„Berlin“}) > 0\\&\mathrm{P}_{\mathrm{Synonym},ort}(\text{„Hansestadt Hamburg“} \mid \text{„Hamburg“}) > 0\\&\qquad\vdots\end{aligned}$$

Auch bei Straßen werden in einigen Fällen Synonyme verwendet, wie „Ku’damm“ an Stelle von „Kurfürstendamm“. Des Weiteren wird sehr häufig bei der Getrennt- und Zusammenschreibung variiert, so wird beispielsweise „Berlinerstr.“ statt „Berliner Str.“ geschrieben und umgekehrt.

$$\begin{aligned}&\mathrm{P}_{\mathrm{Synonym},str}(\text{„Ku’damm“} \mid \text{„Kurfürstendamm“}) > 0\\&\mathrm{P}_{\mathrm{Synonym},str}(\text{„Berlinerstr.“} \mid \text{„Berliner Str.“}) > 0\\&\qquad\vdots\end{aligned}$$

Abkürzungen

Eine weitere ähnliche Art von Abweichungen in der Schreibweise von Orts- und Straßennamen sind Abkürzungen. Statt „Frankfurt am Main“ wird beispielsweise oftmals „Frankfurt a.M.“ geschrieben. Auch der umgekehrte Fall kommt vor, so dass offiziell abgekürzte Ortsnamen wie „Hann. Münden“ auf dem Brief dann ausgeschrieben werden als „Hannoversch Münden“.

$$\begin{aligned}&\mathrm{P}_{\mathrm{Abk},ort}(\text{„Frankfurt a.M.“} \mid \text{„Frankfurt am Main“}) > 0\\&\mathrm{P}_{\mathrm{Abk},ort}(\text{„Hannoversch Münden“} \mid \text{„Hann. Münden“}) > 0\\&\qquad\vdots\end{aligned}$$

[3] Auch die Postleit*zahl* ist eine Zeichenkette über dem Alphabet $\{0, \ldots, 9\}$.

Bei Straßennamen sind Abkürzungen noch verbreiteter, so wird beispielsweise „Prenzl. Prom.“ statt „Prenzlauer Promenade“ und „Bramfelder Ch.“ statt „Bramfelder Chaussee“ geschrieben.

$$\begin{aligned} &\mathrm{P}_{\mathrm{Abk},str}(\text{„Prenzl. Prom.“} \mid \text{„Prenzlauer Promenade“}) > 0 \\ &\mathrm{P}_{\mathrm{Abk},str}(\text{„Bramfelder Ch.“} \mid \text{„Bramfelder Chaussee“}) > 0 \\ &\qquad\vdots \end{aligned}$$

Rechtschreibfehler

Außerdem können dem Absender natürlich Rechtschreibfehler unterlaufen sein, die sich meist auf ein oder zwei aufeinander folgende Buchstaben erstrecken. Hierbei werden einzelne Buchstaben weggelassen, eingefügt, durch andere ersetzt, oder Buchstaben werden miteinander vertauscht (Beispiel: „Stuttgard“ statt „Stuttgart“).

$$\begin{aligned} &\mathrm{P}_{\mathrm{RS},ort}(\text{„Stuttgard“} \mid \text{„Stuttgart“}) > 0 \\ &\mathrm{P}_{\mathrm{RS},str}(\text{„Schlossalle“} \mid \text{„Schloßallee“}) > 0 \\ &\mathrm{P}_{\mathrm{RS},str}(\text{„Potsdammer Platz“} \mid \text{„Potsdamer Platz“}) > 0 \\ &\qquad\vdots \end{aligned}$$

Tippfehler

Tippfehler wirken sich in sehr ähnlicher Weise auf den Text aus wie Rechtschreibfehler. Beim Tippen können ebenfalls Buchstaben weggelassen, eingefügt, ersetzt und vertauscht werden. Ersetzungen geschehen dabei meist durch auf der Tastatur eng beieinander stehende Buchstaben (beispielsweise „v“ statt „b“).

$$\begin{aligned} &\mathrm{P}_{\mathrm{Tipp},ort}(\text{„Belrin“} \mid \text{„Berlin“}) > 0 \\ &\mathrm{P}_{\mathrm{Tipp},str}(\text{„Hauiptstraße“} \mid \text{„Hauptstraße“}) > 0 \\ &\mathrm{P}_{\mathrm{Tipp},plz}(\text{„10693“} \mid \text{„10963“}) > 0 \\ &\mathrm{P}_{\mathrm{Tipp},ort}(\text{„Bad Kleinsu“} \mid \text{„Bad Kleinau“}) > 0 \\ &\qquad\vdots \end{aligned}$$

OCR-Fehler

Die hier am häufigsten zu erwartende Fehlerart sind die Fehler, die bei der optischen Zeichenerkennung geschehen. Zu den möglichen Ursachen gehören schlechte Druckqualität, Reflexion des Sichtfensters, Verdreckung des Briefes oder unscharfe Einstellung der Kamera. Diese Ursachen mindern die Bildqualität und können dadurch das korrekte Erkennen des Textes verhindern. Sie betreffen oftmals mehrere aufeinander folgende Buchstaben (engl.: burst error) oder gar mehrere Attribute (beispielsweise *ort* und *str*). Aber auch bei sehr guter Bildqualität kommt es durch die Ähnlichkeit verschiedener Zeichen oder Zeichenkombinationen häufig zu Fehlern. Diese Fehler hängen von der Klassifikationsmethode der OCR-Software

ab und unterscheiden sich von System zu System.[4] Weil OCR-Fehler häufig auftreten und kaum vollständig zu vermeiden sind, sollten sie möglichst gut modelliert und korrigiert werden. Bei der Erkennung des Briefes in Abbildung 6.2 sind beispielsweise folgende Fehler aufgetreten:

$$
\begin{aligned}
&P_{\mathrm{OCR},str}(\text{„Kaisedn-Augusta-A}\sim\text{ee“} \mid \text{„Kaiserin-Augusta-Allee“}) > 0 \\
&P_{\mathrm{OCR},plz}(\text{„10663“} \mid \text{„10553“}) > 0 \\
&P_{\mathrm{OCR},ort}(\text{„Bedin“} \mid \text{„Berlin“}) > 0 \\
&\quad\vdots
\end{aligned}
$$

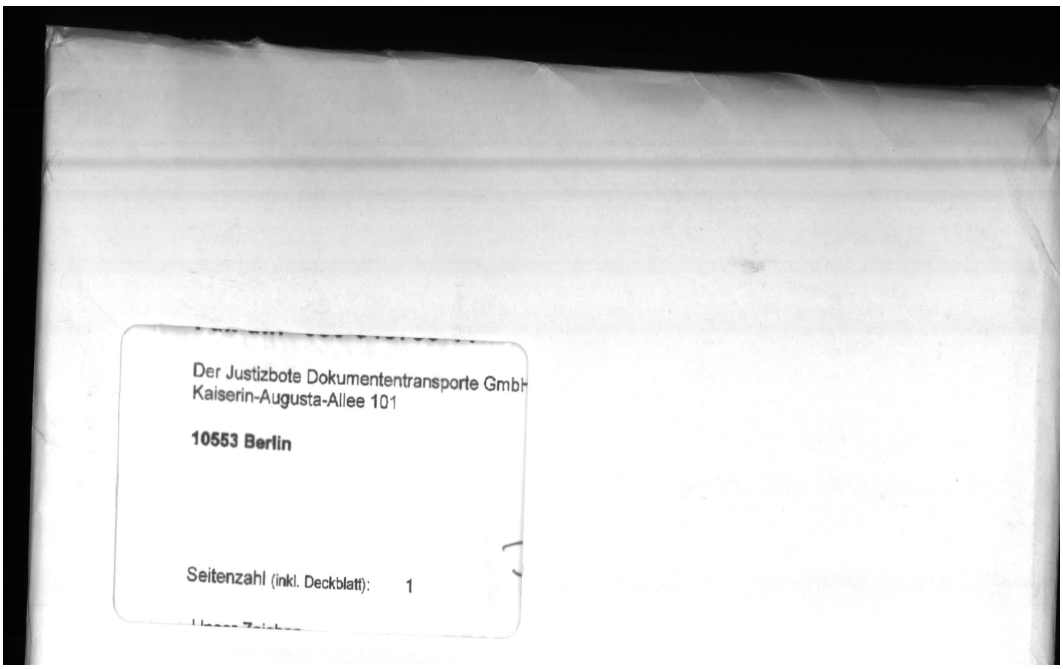

Abbildung 2.2.: Brief mit OCR-Fehlern (erkannte Adresse: „Der Jusübote Dokumententransporte Gmb∼, Kaisedn-Augusta-A∼ee 101, 10663 Bedin“)

Die Auswirkungen können im Allgemeinen in folgende Kategorien eingeteilt werden [Rin03]:

- Substitution: Ein Zeichen wird durch ein anderes ersetzt („c“ → „o“).
- Einfügung: Ein Zeichen wird eingefügt (ϵ → „i“).
- Löschung: Ein Zeichen wird gelöscht („l“ → ϵ).
- Aufteilung: Ein Zeichen wird in mehrere aufgeteilt („n“ → „ii“).
- Verschmelzung: Mehrere Zeichen werden zu einem Zeichen zusammengefasst („rn“ → „m“).

[4] Deshalb werden teilweise mehrere OCR-Systeme gleichzeitig eingesetzt, die für zweifelhafte Buchstaben eine Abstimmung durchführen (engl.: voting system).

- Versagen: Die OCR-Software erkennt, dass an dieser Stelle ein Zeichen stehen müsste, aber es gibt kein Zeichen mit genügend hoher Erkennungswahrscheinlichkeit. Es wird daher ein Platzhalter ausgegeben (beispielsweise „~“).

Diesen einzelnen Fehlern können ebenfalls Wahrscheinlichkeiten zugewiesen werden, die beispielsweise mit Hilfe von Trainingsdaten erlernt werden können. Die Wahrscheinlichkeit für die Überführung einer ganze Zeichenkette in eine andere Zeichenkette kann aus den Wahrscheinlichkeiten der einzelnen Fehler abgeschätzt werden. Dieses Verfahren wird in Abschnitt 7.1 erläutert.

Einige OCR-Systeme sind in der Lage, einen Wert zur Verfügung zu stellen, der angibt, mit welcher Sicherheit jeder einzelne Buchstabe korrekt gelesen wurde. Dieser sogenannte Konfidenzwert kann in der Ähnlichkeitsfunktion eingesetzt werden: Wenn zwei Zeichenketten sich in einem Buchstaben unterscheiden, dieser aber nur mit einer sehr geringen Sicherheit gelesen wurde, sollten die Zeichenketten dennoch als ähnlich gelten.

Eine Evaluation der OCR-Fehler bezüglich einer Testmenge von Briefen findet sich in Abschnitt 8.3.

Weitere Fehlerquellen

Es gibt noch eine Reihe weiterer möglicher Fehlerquellen, auf die hier jedoch nicht weiter eingegangen wird, weil sie in einem anderem Verfahrensschritt des Sortierprozesses behandelt werden sollten. Ein Brief kann beispielsweise falsch herum aufgelegt worden sein, so dass die OCR-Kamera nur die Rückseite des Briefes aufnehmen kann. Außerdem kann das Papier innerhalb des Briefumschlages verrutscht sein, so dass das Adressfeld nicht (vollständig) im Sichtfenster zu sehen ist. Unter diesen Umständen kann die optische Zeichenerkennung keine sinnvolle Adresse extrahieren und somit verspricht auch ein Abgleich mit der Adressdatenbank keinen Erfolg.

Eine weitere Möglichkeit für einen Fehler ist, dass die OCR-Software das falsche Textfeld als Empfänger des Briefes erkennt. Auf Briefen sind normalerweise neben dem Empfänger auch noch ein Textfeld mit der Adresse des Absenders und gegebenenfalls Werbetexte aufgedruckt (siehe Abbildung 2.3). Wenn die OCR-Software fälschlicherweise das Absender-Textfeld als Empfänger identifiziert, wird ein Abgleich mit der Adressdatenbank in der Regel die Korrektheit der Adresse bestätigen. Dadurch würde der Brief dann wieder an den Absender zurück geschickt werden, was natürlich nicht beabsichtigt ist. Dieses Problem kann vermieden werden, wenn im Voraus bekannt ist, wer der Absender ist. Dann könnte diese Information zur Auswahl des richtigen Empfänger-Textfeldes herangezogen werden.

Wenn ein Werbe-Textfeld fälschlicherweise mit dem Empfänger-Textfeld verwechselt wird, kann der Abgleich mit der Adressdatenbank in der Regel feststellen, dass es sich nicht um eine korrekte Adresse handelt. Die OCR-Software kann mit diesem Wissen erneut versuchen das richtige Empfänger-Textfeld zu lesen und dabei das irrtümlich gelesene Werbe-Textfeld bei der Suche ausschließen.

Zusammenfassung

Die erkannte Zeichenkette der Attribute *ort* und *str* ergibt sich aus der eigentlich korrekten Zeichenkette durch die sequentielle Anwendung der Fehlerarten alternative Schreibweise, Abkürzungen, Rechtschreibfehler, Tippfehler und OCR-Fehler. Das Attribut *plz* ist nur von OCR- und Tippfehlern betroffen, weil keine Synonyme oder Abkürzungen für Postleitzahlen existieren.

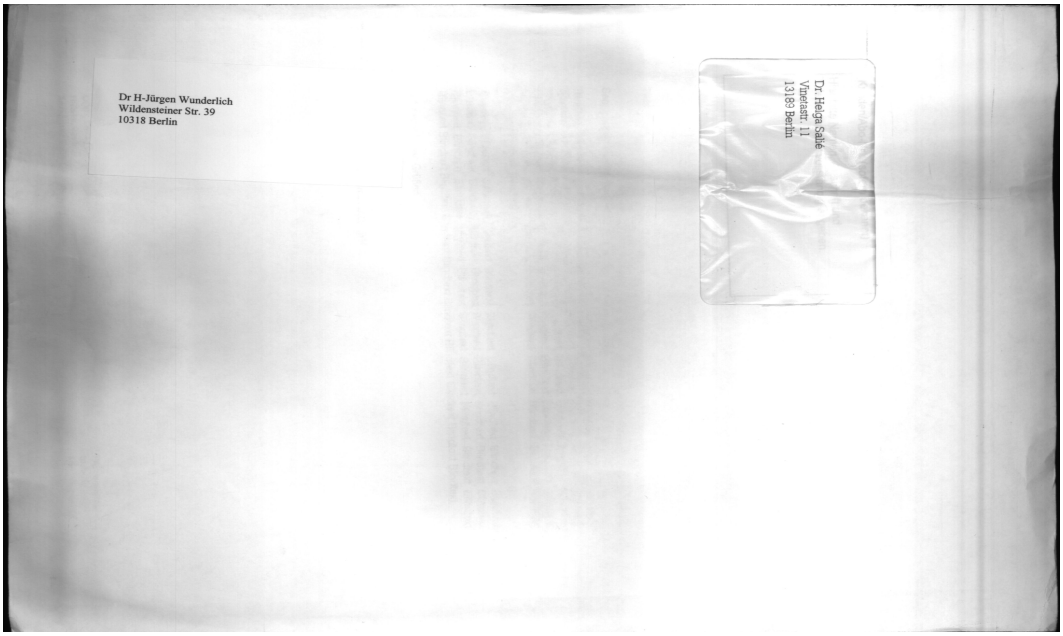

Abbildung 2.3.: Brief mit mehreren Textfeldern

Bei der letztendlich erkannten Zeichenkette kann durch die Überlagerung der Fehler nicht mehr unterschieden werden, welche Ursache z. B. die Ersetzung eines Zeichens durch ein anderes Zeichen hatte (dies kann unter anderem durch Rechtschreib-, Tipp- oder OCR-Fehler passiert sein). Die genaue Ursache zu wissen ist jedoch hier und auch in den meisten anderen Anwendungen nicht nötig [Kuk92].

Man kann die oben genannten Fehler grob in zwei Kategorien aufteilen (siehe Tabelle 2.1). Die Auswirkungen von Rechtschreib-, Tipp-, und OCR-Fehlern sind sehr ähnlich, sie betreffen immer nur ein oder wenige aufeinander folgende Zeichen. Diese Fehler werden im Folgenden als *syntaktische Fehler* bezeichnet. Die zweite Kategorie umfasst die Fehler, die durch alternative Schreibweisen entstehen. Hierbei werden unter vor allem ganze Wörter durch andere ersetzt. Für jeweils einen Ort beziehungsweise für eine Straße gibt es jedoch im Allgemeinen nur relativ wenige alternative Bezeichnungen. Diese Fehler werden hier als *Synonyme* bezeichnet.

	Syntaktische Fehler	Synonyme
Ursache	Rechtschreib-, Tipp- und OCR-Fehler	alternative Bezeichnungen
Betroffen	ein oder wenige Zeichen	ganze Wörter
Anzahl	viele mögliche fehlerhafte Schreibweisen für ein Wort	begrenzte Anzahl alternativer Schreibweisen

Tabelle 2.1.: Fehlerkategorien

Abkürzungen gehören eher in die Kategorie der syntaktischen Fehler, wobei sie sich

allerdings auch auf viele aufeinander folgende Buchstaben erstrecken können (beispielsweise „Ch.“ statt „Chaussee“).

Die Fehler in den beiden Kategorien sind von sehr unterschiedlicher Natur und erfordern deshalb auch verschiedene Maßnahmen, um sie zu erkennen und zu korrigieren. Syntaktische Fehler können mit den in Kapitel 4 definierten Ähnlichkeitsfunktionen für Zeichenketten behandelt werden, wobei sich für die Synonyme eine explizite, möglichst vollständige Speicherung aller alternativer Schreibweisen anbietet.

2.6. Zusammenfassung

In diesem Kapitel wurden einige theoretische Grundlagen des Problems der Adresskorrektur beziehungsweise der Suche von ähnlichen Datensätzen erläutert. Unter anderem wurden die Begriffe Ähnlichkeits- und Abstandsmaß definiert und das Problem als unscharfe Datenbankanfrage sowie als Suche in einem metrischen Raum betrachtet. Außerdem wurden die bei der Adresskorrektur zu behandelnden Fehler erläutert und in die Kategorien *syntaktische Fehler* und *Synonyme* eingeordnet.

3. Verwandte Arbeiten

In diesem Kapitel werden einige verwandte Arbeiten vorgestellt. Insbesondere wurden solche Arbeiten ausgewählt, von denen bestimmte Konzepte und Ideen auf das hier vorliegende Problem übertragen werden können. Dabei sollen vor allem Ideen für die Beantwortung der in Abschnitt 1.4 genannten sechs Fragen gefunden werden. Die hier vorgestellten Arbeiten stammen dabei aus unterschiedlichen Forschungsbereichen, die jeweils bei verschiedenen Teilproblemen der Suche nach ähnlichen Adressen relevant sind.

Zunächst wird der Stand der Technik in verwandten Anwendungsbereichen beschrieben. Im Anschluss daran werden Indexstrukturen, die in mehreren dieser Bereiche Verwendung finden, detaillierter und mit dem aktuellen Forschungsstand dargestellt.

3.1. Verwandte Anwendungsbereiche

In diesem Abschnitt werden vier Anwendungsbereiche erläutert, die ähnliche Problemstellungen wie bei der Suche von ähnlichen Adressen haben:

1. Bei der *Datensatzverknüpfung* geht es um das Finden von doppelten Datensätzen.
2. Bei einer Anfrage an eine relationale *Datenbank mit Unsicherheit* werden keine exakten, sondern unsichere Prädikate verwendet, z. B. basierend auf Wahrscheinlichkeiten.
3. Bei der *Zeichenketten-Suche* sollen Zeichenketten in einem Dokument gefunden werden.
4. Bei der *Rechtschreibkorrektur* soll zu einer möglicherweise falsch geschriebenen Zeichenkette das richtige Wort in einem Wörterbuch ermittelt werden.

Zu jedem dieser Anwendungsbereiche wird zunächst das Problem erläutert, dann der Stand der Technik dargestellt und abschließend die Übertragbarkeit der Konzepte auf das hier vorliegende Problem diskutiert.

3.1.1. Datensatzverknüpfung (Record Linkage)

Problem

Das Ziel der Datensatzverknüpfung ist das Aufspüren von Duplikaten in einer oder mehreren Dateien beziehungsweise Datenbanken.[1] Duplikate sind Datensätze, die sich zwar unterscheiden, aber auf das gleiche Objekt in der realen Welt beziehen. Dieses Problem wurde zuerst bei der Zusammenführung von Datensätzen in Krankenhäusern der USA systematisch untersucht [Dun46]. Es tritt außerdem bei Volkszählungen auf, die mit vergangenen Erhebungen abgeglichen werden sollen. Eine weiterer Anwendungsbereich ist das Filtern von Duplikaten, wenn Kundendatensätze mehrerer Firmen zusammengeführt werden, z. B. bei einer Fusion oder beim Einkaufen von Adressdaten zu Werbezwecken [Sch06].

[1] Dieses Problem tritt in verschiedenen Kontexten auf und hat verschiedene Bezeichnungen. Im Englischen sind dies beispielsweise: record linkage, the merge/purge problem, duplicate detection, hardening soft databases und reference matching. [BM02]

Formal kann das Problem folgendermaßen beschrieben werden: Gegeben ist eine Menge D von Datensätzen. Gesucht ist zu jedem Paar $(r, q) \in D \times D$ eine Aussage, ob die Datensätze zusammen passen, nicht zusammen passen oder möglicherweise zusammen passen. Im dritten Fall muss dann im Nachhinein eine manuelle Einschätzung erfolgen.

Die Schwierigkeiten bei der Datensatzverknüpfung sind zum einen die in der Praxis vorkommenden heterogenen Strukturen von Datensätzen. Das beinhaltet beispielsweise das Datenmodell (relationale Datenbank, XML, CSV,...) aber auch den Aufbau eines einzelnen Datensatzes (z. B. Speicherung der Adresse in einem Textfeld oder in vier Textfeldern: Ort, Postleitzahl, Straße, Hausnummer). Diese Probleme lassen sich jedoch technisch ohne größere Schwierigkeiten lösen. Interessanter ist das Problem, wie algorithmisch bestimmt werden kann, ob sich zwei Datensätze auf das gleiche Objekt beziehen oder nicht. Dies wird dadurch erschwert, dass die Datensätze beispielsweise mit Rechtschreibfehlern behaftet sein können, sich der Name einer Person geändert haben kann, Wörter abgekürzt sind („Prenzlauer Promenade“ ↔ „Prenzl. Prom.“) und so weiter.

Dieses Problem weist eine große Ähnlichkeit zum hier gegebenen Problem der OCR-Nachkorrektur von Adressen auf, weil ebenfalls eine Menge von Datensätzen gegeben ist und zu einem Anfragedatensatz ähnliche Einträge in einer Datenbank gefunden werden sollen. Im folgenden Abschnitt werden einige Lösungsansätze vorgestellt und anschließend die Übertragbarkeit der Ideen auf das vorliegende Problem der Suche von ähnlichen Adressen diskutiert.

Stand der Technik

Die wissenschaftliche Untersuchung der Datensatzverknüpfung begann 1959 mit Newcombe [NKAJ59], zehn Jahre später entwickelten Fellegi und Sunter [FS69] ein theoretisches Modell, das noch bis heute eine wichtige Grundlage in diesem Gebiet darstellt. Einen relativ aktuellen Überblick liefert Winkler [Win99], in dem auch viele offene Forschungsfragen dargestellt werden. Bei der Datensatzverknüpfung wurden vor allem zwei Ansätze verfolgt, die regelbasierte Datensatzverknüpfung und die probabilistische Datensatzverknüpfung.

Bei der *regelbasierten Datensatzverknüpfung* entwickelt ein Domänenexperte eine Menge von Regeln, die bestimmen, wann zwei Datensätze gleich sind. Diese Regeln werden immer weiter verfeinert, um auch Ausnahmen richtig zu behandeln. Ein Verfahren, um zu bestimmen, ob zwei Datensätze Duplikate sind, ist die Verwendung von Entscheidungsbäumen [CCGK07]. Zu einem gegebenen Paar von Datensätzen kann dann mit Hilfe dieses Baumes entschieden werden, ob es sich um Duplikate handelt. Dabei werden die Operatoren des Entscheidungsbaum mit Hilfe einer Trainingsmenge von Daten erlernt, müssen aber in der Regel von einem Experten noch verfeinert werden.

Bei der *probabilistischen Datensatzverknüpfung* wird die Aussage, ob zwei Datensätze das gleiche Objekt repräsentieren, auf Basis einer Ähnlichkeitsfunktion getroffen. Wenn die Ähnlichkeit einen gewissen oberen Schwellwert überschreitet, gelten sie als Duplikate. Wenn die Ähnlichkeit einen anderen unteren Schwellwert unterschreitet, gelten sie als unterschiedlich. Liegt die berechnete Ähnlichkeit zwischen beiden Schwellwerten, muss eine manuelle Einordnung erfolgen. Die Ähnlichkeitsfunktion wird dabei mit einer Trainingsmenge von Datensätzen erlernt, die jeweils als *zusammen passend* oder *nicht passend* markiert sind.

Die Ähnlichkeitsfunktion für Datensätze stützt sich auf Ähnlichkeitsfunktionen für die einzelnen Attribute. Die Ähnlichkeit von Attributwerten wird in der Regel auf Basis der syntaktischen Ähnlichkeit der Zeichenketten bestimmt. Welches syntaktische Ähnlichkeitsmaß zum Einsatz kommt, hängt dabei von den erwarteten Fehlern (beispielsweise Rechtschreib- oder OCR-Fehler) und dem konkreten Attribut (beispielsweise Vorname oder Ortsname)

ab. Speziell für die Datensatzverknüpfung entwickelt wurde das Jaro-Maß [Jar95], welches in [PW97] zusammen mit zwei Weiterentwicklungen erläutert wird. Außerdem wird von verschiedenen Autoren eine Editierdistanz mit erlernbaren Gewichten vorgeschlagen [BM03; CRF03; Yan04] (siehe auch Abschnitt 4.2.4, und Abschnitt 7.1). Eine genauere Betrachtung der Ähnlichkeitsmaße mit den jeweiligen Vor- und Nachteilen erfolgt in Kapitel 4.

Die Ähnlichkeit von Datensätzen wird aus einer Kombination der Ähnlichkeitswerte der Attribute berechnet. In [BM02] und [BM03] wird ein Verfahren vorgestellt, wie die Ähnlichkeitsfunktion für Datensätze mit Hilfe einer Trainingsmenge erlernt werden kann. Dabei werden die Attribut-Ähnlichkeiten eines Datensatzpaares als Vektor von reellen Zahlen betrachtet. Die Vektoren der gekennzeichneten Trainingsmenge werden von einer Support Vector Machine (SVM) verwendet, um die Klassifikation der Datensätze in *passend* und *nicht passend* zu erlernen (siehe dazu auch Abschnitt 7.2).

Alle Paare von Datensätzen zu überprüfen, wäre mit einem sehr großen Rechenaufwand verbunden. Deshalb besteht das allgemeine Verfahren bei der Datensatzverknüpfung aus zwei Schritten:

1. Partitionieren der Menge der Datensätze in Gruppen

2. Bestimmen der paarweisen Ähnlichkeit innerhalb einer Gruppe

Die Aufteilung erfolgt mittels leicht überprüfbarer Merkmale wie beispielsweise Wohnort und Geschlecht. Innerhalb jeder Gruppe wird dann für alle Paare von Datensätzen geprüft, ob es sich um Duplikate handelt.

In einigen Arbeiten wird die Suche nach Duplikaten auch als Suche in einem metrischer Raum formuliert [Sch06]. Das ist möglich, wenn die Abstandsfunktion von Datensätzen eine Metrik ist. Dieses Verfahren bietet den Vorteil, dass schon einige verschiedene Indexstrukturen für metrische Räume existieren. Diese sogenannten metrischen Bäume werden auch bei der approximativen Zeichenketten-Suche eingesetzt und in Abschnitt 3.2.1 detaillierter betrachtet.

In Verbindung mit Forschung zur Datensatzverknüpfung [CRF03] ist auch SecondString entstanden, ein System in der Programmiersprache Java, das die Implementierung und Evaluation von Datensatz- und Zeichenketten-Ähnlichkeitsmaßen erleichtern soll [Coh06].

Übertragbarkeit

Das Problem der Datensatzverknüpfung weist einige Parallelen zur OCR-Nachkorrektur von Adressen auf. So ist ebenfalls zu einem Datensatz ein ähnlicher Datensatz gesucht. Gut verwenden lassen sich beispielsweise die untersuchten Ähnlichkeitsmaße und Lernverfahren für Zeichenketten (Editierdistanz, Jaro-Maß) und Datensätze (Support Vector Machine). Entscheidungsbäume eignen sich hingegen nicht, um die Ähnlichkeit von Datensätzen zu bestimmen, weil sie nur eine binäre Klassifikation ermöglichen.

Bei der Datensatzverknüpfung spielt die Geschwindigkeit nur eine untergeordnete Rolle, weil zur Bestimmung der Duplikate einer Menge in der Regel einige Stunden gerechnet werden kann. Außerdem kann in Einzelfällen eine manuelle Nachbearbeitung erfolgen. Bei der Suche nach der ähnlichsten Adresse ist im laufenden Betrieb kein manueller Eingriff möglich und die Rechenzeit ist für eine Anfrage auf den Bruchteil einer Sekunde begrenzt.

3.1.2. Datenbank mit Unsicherheit

Problem

Wie in Abschnitt 2.3 erörtert, lässt sich das Problem des Auffindens ähnlicher Datensätze als ungenaue Anfrage an eine relationale Datenbank formulieren. Dabei werden auch Prädikate zugelassen, die nicht *wahr* oder *falsch* sind, sondern mit einer gewissen Unsicherheit verbunden sind. Diese Abschwächung von Anfragen wird auch als *Query Relaxation* bezeichnet [LWY07] und kann in verschiedenen Bereichen zur Anwendung kommen. Einerseits können die Daten in der Datenbank mit einer Unsicherheit behaftet sein, die beispielsweise durch Messfehler entstanden ist. Andererseits verfügt möglicherweise der Benutzer nur über unvollständiges Wissen und möchte deshalb eine unscharfe Anfrage formulieren.

Formal kann das Problem folgendermaßen beschrieben werden: Gegeben ist eine Datenbank D und eine Anfrage q, die unscharfe Prädikate enthält. Gesucht sind die Ergebnistupel der Anfrage, sortiert nach der Ähnlichkeit bezüglich der Prädikate.

Stand der Technik

Zum Einsatz kommen unter anderem Verfahren, die basierend auf den unscharfen Prädikaten eine Wahrscheinlichkeit für jeden Datensatz berechnen. Die Anfrage wird dann auf einer probabilistischen Datenbank ausgewertet und die Wahrscheinlichkeit der Ergebnistupel in den relationalen Operatoren berechnet [DS07]. Die Wahrscheinlichkeit eines Ergebnistupels repräsentiert damit die Wahrscheinlichkeit, dass es auf die gegebene unscharfe Anfrage passt.

Übertragbarkeit

Bei der Beantwortung unscharfer Anfragen wird versucht die Anfrageausführung möglichst effizient durchzuführen. Weil das Problem jedoch sehr allgemein formuliert ist, sind auch die Lösungen sehr allgemein und nicht notwendigerweise besonders effizient. Das in dieser Arbeit gegebene Problem der Suche von ähnlichen Datensätzen ist ein Spezialfall, bei dem nur eine Relation beteiligt ist, so dass für diesen Fall optimierte Verfahren wahrscheinlich effizienter sind. Bei der Adresskorrektur wird überdies Echtzeitfähigkeit gefordert, die von den Verfahren zur Beantwortung einer Anfrage auf einer Datenbank mit Unsicherheit nicht garantiert werden kann.

3.1.3. Zeichenketten-Suche (String Matching)

Problem

Ziel der Zeichenketten-Suche (engl.: string matching) ist es, Vorkommen einer Zeichenkette in einem Text zu finden. Bei der *approximativen* Zeichenketten-Suche wird bei der Suche eine kleine Abweichung toleriert, so dass auch nicht gleiche, aber ähnliche Wörter gefunden werden.[2] Anwendung findet die approximative Zeichenketten-Suche beispielsweise bei Internet-Suchmaschinen, weil auch Dokumente gefunden werden sollen, die einen Anfrageterm nicht exakt enthalten. So sollen bei der Suche nach „Levenstein“ auch „Levens*h*tein“ und bei der Suche nach „Reisen“ auch „Reise“ gefunden werden. Ein weiteres Anwendungsbeispiel für die approximative Suche in Zeichenketten ist das Auffinden von Mustern in DNA-Sequenzen, bei dem ebenfalls eine gewisse Toleranz zugelassen wird.

[2] Hier ist *approximativ* also nicht im Sinne einer näherungsweisen Lösung zu verstehen, bei der beispielsweise nicht alle Ergebnisse gefunden werden müssen. Im Gegenteil, hier werden im Allgemeinen sogar mehr Vorkommen im Text gefunden als bei der exakten Suche.

Formal kann das Problem folgendermaßen beschrieben werden: Gegeben ist ein Text T, eine Zeichenkette s und die Suchtoleranz $e \in \mathbb{R}_0^+$. Gesucht sind alle Teilzeichenketten t in T, mit $\delta(t, s) \leq e$. Dabei ist δ eine beliebige Abstandsfunktion für Zeichenketten.

Das Problem der approximativen Zeichenketten-Suche ist verwandt mit dem Problem der Suche nach ähnlichen Adressen, weil in beiden Fällen eine gegebene Menge von Zeichenketten mit einer gewissen Toleranz durchsucht werden soll. Im folgenden Abschnitt wird der Stand der Technik dargestellt und anschließend diskutiert, welche Konzepte sich auf die Suche nach ähnlichen Adressen übertragen lassen.

Stand der Technik

Die Algorithmen zur Zeichenketten-Suche können grob in zwei Gruppen aufgeteilt werden. Ein Algorithmus, der ohne Aufbau eines Indexes auf dem Text auskommt, heißt *online*; wenn der Text vorverarbeitet und indiziert wird, heißt der Algorithmus *offline*. Eine weitere dazu orthogonale Klassifizierung teilt die Algorithmen in zwei Klassen ein, je nachdem ob *exakt* oder *approximativ* gesucht wird.

Für die exakte Zeichenketten-Suche gibt es bereits viele verschiedene Algorithmen, unter anderem den Karp-Rabin- [KR87], Boyer-Moore- [BM77], Knuth-Morris-Pratt- [KMJP77], Baeza-Yates-Gonnet- (auch Shift-Or-, Shift-And- oder Bitap-) [BYG92] Algorithmus. Diese Algorithmen eignen sich jedoch nicht für die approximative Suche, so dass sie hier nicht weiter besprochen werden.

Einen sehr umfassenden Überblick über die *approximative* Zeichenketten-Suche *ohne Indizierung* gibt Navarro [Nav01]. Dort werden verschiedene algorithmische Ansätze für dieses Problem dargestellt: Dynamische Programmierung, Verwendung von Automaten, Bit-Parallelisierung und die wegen ihrer Effizienz bedeutende Klasse der Filteralgorithmen.

Die approximative Zeichenketten-Suche *mit Indizierung* wird im gleichen Jahr vom gleichen Autor ebenfalls übersichtsartig dargestellt [NBYST01]. Im Folgenden werden kurz drei verschiedene Ansätze zur Indizierung bei der approximativen Zeichenketten-Suche vorgestellt:

- k-Gramme
- Suffix-Bäume (Tries)
- Metrische Bäume

Die *k-Gramme* einer Zeichenkette sind alle Teilzeichenketten der Länge k. Zu allen k-Grammen werden die Positionen gespeichert, an denen sie im Text vorkommen. Durch diese invertierte Struktur kann bei der Anfragebearbeitung die Suche auf kleine Bereiche des Textes beschränkt werden. Es ist dadurch möglich, relativ effizient nach ähnlichen Zeichenketten in einem Text suchen. Verwendet wird dieses Verfahren beispielsweise von Ukkonen [Ukk92], der auch eine Verbindung zwischen k-Grammen und der Editierdistanz herstellt. Indizierungsverfahren mit k-Grammen werden auch bei der Rechtschreibkorrektur eingesetzt und in Abschnitt 3.2.2 detaillierter betrachtet.

Suffix-Bäume sind Tries, die alle Suffixe eines Textes enthalten, und ermöglichen die effiziente approximative Suche nach Teilzeichenketten. Tries werden auch bei der Rechtschreibkorrektur eingesetzt und in Abschnitt 3.2.3 detaillierter erläutert. Eine alternative Speicherungsform von Suffix-Bäumen sind Suffix-Arrays, in denen die Positionen der Suffixe in lexikographischer Reihenfolge abgespeichert werden.

Ein weiterer Ansatz wird in [BYN98] verfolgt. Die Suche nach einem ähnlichen Wort wird dabei auf die Suche in einem metrischen Raum reduziert. Die Metrik ist hierbei die

Editierdistanz und als Indexstrukturen werden *metrische Bäume* verwendet. Das Verfahren ist schneller als die Online-Algorithmen, jedoch nicht so effizient wie die Offline-Verfahren. Ein Vorteil ist bei der Verwendung von Indexstrukturen für metrische Räume jedoch die Flexibilität, weil statt der einfachen Editierdistanz auch eine beliebige andere Distanzfunktion gewählt werden kann (solange sie eine Metrik ist). Eine deutliche Verringerung des Speicherplatzes wird in [CN02] dadurch erreicht, dass nicht alle Teilzeichenketten, sondern nur die Knoten des Suffix-Baumes indiziert werden. Eine deutliche Verbesserung der Anfrage-Effizienz kann erreicht werden, wenn die relativ rechenaufwändige Editierdistanz durch die Bag-Distanz[3] angenähert wird [BCP02]. Indizierungsverfahren metrischer Räume werden auch bei der Datensatz-Verknüpfung eingesetzt und in Abschnitt 3.2.1 detaillierter betrachtet.

Weil die Suche mit einer hohen Suchtoleranz bei den meisten Verfahren (unter anderem bei Tries und metrischen Bäumen) nicht effizient gelöst werden kann, gibt es verschiedene Varianten. Anstatt im Index mit einer großen Suchtoleranz zu suchen, kann die Anfrage-Zeichenkette in kleinere Teile aufgespalten werden, so dass mindestens ein Teil keinen Fehler beinhaltet. Dies Teilzeichenkette kann mit einer exakten Suche effizient gefunden werden und die Fundstellen müssen danach nur noch verifiziert werden [BYN97]. Anstatt zu fordern, dass ein Teil *keinen* Fehler beinhaltet, kann die Zeichenkette auch aufgeteilt werden, so dass mindestens ein Teil nur einen *kleineren* Fehler beinhaltet. Nach diesen Teilen kann relativ effizient gesucht werden. Dieses Verfahren wurde beispielsweise in [NBY00] angewandt.

Es sind jedoch leider keine Untersuchungen bekannt, die die Effizienz der verschiedenen Indizierungsverfahren für die approximative Zeichenketten-Suche (k-Gramme, Suffix-Bäume, Suffix-Array, metrische Bäume) mit denselben Rahmenbedingungen systematisch vergleichen.

Übertragbarkeit

Die approximative Zeichenketten-Suche weist einige Gemeinsamkeiten zur Suche von ähnlichen Adressen auf. Unter anderem wird bei beiden Problemen eine Indexstruktur benötigt, die die effiziente Suche nach ähnlichen Zeichenketten unterstützt. Bei der Zeichenketten-Suche werden jedoch lange Texte (Dokumente) durchsucht, während bei der Suche nach Adressen Datensätze mit kurzen Textfeldern gesucht werden. Übernommen werden können vor allem Ideen der verwendeten Ähnlichkeitsmaße (Editierdistanz, k-Gramme) sowie die Indexstrukturen der Offline-Algorithmen (Tries, invertierte Listen von k-Grammen).

3.1.4. Rechtschreibkorrektur

Problem

Ziel der automatischen Rechtschreibkorrektur ist es, Fehler in Wörtern oder Texten zu entdecken und zu korrigieren. Dabei findet ein Abgleich mit einer Datenbank der als korrekt definierten Wörter statt, dem sogenannten Wörterbuch. Die Verfahren sind dabei nicht auf Rechtschreibfehler im engeren Sinne begrenzt, sondern eignen sich auch zur Korrektur von beispielsweise Tipp- und OCR-Fehlern. Anwendungen der automatischen Rechtschreibkorrektur finden sich in vielen Bereichen. Bei Textverarbeitungsprogrammen werden die Wörter bei der Eingabe auf Fehler geprüft und bei Internet-Suchmaschinen werden für falsch geschriebene Wörter Alternativen vorgeschlagen (oftmals bezeichnet mit „Did you mean?“). Auch bei der Eingabe mit einem Stift auf einem Touchpad und beim Einlesen von

[3] Die Bag-Distanz zweier Zeichenketten ist die Anzahl der Zeichen, die in der einen Zeichenkette vorkommen, aber nicht in der anderen. Sie bildet eine untere Schranke der einfachen Editierdistanz.

Dokumenten mit optischer Zeichenerkennung können die Wörter automatisch mit einem Wörterbuch abgeglichen werden.

Formal kann das Problem folgendermaßen beschrieben werden: Gegeben sind eine Menge von Wörtern W (das Wörterbuch) und ein Wort s. Gesucht ist das Wort $t \in W$, für das $\sigma(t, s)$ maximal ist, dabei ist σ eine beliebige Ähnlichkeitsfunktion für Zeichenketten. In manchen Systemen werden auch mehrere Ergebnisse zurückgegeben und der Benutzer kann dann aus einer Liste das richtige Wort auswählen.

Das Problem der Rechtschreibkorrektur weist viele Gemeinsamkeiten mit der Suche der ähnlichsten Adresse auf. Bei beiden Problemen ist eine Menge von korrekten Wörtern (beziehungsweise Straßen- und Ortsnamen), sowie ein möglicherweise fehlerbehaftetes Wort gegeben, das korrigiert werden soll. Im folgenden Abschnitt wird der Stand der Technik bei der Rechtschreibkorrektur erläutert und anschließend diskutiert, in wie weit sich die Ideen auf die Suche von ähnlichen Adressen übertragen lassen.

Stand der Technik

Einen sehr umfassenden Überblick über die automatische Rechtschreibfehler-Erkennung und -Korrektur stammt von Kukich [Kuk92]. Darin findet sich neben der Erläuterung verschiedener Ähnlichkeitsfunktionen auch eine Darstellung der Geschichte der verschiedenen Algorithmen und Indexstrukturen.

Die automatische Rechtschreibprüfung beschränkte sich zunächst auf die automatische Erkennung von potentiellen Fehlern. Dies geschah zum einen direkt durch Nachschlagen in einem Wörterbuch, wie beispielsweise beim Unix-Programm SPELL. Bei Eingabe eines Dokumentes wird jedes Wort in einem Wörterbuch mit ≈ 30.000 Einträgen nachgeschlagen und dann eine Liste der nicht bekannten Wörter ausgegeben [McI82]. Bei einem anderen Ansatz wird das Wörterbuch zunächst analysiert und zu jedem 3-Gramm die Anzahl der Vorkommen gespeichert [MC75]. In dem zu prüfenden Dokument wird dann für jedes Wort anhand der enthaltenen 3-Gramme ein Besonderheitswert (engl.: index of peculiarity) berechnet. Die Wörter werden nach diesem Wert sortiert und der Benutzer korrigiert die tatsächlich falsch geschriebenen Worte.

Ziel der automatischen Rechtschreib*korrektur* ist nicht nur das Erkennen sondern auch das Korrigieren der gefundenen Fehler. Ein Wort soll dabei immer auf das ähnlichste Wort (bezüglich einer Ähnlichkeitsfunktion) korrigiert werden. Die Ähnlichkeitsfunktion sollte dabei die zu erwartenden Fehler möglichst gut abbilden. Wenn Rechtschreib- oder Tippfehler erwartet werden, sollte die Ähnlichkeitsfunktion beispielsweise anders geartet sein, als bei Fehlern der optischen Zeichenerkennung. Im nächsten Absatz werden kurz einige der verwendeten Ähnlichkeitsmaße vorgestellt. Dabei wird auch erläutert, wie diese Funktionen häufige Fehler lernen können, um so bessere Ergebnisse zu erzielen.

Die am weitesten verbreitete Ähnlichkeitsfunktion für Zeichenketten ist die Editierdistanz [Kuk92, S. 17]. Die einfache Editierdistanz wird beispielsweise in [BHS07] und die gewichtete in [KCG90] verwendet. In [BM00] und [CB04] werden verallgemeinerte, gewichtete Versionen der Editierdistanz angewandt, die auch Ersetzungen von mehr als einem Zeichen zulassen. Die Gewichte der Editierdistanz werden in [KCG90] mit Hilfe von einem Trainingskorpus mit Rechtschreibfehlern, bei [CB04] durch das Auswerten von Suchmaschinen-Protokollen erlernt. In bestimmten Anwendungsbereichen werden phonetische Verfahren verwendet [TM01], insbesondere wenn die Fehler die Aussprache der Wörter unverändert lassen („Mayer“ $\leftrightarrow$ „Meier“). Teilweise werden auch verschiedene Ähnlichkeitsmaße kombiniert, um eine bessere Genauigkeit und Vollständigkeit zu erhalten [HA03].

Ein naiver Algorithmus zur Rechtschreibkorrektur würde das gesamte Wörterbuch durchlaufen, jeweils die Ähnlichkeit bestimmen und die am besten passende Zeichenkette als Ergebnis zurückliefern. Weil die Rechtschreibkorrektur eines Wortes in den meisten Anwendungsfällen jedoch in Bruchteilen einer Sekunde geschehen sollte, ist dieses Verfahren selbst für Wörterbücher moderater Größe nicht einsetzbar. Deshalb wurde eine Reihe von Indexstrukturen entwickelt, die die Suche nach ähnlichen Zeichenketten im Wörterbuch beschleunigen sollen. Diese bestimmen in der Regel eine Menge von Kandidaten, die dann anschließend nach der Ähnlichkeit zum Eingabewort sortiert werden.

Die Indexstrukturen bei der Rechtschreibkorrektur sind denen der approximativen Zeichenkettensuche (Abschnitt 3.1.3) sehr ähnlich. Es werden ebenfalls unter anderem k-Gramme sowie Tries verwendet. Diese werden deshalb gemeinsam in Abschnitt 3.2 erläutert.

Übertragbarkeit

Das Problem der Rechtschreibkorrektur ist eng verbunden mit dem Problem der effizienten Suche von Zeichenketten, welche auch die Basis für die Suche nach ähnlichen Adressen bildet. Wörterbücher sind jedoch in der Regel in der Größenordnung von 20.000 bis 100.000 Einträgen, während es ungefähr 1,2 Mio Straßen in Deutschland gibt. Übernommen werden können aus diesem Bereich Ideen bezüglich der Ähnlichkeitsmaße von Zeichenketten (Editierdistanz, k-Gramme) sowie bezüglich der vorgeschlagenen Indexstrukturen (Tries, invertierte Listen von k-Grammen).

3.2. Indizierungsverfahren

Im Folgenden werden drei Indexstrukturen und deren aktueller Stand der Forschung erläutert. Metrische Bäume indizieren einen metrischen Raum; k-Gramme und Tries werden zur Indizierung von Texten oder Mengen von Zeichenketten verwendet.

3.2.1. Metrische Bäume

Problem

In diesem Abschnitt werden Indizierungsverfahren für metrische Räume vorgestellt (Definition *Metrik*: Abschnitt 2.4). Metrische Räume werden unter anderem einerseits für die Suche nach ähnlichen Datensätzen (Abschnitt 3.1.1) und andererseits für die Suche nach ähnlichen Zeichenketten (Abschnitte 3.1.3 und 3.1.4) verwendet. Andere Anwendungsgebiete sind die Suche in einer Datenbank von Bildern, Fingerabdrücken oder Audiodateien oder auch die Ähnlichkeitssuche in einer Menge von DNA-Sequenzen. Die im Folgenden vorgestellten Verfahren sind immer genau dann anwendbar, wenn es sich bei der Abstandsfunktion um eine Metrik handelt, sie also insbesondere die Dreiecksungleichung erfüllt.

In einer Menge von Objekten (hier Datensätze beziehungsweise Zeichenketten) ist zu einem gegebenen Anfrageobjekt das Element mit dem geringsten Abstand gesucht. Formal kann dieses Problem folgendermaßen beschrieben werden: Gegeben ist ein metrischer Raum (U, δ), eine Datenbank $D \subset U$ und ein Anfrage-Element $q \in U$. Gesucht ist das Element $r \in D$ für das $\delta(r, q)$ minimal ist.

Ein naiver Algorithmus würde für ein Anfrageobjekt die Abstände zu allen Objekten berechnen. Weil dies aber bei größeren Datenbanken zu aufwändig ist, werden Indexstrukturen eingesetzt, die die Suche auf eine kleine Teilmenge einschränken sollen. Indexstrukturen für metrische Räume sind universell einsetzbar und betrachten die Objekte als „black box“. Die

Objekte sind nicht durch ihre Eigenschaften bestimmt, sondern nur durch ihren relativen Abstand zueinander.

Stand der Technik

Indexstrukturen für metrische Räume sind in der Regel Bäume und werden daher metrische Bäume genannt. Diese metrischen Bäume sollen effizient die Suche nach ähnlichen Objekten unterstützen. Dafür werden in der Regel die Objekte so partitioniert, dass bei der Suche nicht mehr die gesamte Menge, sondern nur noch ein Teil durchsucht werden muss. Diese Aufteilung der Objekt erfolgt bei den meisten metrischen Bäumen entweder durch Kugel-Partitionierung oder durch Hyperebenen-Partitionierung [HS03] (siehe Abbildung 3.1). Bei der *Kugel-Partitionierung* wird ein Element ausgewählt, das als Pivot-Element fungiert. Die übrigen Objekte werden dann aufgeteilt, je nachdem ob sie sich innerhalb oder außerhalb einer Kugel um das Pivot-Element befinden. Bei der *verallgemeinerten Hyperebenen-Partitionierung* werden zwei Objekte ausgewählt und die übrigen Objekte jeweils dem Element zugeordnet, zu dem der Abstand geringer ist. Beide Verfahren werden rekursiv für die Teilmengen wieder angewendet. Bei der Suche nach ähnlichen Elementen kann dann die Dreiecksungleichung ausgenutzt werden, so dass nicht alle Elemente durchsucht werden müssen.

Metrische Bäume, die das Konzept der Kugel-Partitionierung verwenden, sind beispielsweise der BK-Baum (nach [BK73]), der VP-Baum („vantage point“ [Yia93]), der MVP-Baum („multiple vantage point“ [BO97]) und der FQ-Baum („fixed query“ [BYCMW94]). Hyperebenen-Partitionierung wird unter anderem beim GH-Baum („generalized hyperplane“ [Uhl91]) und beim BS-Baum („bisector“ [KM83]) verwendet.

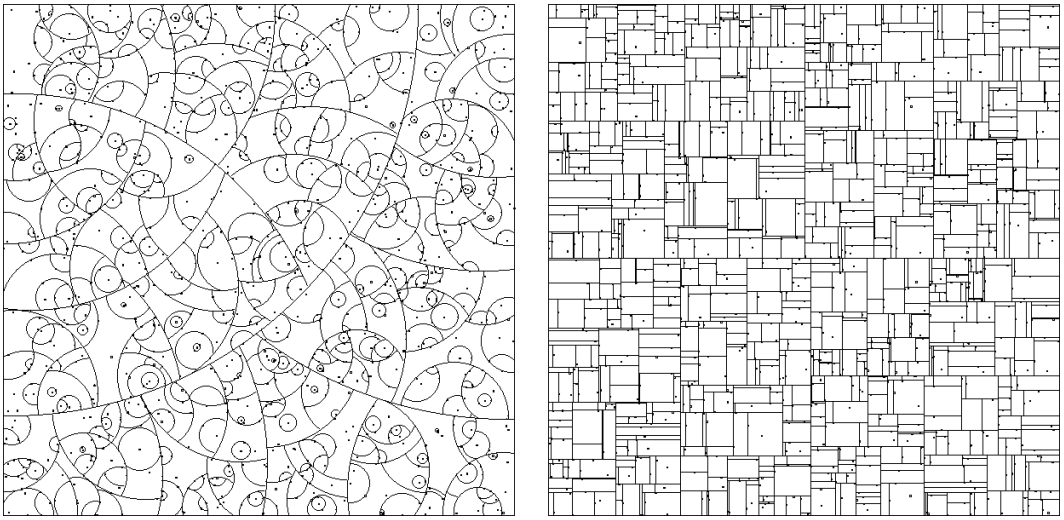

Abbildung 3.1.: Kugel-Partitionierung und Hyperebenen-Partitionierung [Yia93]

Im Folgenden wird kurz die Funktionsweise des VP-Baums genauer erläutert. Dieses Verfahren erreichte bei einem Vergleich der Suchlaufzeiten von BK-, VP- und BS-Baum die besten Ergebnisse [Sch06]. Der VP-Baum ist ein Binärbaum. Bei der Konstruktion eines VP-Baums wird zunächst ein beliebiges Element x als Pivot-Element ausgewählt (hier *vantage point* genannt). Für die übrigen Elemente wird die Distanz $\delta(y, x)$ berechnet und der

Median m dieser Distanzen bestimmt. Alle Elemente y mit $\delta(y, x) \leq m$ werden dem linken Teilbaum zugeordnet, die Elemente mit einer größeren Distanz dem rechten Teilbaum. Dieser Vorgang wird rekursiv für beide Teilmengen wiederholt. In jedem Baumknoten wird das Pivot-Element, der Median, sowie Verweise zum linken und rechten Teilbaum gespeichert.

Bei der Suche nach einem Element q mit Suchtoleranz $e \in \mathbb{R}_0^+$ wird die Distanz $\delta(q, x)$ zum Wurzelknoten x bestimmt. Ist diese Entfernung kleiner oder gleich e, wird der aktuelle Knoten in die Ergebnismenge aufgenommen. Der linke Teilbaum wird rekursiv durchsucht, wenn $\delta(q, x) - e \leq m$. Der rechte Teilbaum wird durchsucht, wenn $\delta(q, x) + e > m$. Diese Bedingungen schließen sich nicht gegenseitig aus, so dass auch beide Teilbäume durchsucht werden können.

In neueren Arbeiten wird unter anderem versucht, die metrischen Bäume für die Speicherung auf Festplatten zu optimieren, wie beim M-Tree [CRPZ97; ZWYY03] (Implementierung unter [CRZP02]) oder Slim-Tree [TJTSF00].

Übertragbarkeit

Metrische Bäume sind generisch und sehr flexibel für verschiedene Anwendungen einsetzbar. Sowohl bei der Suche nach ähnlichen Datensätzen als auch bei der Suche nach ähnlichen Zeichenketten können metrische Bäume verwendet werden (solange die Abstandsfunktion eine Metrik ist). Weil sie jedoch so allgemein gestaltet sind, können sie nicht immer alle verfügbaren Informationen zur Indizierung ausnutzen. In der Praxis sind sie deshalb nicht immer gut einsetzbar, wenn die Effizienz eine große Rolle spielt, wie im vorliegenden Problem. Im Rahmen der Datensatzverknüpfung wurden die Suchlaufzeiten von BK-, VP- und BS-Baum auf einer Datenbank mit 338.000 Adressen untersucht [Sch06]. Die Experimente wurden auf einem Rechner mit aktueller Hardwareaustattung durchgeführt (2,8 GHz Pentium 4 Prozessor, 1 Gigabyte Arbeitsspeicher). Eine Anfrage mit einer Suchtoleranz von 5 (bezogen auf die einfache Editierdistanz) benötigte bei der schnellsten Indexstruktur (VP-Baum) etwas über eine Sekunde.

3.2.2. k-Gramme

In diesem Abschnitt wird das Konzept der k-Gramme erläutert und wie sie zur Indizierungen von Zeichenketten eingesetzt werden können.[4] Die k-Gramme einer Zeichenkette sind alle Teilzeichenketten der Länge $k \in \mathbb{N}$. Sie werden vor allem bei der Verarbeitung von natürlicher Sprache eingesetzt, weil sie die Verteilung der Häufigkeiten aufeinander folgender Buchstaben und damit Charakteristiken der Sprache abbilden [Kuk92]. Sie werden wie oben erwähnt unter anderem bei der Zeichenketten-Suche (Abschnitt 3.1.3) und der Rechtschreibkorrektur (Abschnitt 3.1.4) verwendet.

Bei der Indizierung von Texten oder Wörterbüchern wird ein invertierter Index auf den k-Grammen aller Wörter aufgebaut. Darin wird zu jedem k-Gramm gespeichert, in welchen Wörtern es vorkommt. Bei der approximativen Suche in diesem Wörterbuch wird der k-Gramm-Index folgendermaßen ausgenutzt. Es werden zunächst alle k-Gramme der Anfrage-Zeichenkette gebildet. Mit Hilfe des Indexes wird zu jedem k-Gramm die Menge der zugehörigen Zeichenketten ermittelt und die Vereinigung dieser Mengen gebildet. Die so gewonnene Menge enthält Ergebnis-Kandidaten, die abschließend gegebenenfalls mit einer Ähnlichkeitsfunktion wie der Editierdistanz verifiziert werden.

Ein Verfahren zur Verkleinerung des Indexes ist der Einsatz von sogenannten *algebraischen Signaturen*, bei denen mehrere aufeinander folgende Buchstaben zu einem Zeichen zusam-

[4] k-Gramme werden in anderen Arbeiten auch beispielsweise als *n-Gramme* oder *q-Gramme* bezeichnet.

mengefasst werden [LMRS07]. Ein weiteres Verfahren zum Einsparen von Speicherplatz ist die Verwendung von *variablen Grammen*, die nicht mehr eine im Voraus festgelegte Länge k haben [LWY07]. Es werden dabei von allen möglichen Grammen verschiedener Länge solche Gramme zur Indizierung ausgewählt, die einen hohen Nutzen bei der Anfrage-Beantwortung versprechen. Insbesondere werden bei diesem Ansatz nur kurze Gramme ausgewählt sowie Gramme, die häufig vertreten sind. Die Autoren analysieren die Verbindung zwischen der Ähnlichkeit bezüglich der variablen Gramme von zwei Zeichenketten und deren Editierdistanz. In Experimenten wird der Einfluss verschiedener Parameter des Verfahrens (Maximale Größe der Gramme, minimale Häufigkeit etc.) untersucht.

3.2.3. Tries

In diesem Abschnitt wird erläutert, was ein Trie ist, wie er zur Indizierung von Zeichenketten eingesetzt wird und welche Varianten es gibt. Ein Trie ist eine Datenstruktur zur Verwaltung einer Menge von Zeichenketten in sortierter Reihenfolge. Tries bieten effiziente Unterstützung für die exakte sowie für die approximative Suche und werden deshalb unter anderem bei der approximativen Zeichenketten-Suche und der Rechtschreibkorrektur eingesetzt. Ein Trie besteht aus einer Baumstruktur, bei der jeder Knoten bis zu $|\Sigma|$ Kinder haben kann. Der Wurzelknoten repräsentiert die leere Zeichenkette ϵ. Jede von einem Knoten ausgehende Kante steht für einen Buchstaben. Jeder Knoten repräsentiert implizit die Zeichenkette, die durch die Kanten auf dem Weg von der Wurzel bis zum Knoten gegeben ist. Die Blattknoten repräsentieren die tatsächlich gespeicherten Zeichenketten.[5]

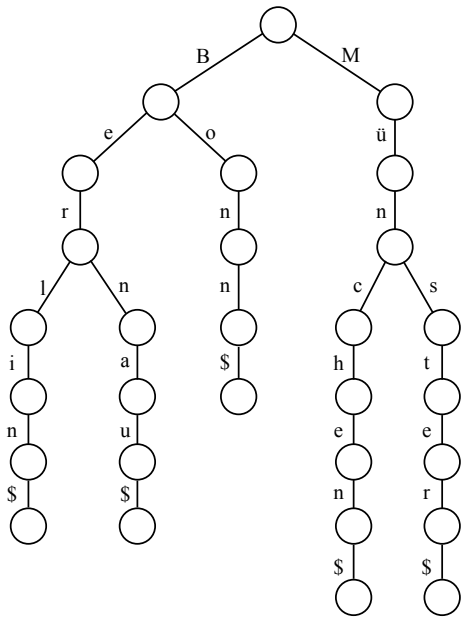

Abbildung 3.2.: Trie mit fünf Zeichenketten

In einem Trie müssen in jedem Knoten Verweise auf die Kinder gespeichert werden, was

[5] Damit auch Zeichenketten gespeichert werden können, die ein Präfix einer anderen gespeicherten Zeichenkette sind, werden alle Wörter mit einem zusätzlichen Zeichen „$“ abgeschlossen.

einen recht hohen Speicheraufwand darstellt. Andererseits werden gemeinsame Präfixe der Zeichenketten nicht mehrfach sondern nur einmal abgespeichert. Dadurch wird Speicherplatz eingespart, was in der Praxis teilweise sogar zu einer Komprimierung der Daten führen kann [SM96].

Zuerst vorgeschlagen wurden Tries von de la Briandais [Bri59], der Begriff wurde von Fredkin [Fre60] geprägt und stammt von „information re*trie*val“.[6] Die Verwendung von Tries als Datenstruktur für die exakte Suche in einem Wörterbuch wurde von Knuth [Knu73] vorgeschlagen: Um eine Zeichenkette zu finden, muss nur dem Zeiger des jeweils nächsten Buchstabens in der Zeichenkette gefolgt werden. Später werden Tries auch für die approximative Suche verwendet [DM81]. Um zu einer Anfragezeichenkette alle Zeichenketten innerhalb einer Suchtoleranz e zu finden, wird grundsätzlich wie bei der exakten Suche vorgegangen. Anstatt aber immer genau dem Zeiger zu folgen, der dem nächsten Buchstaben der Anfrage-Zeichenkette entspricht, werden auch andere Zeiger verfolgt, solange die Suchtoleranz e noch nicht überschritten ist. Dies wird genauer im Abschnitt 5.2 erläutert.

Reduzierung des Speicherbedarfs

Es gibt verschiedene Techniken, um den Speicherbedarf von Tries zu verringern. Ein Patricia-Trie [Mor68] (engl. auch radix tree) ist ein Trie, bei dem alle aufeinander folgenden Knoten ohne Verzweigungen zu einem Knoten zusammengefasst werden. Die Beschriftung der Kanten ist dann nicht mehr ein einzelner Buchstabe sondern eine Folge von Buchstaben. Bei einer alternativen Variante wird an einer Kante nur der erste Buchstabe der zugehörigen Buchstabenfolge und ihre Länge gespeichert. Diese zweite Variante genügt für einige Anwendungen, eignet sich jedoch nicht für die approximative Suche, weil durch das Weglassen der Buchstaben die Zeichenketten nicht rekonstruiert werden können.

Normalerweise werden die Zeiger zu den Kindknoten in einem Array gespeichert. Dadurch ist ein sehr schneller gezielter Zugriff auf den Kindknoten mit einem bestimmten Buchstaben möglich. Die Speicherung der Arrays benötigt jedoch relativ viel Speicherplatz, weil in der Regel nur ein wenig Einträge des Arrays belegt sind. Eine platzsparende Alternative wären Listen, die jedoch den Nachteil einer längeren Suchzeit haben. Mit den Dreiwege-Suchbäumen (engl.: ternary search tree, TST) wird ein Kompromiss zwischen geringem Platzbedarf und kurzer Suchezeit vorgeschlagen [BS97] [BS98]. Diese Datenstruktur verwendet konzeptionell Binärbäume zur Speicherung der Verweise auf die Kindknoten und wird in Abschnitt 5.2 im Detail erläutert.

Eine weitere Variante von Tries, die versucht den Speicherbedarf zu reduzieren, ist der Burst-Trie [HZW02]. Dabei werden Teilbäume, die nur wenige Zeichenketten enthalten, nicht als Baum abgespeichert, sondern als Binärbaum oder verkettete Listen. Burst-Tries benötigen in experimentellen Untersuchungen der Autoren weniger Speicherplatz und eine geringere Suchzeit als beispielsweise Ternary-Search-Trees. Eine Weiterentwicklung des Burst-Tries ist der HAT-Trie [AS07], der an Stelle der verketteten Listen eine Array-Datenstruktur verwendet. Dadurch soll die Verwendung des Prozessor-Caches besser unterstützt werden. In experimentellen Untersuchungen zeigen die Autoren eine Verbesserung der Suchzeit gegenüber dem herkömmlichen Burst-Trie.

[6] Die korrekte Aussprache ist davon ausgehend [tri] (wie bei „tree“). Andere bevorzugen zur Unterscheidung von einem herkömmlichen Baum die Aussprache [trai] (wie bei „try“).

Beschleunigung der Suche

Weil der Zeitbedarf der approximativen Suche exponentiell mit der Suchtoleranz e steigt, gibt es verschiedene Verfahren, um die Suche mit einer hohen Toleranz auf mehrere Suchen mit einer geringeren Suchtoleranz zurückzuführen. Zwei Möglichkeiten werden im Folgenden vorgestellt, einerseits die Nachbar-Erzeugung und andererseits das Rückwärtswörterbuch.

Bei der *Nachbar-Erzeugung* (engl.: neighborhood generation) werden zu einer Anfrage-Zeichenkette alle Zeichenketten konstruiert, die innerhalb der Suchtoleranz e liegen. Mit diesen Zeichenketten kann dann effizient exakt im Index gesucht werden. Es gibt jedoch schon bei kleiner Toleranz sehr viele solcher zu generierenden Zeichenketten. Anstatt also alle Zeichenketten der Toleranz e zu erzeugen und exakt zu suchen, können alternativ auch alle Zeichenketten mit einer kleineren Toleranz $e' < e$ erzeugt werden, gefolgt von einer Suche im Index mit Toleranz $e - e'$. So werden insgesamt wieder alle Zeichenketten innerhalb der Toleranz e gefunden. Es muss hier also ein Kompromiss gefunden werden zwischen einer hohen Toleranz bei der Erzeugung der Nachbarn und einer hohen Toleranz bei der Suche. Dies ist momentan ein wichtiger Teil der Forschung in diesem Gebiet. In [RO05] wird dieses Verfahren erläutert.

Noch weiter geht ein Ansatz, der sogenannte *Auslassungen* (engl.: deletions) verwendet. Dabei werden zu den Zeichenketten im Wörterbuch die Nachbarn mit einer gewissen Toleranz schon im Index abgespeichert. (Verwendet werden in der Regel jedoch nur Nachbarn, die durch Auslassen von einzelnen Zeichen entstehen.) Dies bewirkt eine Vergrößerung des Indexes, kann die Suche jedoch deutlich beschleunigen. Die Idee wurde ursprünglich in Verbindung mit Hash-Tabellen angewandt, geht unter anderem zurück auf [MF82] und wurde beispielsweise in [BHS07] aufgegriffen. Das Verfahren wurde jedoch auch erfolgreich für Tries eingesetzt [MS04] und erreicht vor allem bei kurzen Zeichenketten und geringer Suchtoleranz eine Beschleunigung.

Eine weitere Variante ist die Verwendung eines zusätzlichen Tries, der die Zeichenketten in umgekehrter Reihenfolge enthält, genannt *Rückwärtswörterbuch* (engl.: backwards dictionary [MS04]). Bei der Suche wird eine Zeichenkette s aufgeteilt in zwei ungefähr gleich große Zeichenketten s_1 und s_2. Zunächst wird im Vorwärtswörterbuch die Zeichenkette s_1 mit einer Toleranz von $\lceil \frac{e}{2} \rceil$ gesucht. Die Suche wird dann von allen erreichten Knoten mit der vollen Toleranz e fortgesetzt und die gefundenen Zeichenketten zur Ergebnismenge hinzugefügt. Analog wird anschließend $s^r = s_2^r \circ s_1^r$ im Rückwärtswörterbuch gesucht.[7] Zunächst wird nach s_2^r mit der halben Toleranz und von den gefundenen Knoten anschließend nach s_1^r mit der vollen Toleranz gesucht. Durch dieses Verfahren wird verhindert, dass die gesamte Zeichenkette mit einer hohen Toleranz gesucht wird. Stattdessen werden zwei Suchen mit der halben Toleranz durchgeführt, was deutlich effizienter ist. In der jeweils zweiten Hälfte der Zeichenkette wird dann zwar wieder mit der vollen Toleranz gesucht, aber dies geschieht erst relativ weit hinten im Baum, so dass nur wenige Knoten zusätzlich besucht werden müssen. Experimentelle Untersuchungen der Autoren zeigen eine drastische Verbesserung der Suchzeiten (für $e = 3$ in einem deutschen Wörterbuch um Faktor 35 schneller als ohne Rückwärtswörterbuch) [MS04, S. 20].

[7] s^r bezeichnet die Zeichenkette s in umgekehrter Reihenfolge.

3.3. OCR-Nachkorrektur

Problem

Ziel der OCR-Nachkorrektur ist es, den von einer OCR-Software erkannten Text so mit einem Lexikon abzugleichen, dass danach im Text möglichst wenig Fehler vorhanden sind. Formal kann dies wie folgt beschrieben werden: Gegeben ist ein Wörterbuch W und ein Anfragewort s. Gesucht ist das Original-Wort $t \in W$, welches die OCR-Software möglicherweise fehlerhaft als s gelesen hat.

Einerseits müssen bei diesem Problem die Wörter im Wörterbuch gefunden werden, die ähnlich zum Anfragewort sind. Andererseits soll aus diesen Wörtern genau das Wort ausgewählt werden, das mit möglichst großer Wahrscheinlichkeit das eigentlich korrekte Originalwort war. Das Suchen in Wörterbüchern wurde schon im Zusammenhang mit der Rechtschreibkorrektur behandelt (Abschnitt 3.1.4), hier liegt der Fokus daher auf der Bestimmung des richtigen Korrekturwortes und wie die Parameter des Verfahrens in einem Trainingsprozess erlernt werden können.

Stand der Technik

Das Einlesen eines Textes wird bei der OCR-Nachkorrektur in den meisten Arbeiten als gestörter Kanal (engl.: noisy channel) betrachtet [JHZ02; KR02; Sch03]. Es wird also der Originaltext „versendet“, der Kanal (hier die OCR-Software) ersetzt einige Wörter durch andere Zeichenketten und beim Empfänger kommt ein fehlerhafter Text an (siehe Abbildung 3.3).

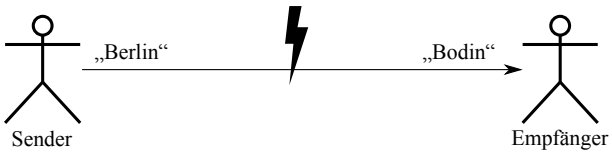

Abbildung 3.3.: Gestörter Kanal

Es werden zwei grundlegend verschiedene Ansätze der Korrektur verfolgt [Sch03]. Zum einen wird versucht die Zeichenketten mit Hilfe von statistischen Methoden zu korrigieren (*bottom-up*), zum anderen werden lexikalische Verfahren (*top-down*) verwendet.

Beim den *statistischen Korrekturmethoden* wird versucht, die Fehler in den Zeichenketten allein auf Basis von Statistiken zu ermitteln und zu korrigieren. Dabei werden unwahrscheinliche Buchstabenfolgen gesucht und durch wahrscheinlichere ersetzt. Die Statistiken werden aus dem Lexikon gewonnen und beinhalten beispielsweise die Buchstabenübergangs-Wahrscheinlichkeiten (also die Wahrscheinlichkeit, dass auf einen Buchstabe u der Buchstabe w folgt) und die Häufigkeiten aller möglichen k-Gramme der Wörter im Wörterbuch. Aus diesen Werten können falsche Zeichenketten mit Hilfe von statistischen Methoden korrigiert werden. Dieses Verfahren hat jedoch einige Nachteile. Es kann beispielsweise auch geschehen, dass ein nicht im Wörterbuch vorhandenes Wort generiert wird [Sch03, S. 48] und wird deshalb häufig in Kombination mit einem lexikalischen Verfahren eingesetzt.

Bei den *lexikalischen Verfahren* wird eine Ähnlichkeitssuche im Wörterbuch durchgeführt. Zu einer Anfragezeichenkette s soll das Wort $t \in W$ gefunden werden, das mit möglichst großer Wahrscheinlichkeit das korrekte Originalwort darstellt. Diese Wahrscheinlichkeit wird dabei

häufig mit der Methode von Bayes modelliert [TE96; KR02; Sch03; Rin03; LH03]. (Genaueres zu diesem Verfahren findet sich in 2.5.) Die Wahrscheinlichkeiten können beispielsweise mit einem Korpus von Trainingsdaten erlernt werden. Wenn in der Trainingsmenge zu jedem von der OCR-Software erkannten Wort das eigentlich korrekte Wort bekannt ist, kann bestimmt werden, welche Fehler bei dieser OCR-Software wie häufig auftreten. Dies kann unter anderem mit dem *Viterbi-Algorithmus* geschehen, einem dynamischen Programmierungs-Algorithmus, der in einem Verborgenen-Markow-Modell (engl.: hidden Markov model) die Folge der wahrscheinlichsten Zustandsübergänge findet.

Übertragbarkeit

Das oben erläuterte lexikographische Verfahren lässt sich sehr gut auch auf das Korrigieren von Adressdatensätzen anwenden. Auch dort ist ein Wörterbuch von korrekten Wörtern (Straßen- und Ortsnamen) gegeben und eine Anfragezeichenkette soll auf die Zeichenkette korrigiert werden, die mit der größten Wahrscheinlichkeit die Originalzeichenkette darstellt.

3.4. Adresskorrektur

Problem

Ziel der automatischen Adresskorrektur ist es, eine gegebene, möglicherweise falsche Adresse mit Hilfe einer Datenbank von korrekten Adressen zu korrigieren. In diesem Bereich gibt es zwar auch wissenschaftliche Beiträge, jedoch insgesamt nur relativ wenige. Dies liegt wahrscheinlich daran, dass die Adresskorrektur vor allem im kommerziellen Umfeld Anwendung findet. Haupteinsatzgebiet ist die Korrektur von Adressen zum Zweck der Sortierung von Briefen.

Stand der Technik

Noch vor einigen Jahren war die verfügbare Rechenleistung deutlich geringer. Deshalb wird in [RJH91] ein zweistufiges Verfahren vorgeschlagen. In der ersten Stufe wird nur eine Korrektur von Postleitzahl und Ort vorgenommen. Während die Briefe dann zum Bestimmungsort transportiert werden, wird in einer zweiten Stufe eine Korrektur auch unter Einbeziehung des Straßennamens vorgenommen. Für die Korrektur stehen somit mehrere Stunden zur Verfügung.[8]

Bei der optischen Erkennung der Empfänger-Adresse kommt es vor, dass das Textfeld nur teilweise lesbar ist, beispielsweise weil es gegen das Sichtfenster des Briefes verschoben ist. In [BM05] wird ein Verfahren vorgeschlagen, mit dem das Problem behandelt wird und die fehlenden Buchstaben rekonstruiert werden.

Eine Vervollständigung von nur teilweise erkannten Adressen wird auch in [BQ06] vorgeschlagen. Die fehlenden Bestandteile werden dabei mit Hilfe von in der Vergangenheit gelesenen Adressen rekonstruiert.

Es gibt auch Firmen, deren Hauptgeschäftsfeld die Entwicklung von Methoden zur Adresskorrektur ist. Die Firma Syslore ist beispielsweise auf die unscharfe Suche in großen Adressdatenbanken unter Echtzeitbedingungen spezialisiert [Hyv04]. Dabei entwickelte sie ein skalierbares System von Softwareagenten, das auf mehreren Rechnern verteilt laufen kann. Als Index werden dabei Tries eingesetzt, die für den speziellen Einsatz optimiert wurden.

[8] In dieser Diplomarbeit soll die Adresskorrektur hingegen innerhalb von ungefähr 100 ms geschehen.

Die Indexstruktur kann dabei im Arbeitsspeicher gehalten werden und die Anfragezeit ist sublinear. Das Verfahren ist patentiert und wird in [Hyv05] genauer beschrieben.

Übertragbarkeit

Das Problem der Adresskorrektur ist genau das Problem, das auch in dieser Arbeit gelöst werden soll. Es gibt dazu jedoch nur relativ wenig wissenschaftliche Beiträge. Die Verfahren sind entweder veraltet (Korrektur während des Transportes), nicht genau auf das hier vorliegende Problem übertragbar (Behandlung von abgeschnittenen Textfeldern) oder nur vage ausgeführt. Das Patent der Firma Syslore habe ich nicht ausgewertet, weil das darin verwendete Verfahren patentiert ist und daher hier ohnehin nicht direkt übernommen werden kann.

3.5. Zusammenfassung

In diesem Kapitel wurden verwandte Arbeiten aus verschiedenen Gebieten der Informatik vorgestellt. Dabei wurden einige Ideen für die Beantwortung der in Abschnitt 1.4 gestellten sechs Fragen gesammelt.

1. Die Ähnlichkeit von Zeichenketten wird bei der Datensatzverknüpfung, der Rechtschreibkorrektur und der OCR-Nachkorrektur unter anderem mit Hilfe von k-Grammen und dem Jaro-Maß definiert. Überwiegend wird jedoch die Editierdistanz mit verschiedenen Varianten verwendet.

2. Die Ähnlichkeit von Datensätzen wird bei der Datensatzverknüpfung in vielen Arbeiten mit Hilfe einer Support Vector Machine bestimmt.

3. Als Datenstruktur für die Suche nach ähnlichen Zeichenketten werden bei der Rechtschreibkorrektur und bei der Zeichenketten-Suche Tries und invertierte Listen von k-Grammen eingesetzt.

4. Als Datenstruktur für die Suche nach ähnlichen Datensätzen werden bei der Datensatzverknüpfung teilweise metrische Bäume eingesetzt. Diese sind jedoch nicht für den Einsatz bei der Adresskorrektur unter Echtzeitbedingungen geeignet.

5. Das Erlernen der Ähnlichkeitsfunktion für Zeichenketten geschieht bei der Datensatzverknüpfung und der OCR-Nachkorrektur überwiegend durch eine Anpassung der Gewichte der Editierdistanz, je nachdem wie häufig eine Operation in einer Trainingsmenge auftritt.

6. Das Erlernen der Ähnlichkeitsfunktion für Datensätze geschieht bei der Datensatzverknüpfung überwiegend durch Training der Klassifikationsfunktion einer Support Vector Machine.

4. Ähnlichkeitsmaße

In diesem Kapitel werden die allgemeinen Anforderungen an Ähnlichkeitsfunktionen für Zeichenketten und Datensätze formuliert und erörtert, wie diese Anforderungen konkret bei der Adresskorrektur aussehen. Nachfolgend werden verschiedene Ähnlichkeitsmaße für Zeichenketten vorgestellt. Abschließend wird die Ähnlichkeit von Datensätzen behandelt, basierend auf der Ähnlichkeit der einzelnen Attribute. (In Abschnitt 2.2 wurde bereits formal definiert, was eine Ähnlichkeits- und was eine Abstandsfunktion ist.)

Die Auswahl einer geeigneten Ähnlichkeitsfunktion ist bei der Suche von ähnlichen Datensätzen von großer Bedeutung, weil anhand dieses Kriteriums in den Datensätzen gesucht wird. Bei der Sichtweise als relationale Datenbank (Abschnitt 2.3) wird dieses Maß für die Sortierung verwendet, bei der Betrachtung als metrischer Raum bestimmt dieses Maß die Verteilung der Datensätze im Raum (Abschnitt 2.4).

4.1. Anforderungen

4.1.1. Allgemein

In diesem Abschnitt werden allgemein die Anforderungen an ein Ähnlichkeitsmaß erläutert. Dies beinhaltet sowohl die Qualität der induzierten Ähnlichkeiten als auch die Laufzeit der Berechnung und die Indizierbarkeit bezüglich der Ähnlichkeitsfunktion.

Aus Basis des Ähnlichkeitsmaßes soll zu einem fehlerbehafteten Datensatz der am besten passende korrekte Datensatz gefunden werden. Die Ähnlichkeit von Datensätzen wird zurückgeführt auf die Ähnlichkeit der einzelnen Attribute. Die jeweiligen Ähnlichkeitsfunktionen der Attribute sollte die möglichen Fehler bei diesem Attribut widerspiegeln. Bei Attributen vom Typ Zeichenkette sollte die Ähnlichkeitsfunktion beispielsweise zwei Attributwerte genau dann als ähnlich einstufen, wenn sie durch einen Fehler ineinander überführt werden können. Dabei sollten insbesondere häufige Fehler gut modelliert werden (siehe auch Abschnitt 2.5.2).

Neben den qualitativen Anforderungen gibt es auch technische Anforderungen an die Ähnlichkeitsfunktion. Sie sollte in jedem Fall effizient berechenbar sein, weil sie bei der Anfragebeantwortung (Suche des Datensatzes mit der größten Ähnlichkeit) gegebenenfalls für viele Paare von Attributen beziehungsweise Datensätzen aufgerufen wird. Für die Ähnlichkeitsmaße für Zeichenketten wird in Abschnitt 4.2 jeweils die asymptotische Laufzeit im ungünstigsten Fall (engl.: worst case) angegeben. Eine weitere wünschenswerte Eigenschaft ist, dass die Ähnlichkeitsfunktion die Erstellung eines Indexes zulässt, mit dem effizient nach ähnlichen Datensätzen gesucht werden kann. Für einige der Zeichenketten-Ähnlichkeitsfunktionen gibt es Verfahren, die eine ungenaue Suche auf Basis der Ähnlichkeitsfunktion zulassen.

4.1.2. Adresskorrektur

Beim konkreten Problem der Suche von ähnlichen Adressen gibt es spezielle Anforderungen an die Ähnlichkeitsfunktion, die in diesem Abschnitt erläutert werden. Die Ähnlichkeit von zwei Adressdatensätzen basiert auf der Ähnlichkeit der Attribute plz, ort und str. Diese

drei Attribute können, wie in Abschnitt 2.5.2 erläutert, von verschiedenen Fehlern betroffen sein. Bei allen drei Attributen können Tipp- und OCR-Fehler auftreten. Bei den Attributen *ort* und *str* gibt es zusätzlich noch Synonyme, Abkürzungen und Rechtschreibfehler. Eine Ähnlichkeitsfunktion sollte dieses Fehler möglichst gut abbilden. Weil die Attribute von verschiedenen Fehlern betroffen sind, sollten also auch verschiedene Ähnlichkeitsmaße verwendet werden. Bei den im nächsten Abschnitt vorgestellten Ähnlichkeitsmaßen für Zeichenketten wird jeweils diskutiert, in wie fern es für die Ähnlichkeitssuche bei Adressen geeignet ist. Das Problem der Synonyme lässt sich nicht gut durch eine syntaktische Ähnlichkeitsfunktion lösen und wird daher separat behandelt (siehe Abschnitt 5.3.3).

4.2. Ähnlichkeit von Zeichenketten

Im Folgenden werden einige Ähnlichkeitsfunktionen für Zeichenketten vorgestellt. Es gibt dabei sehr verschiedene Ansätze: Einige Funktionen vergleichen die Zeichenketten Buchstabe für Buchstabe, andere zählen die Anzahl an Operationen die nötig sind, um eine Zeichenkette in die andere zu überführen. Wieder andere betrachten jeweils Teilzeichenketten fester Länge oder basieren auf der Aussprache der Wörter. Für jede Ähnlichkeitsfunktion wird jeweils die Idee, die Definition und mindestens ein Beispiel angegeben. Außerdem werden die Eigenschaften der Symmetrie und Dreiecksungleichung sowie die benötigte Laufzeit analysiert. Des Weiteren wird die Beschränkung der Funktion analysiert, welche unter anderem für eine Normalisierung des Wertes (beispielsweise auf das Intervall $[0, 1]$) erforderlich ist. Abschließend wird jeweils die Eignung für die Adresskorrektur erörtert. Die absoluten Ergebnisse der verschiedenen Ähnlichkeitsfunktionen sind nicht ohne Weiteres miteinander vergleichbar und stellen insbesondere im Allgemeinen keine Wahrscheinlichkeiten dar.

Ein guter Überblick über Zeichenketten-Ähnlichkeits-Funktionen in findet sich auch in [Nav01] und [PW97]. Implementierungen von einigen Ähnlichkeitsfunktionen sind in der Software-Bibliothek SimMetrics [Cha06] enthalten.

4.2.1. Hamming-Distanz

Definition

Die Hamming-Distanz δ_{ham} ist ein Maß für den Abstand zweier Binärfolgen gleicher Länge. In [Ham50] ist sie ist definiert als die Anzahl der Stellen, an denen sich die Binärfolgen unterscheiden. Der Hamming-Abstand δ_{ham} von zwei Zeichenketten kann analog definiert werden, mit dem einzigen Unterschied, dass nicht nur 0-1-Folgen sondern Zeichenfolgen über einem beliebigen Alphabet Σ zugelassen sind.

Beispiel

Hamming-Distanz δ_{ham} von „Berlin“ und „Bergen“ ist 2, weil sie sich an zwei Stellen unterscheiden.

Analyse

Die Hamming-Distanz ist eine Metrik, weil sie Symmetrie und die Dreiecksungleichung erfüllt. Des Weiteren gilt:

$$0 \leq \delta_{\text{ham}}(s,t) \leq |s| \,(= |t|)$$

Die Laufzeit von der Berechnung von $\delta_{\text{ham}}(s,t)$ ist $O(|s| + |t|)$, weil die Zeichenketten Buchstabe für Buchstabe durchlaufen und beide Zeichen auf Gleichheit geprüft werden können.

Eignung

Die Hamming-Distanz ist beschränkt auf Zeichenketten gleicher Länge und eignet sich vor allem für Situationen, in denen bei der Übertragung von Binärfolgen einzelne Bits oder Zeichen verändert werden. OCR-Fehler, Rechtschreib- und Tippfehler werden durch dieses Modell jedoch nicht gut abgebildet, da bei diesen nicht nur einzelne Buchstaben verändert, sondern auch Buchstaben eingefügt und gelöscht sowie aufeinander folgende Buchstaben vertauscht werden. Die Hamming-Distanz eignet sich offensichtlich nicht gut, um diese Art von Fehlern abzubilden.

4.2.2. Schreibmaschinendistanz

Definition

Die Schreibmaschinendistanz (engl.: typewriter distance [Wik08b]) δ_{tw} ist eine Erweiterung der Hamming-Distanz, deren Fokus vor allem auf Tippfehlern liegt. Die Kosten für Ersetzen von einem Buchstaben durch einen anderen sind nicht mehr immer 1, sondern variieren, je nachdem wie weit die zugehörigen Tasten auf der Tastatur voneinander entfernt sind. Die Idee dabei ist, dass bei einem Tippfehler in den meisten Fällen aus Versehen eine Taste gedrückt wird, die nah an der eigentlich korrekten Taste ist. Der Abstand $d(u,w)$ zweier Zeichen u und w berechnet sich als der Manhattan-Abstand[1] der entsprechenden Tasten auf einer „Qwertz“-Tastatur. Der Abstand von zwei Zeichenketten ist dann die Summe der Abstände der paarweise verglichenen Zeichen. Falls die beiden Zeichenketten nicht gleich lang sind, können für jedes überstehende Zeichen z. B. Kosten von $d_{\max} = 13$ veranschlagt werden, das ist eins mehr als der maximal mögliche Abstand zweier Tasten.

Abbildung 4.1.: Tastatur

[1] Der Manhattan-Abstand ist definiert als die Summe des Abstands in x-Richtung und des Abstands in y-Richtung.

Beispiel

$$\begin{aligned}&\delta_{\text{tw}}(\text{„Berlin“}, \text{„Belgien“})\\ =&d(\text{„R“}, \text{„L“}) + d(\text{„L“}, \text{„G“}) + d(\text{„N“}, \text{„E“}) + d_{\max}\\ =&6 + 4 + 6 + 13 = 29.\end{aligned}$$

Analyse

Die Schreibmaschinendistanz ist eine Metrik, weil sie Symmetrie und die Dreiecksungleichung erfüllt. Des Weiteren gilt:

$$0 \leq \delta_{\text{tw}}(s,t) \leq 13 * |s| = 13 * |t|$$

Die Laufzeit der Berechnung von $\delta_{\text{tw}}(s,t)$ ist $O(|s| + |t|)$, weil die Zeichenketten Buchstabe für Buchstabe durchlaufen und die jeweiligen Kosten addiert werden können.

Eignung

Die Schreibmaschinendistanz eignet sich sehr gut, wenn das Augenmerk vor allem auf Tippfehlern liegt. Für Rechtschreib-, OCR- und sonstige Fehler ist sie jedoch weniger geeignet.

4.2.3. Editierdistanz

Definition

Die einfache Editierdistanz (sogenannte Levenshtein-Distanz, vorgeschlagen von Vladimir Levenshtein in [Lev66]) zwischen zwei Zeichenketten s und t ist definiert als die Kosten der Operationen, die nötig sind, um s in t zu überführen. Erlaubte Operationen sind dabei (Definition nach [Cha06]):

- Übernehmen eines Zeichens (Kosten = 0)
- Ersetzen eines Zeichens durch ein anderes (Kosten = 1)
- Einfügen eines Zeichens (Kosten = 1)
- Löschen eines Zeichens (Kosten = 1)

Die Levenshtein-Distanz zweier Zeichenketten s und t lässt sich berechnen als $\delta_{\text{edit}}(s,t) := L_{|s|,|t|}$, wobei für L folgende Rekursionsgleichung gilt:

$$L_{0,0} = 0$$

$$L_{i,j} = \min \begin{cases} L_{i-1,j-1} + 0 & \text{Übernehmen (wenn } s[i] = t[j]) \\ L_{i-1,j-1} + 1 & \text{Ersetzen (wenn } s[i] \neq t[j]) \\ L_{i-1,j} + 1 & \text{Einfügen} \\ L_{i,j-1} + 1 & \text{Löschen} \end{cases}$$

		B	e	l	g	i	e	n
	0	1	2	3	4	5	6	7
B	1	0	1	2	3	4	5	6
e	2	1	0	1	2	3	4	5
r	3	2	1	1	2	3	4	5
l	4	3	2	1	2	3	4	5
i	5	4	3	2	2	2	3	4
n	6	5	4	3	3	3	3	3

Tabelle 4.1.: Levenshtein-Matrix für „Berlin“ und „Belgien“

Beispiel

Die Levenshtein-Distanz von „Berlin“ und „Belgien“ beträgt 3. Dies kann an der in Tabelle 4.1 angegebenen Matrix abgelesen werden. Eine mögliche Folge von Transformationen ist:

- Übernehmen von „B“ (Kosten = 0)
- Übernehmen von „e“ (Kosten = 0)
- Löschen von „r“ (Kosten = 1)
- Übernehmen von „l“ (Kosten = 0)
- Einfügen von „g“ (Kosten = 1)
- Einfügen von „e“ (Kosten = 1)
- Übernehmen von „n“ (Kosten = 0)

Analyse

Die einfache Editierdistanz ist eine Metrik, weil sie Symmetrie und die Dreiecksungleichung erfüllt. Des Weiteren gilt:

$$||s| - |t|| \leq \delta_{\text{edit}}(s,t) \leq \max(|s|,|t|)$$

Die Laufzeit der Berechnung von $\delta_{\text{edit}}(s,t)$ ist $O(|s| \cdot |t|)$, weil in dieser Zeit alle Werte der Matrix L berechnet werden können.

Eignung

Die einfache Editierdistanz ist für viele Anwendungsfälle ein sinnvoller Ansatz. Sie beruht auf einem realistischen Fehlermodell, das Ersetzungen, Einfügungen und Löschungen vorsieht und kann in einer einfachen Rekursionsgleichung formuliert werden. Das Modell geht jedoch von gleichen Kosten für Einfügen, Löschen und Ersetzen aus, was in einigen Anwendungen (z. B. bei der Nachkorrektur von OCR-Adressen) nicht unbedingt sinnvoll ist. Es kann aber leicht erweitert werden, so dass variable Kosten für die verschiedenen Operationen veranschlagt werden (siehe 4.2.4). Ein weiterer Vorteil der Editierdistanz ist, dass eine Menge von Zeichenketten mit Hilfe von Tries bezüglich der Editierdistanz indiziert werden können (siehe Abschnitt 5.2).

4.2.4. Editierdistanz mit Gewichten

Definition

Die gewichtete Editierdistanz $\delta_{\text{edit}'}$ ist eine Erweiterung der einfachen Levenshtein-Distanz. Dabei werden nicht mehr uniforme Kosten für die Operationen Einfügen, Löschen und Ersetzen gefordert, sondern jede dieser Operationen kann unterschiedliche Kosten haben. Die Kosten können sogar variieren, je nachdem welches Zeichen gelöscht, eingefügt oder ersetzt wird. Die Idee dabei ist, dass einige Fehler deutlich häufiger auftreten und deshalb geringere Kosten haben sollten. So ist es in der Praxis relativ unwahrscheinlich, dass bei der optischen Zeichenerkennung ein „W“ an Stelle eines „o“ gelesen wird, eine Verwechslung von „i“ mit „l“ (kleines „L“) ist hingegen recht wahrscheinlich und sollte daher nur geringe Kosten haben. Außerdem könnten die Kosten wie schon bei der Schreibmaschinendistanz 4.2.2 auf dem Abstand der Buchstaben auf der Tastatur basieren. (Für eine effiziente Berechenbarkeit wird gefordert, dass kein Buchstabe mehrmals verändert wird, also z. B. erst durch ein Zeichen und dieses gleich darauf durch ein anderes ersetzt wird.)

Die Funktion $d\colon \Sigma_\epsilon \times \Sigma_\epsilon \to \mathbb{R}_0^+$ repräsentiert die Gewichte der möglichen Ersetzungen.[2] $d(\epsilon, \text{„i“}) = 2$ bedeutet beispielsweise, dass das Einfügen des Buchstaben „i“ mit Kosten von 2 verbunden ist, $d(\text{„e“}, \text{„c“}) = 1$ bedeutet, dass der Buchstabe „e“ mit Kosten 1 in den Buchstaben „c“ umgewandelt werden kann. Die Funktion d kann beispielsweise als Tabelle gespeichert werden, in der zu jedem Buchstabenpaar die zugehörigen Kosten vermerkt sind.

Die gewichtete Editierdistanz $\delta_{\text{edit}'}$ zweier Zeichenketten s und t ist definiert als $\delta_{\text{edit}'}(s,t) := L_{|s|,|t|}$, wobei für L folgende Rekursionsgleichung gilt:

$$L_{0,0} = 0$$

$$L_{i,j} = \min \begin{cases} L_{i-1,j-1} + 0 & \text{Übernehmen (wenn } s[i] = t[j]) \\ L_{i-1,j-1} + d(s[i], t[j]) & \text{Ersetzen (wenn } s[i] \neq t[j]) \\ L_{i,j-1} + d(\epsilon, t[j]) & \text{Einfügen} \\ L_{i-1,j} + d(s[i], \epsilon) & \text{Löschen} \end{cases}$$

Es wird gefordert, dass $d(u,u) = 0$ gilt und somit kann die Rekursionsgleichung vereinfacht werden zu:

$$L_{i,j} = \min \begin{cases} L_{i-1,j-1} + d(s[i], t[j]) & \text{Ersetzen} \\ L_{i,j-1} + d(\epsilon, t[j]) & \text{Einfügen} \\ L_{i-1,j} + d(s[i], \epsilon) & \text{Löschen} \end{cases}$$

Beispiel

Gegeben ist beispielsweise folgende Kostenfunktion für die Editieroperationen:

$$d(u,w) = \begin{cases} 0 & \text{für } u = w \\ \frac{1}{2} & u \text{ und } w \text{ beides Kleinbuchstaben} \\ 1 & \text{andernfalls} \end{cases}$$

$$d(\epsilon, u) = 2 \text{ (Einfügen)}$$

$$d(u, \epsilon) = 2 \text{ (Löschen)}$$

[2] Σ_ϵ bezeichnet die Menge $\Sigma \cup \{\epsilon\}$.

Die gewichtete Editierdistanz von „Berlin“ und „BelGien“ beträgt 3,5, weil beispielsweise folgende Transformation Kosten von 3,5 hat:

- Übernehmen von „B“ (Kosten = 0)
- Übernehmen von „e“ (Kosten = 0)
- Ersetzen von „r“ durch „l“ (Kosten = $\frac{1}{2}$)
- Ersetzen von „l“ durch „G“ (Kosten = 1)
- Übernehmen von „i“ (Kosten = 0)
- Einfügen von „e“ (Kosten = 2)
- Übernehmen von „n“ (Kosten = 0)

Analyse

Die so definierte Abstandsfunktion $\delta_{\text{edit}'}$ ist nicht mehr automatisch symmetrisch. Symmetrie gilt genau dann, wenn die Funktion d symmetrisch ist. Im Gegensatz zu [Nav01, S. 37] bin ich jedoch der Meinung, dass die gewichtete Editierdistanz keine Metrik darstellt, weil die Dreiecksungleichung im Allgemeinen nicht erfüllt ist. Gegenbeispiel: Gegeben ist die Kostenfunktion d:

$$d(u,w) = \begin{cases} 0{,}1 & \text{für } u = \text{„H“}, w = \text{„f“} \\ 0{,}1 & \text{für } u = \text{„f“}, w = \text{„i“} \\ 1{,}0 & \text{andernfalls} \end{cases}$$

Der Abstand von „H“ zu „i“ ist dann größer als der Abstand von „H“ zu „f“ plus der Abstand von „f“ zu „i“:

$$\begin{aligned} \delta_{\text{edit}'}(\text{„H“}, \text{„i“}) &\leq \delta_{\text{edit}'}(\text{„H“}, \text{„f“}) + \delta_{\text{edit}'}(\text{„f“}, \text{„i“}) \\ \Leftrightarrow 1{,}0 &\overset{\lightning}{\leq} 0{,}1 + 0{,}1 \end{aligned}$$

□

Die Editierdistanz mit Gewichten ist folgendermaßen beschränkt:

$$0 \leq \delta_{\text{edit}'}(s,t) \leq \max(|s|,|t|) \cdot \max_{u,w \in \Sigma_\epsilon} \{d(u,w)\}$$

Die Laufzeit der Berechnung von $\delta_{\text{edit}'}(s,t)$ ist $O(|s| \cdot |t|)$, weil in dieser Zeit alle Werte der Matrix L berechnet werden können.

Eignung

Die Editierdistanz mit Gewichten ist eine kanonische Verallgemeinerung der einfachen Editierdistanz, die unterschiedliche Kosten für jede einzelne Operation erlaubt. Dies ist in diesem Kontext eine sinnvolle Erweiterung, weil somit häufige Rechtschreib-, Tipp- und OCR-Fehler direkt im Zeichenketten-Abstandsmaß modelliert werden können. Ein weiterer Vorteil dieses Ähnlichkeitsmaßes ist die Verwendung einer flexiblen Gewichtsfunktion. Die Gewichte für die einzelnen Operationen könnten mit der Zeit gelernt werden, um dadurch häufige Fehler noch besser korrigieren zu können.

4.2.5. Editierdistanz mit erweiterten Operationen

Definition

Die Editierdistanz mit erweiterten Operationen $\delta_{\text{edit}''}$ ist eine noch weiter verallgemeinerte Variante der Editierdistanz. Dabei werden nicht nur Operationen zugelassen, die jeweils *einen* Buchstaben verändern, sondern auch solche, die sich auf beliebige Teilzeichenketten beziehen. Die erlaubten Operationen sind als Menge von Regeln $R \subset \Sigma^* \times \Sigma^*$ gegeben. Die Funktion $d\colon \Sigma^* \times \Sigma^* \to \mathbb{R}_0^+$ repräsentiert wieder die Gewichte der möglichen Ersetzungen.

Die erweiterte Editierdistanz $\delta_{\text{edit}''}$ zweier Zeichenketten s und t ist definiert als $\delta_{\text{edit}''}(s,t) := L_{|s|,|t|}$, wobei für L folgende Rekursionsgleichung gilt [Nav01, S. 16, 17]:

$$L_{i,j} = \min_{(u,w)\in R \text{ mit } s[1..i]=s'u \wedge t[1..i]=t'w} L_{i-|u|,j-|w|} + d(u,w)$$

Beispiel

Gegeben ist beispielsweise die Regelmenge $R = \{(u,u) \mid u \in \Sigma\} \cup \{(\text{„rl“}, \text{„d“})\}$ und folgende Gewichtsfunktion:

$$d(u,w) = \begin{cases} 0 & \text{für } u = w \\ \frac{1}{2} & \text{für } u = \text{„rl“}, w = \text{„d“} \end{cases}$$

Die Distanz $\delta_{\text{edit}''}(s,t)$ mit $s =$ „Berlin“ zu $t =$ „Bedin“ ist damit 4,5, weil mit folgenden Operationen die Zeichenkette s in t überführt werden kann. (Es gibt keine Folge von Operationen mit geringeren Kosten.)

- Übernehmen von „B“ (Regel 1, Kosten = 0)
- Übernehmen von „e“ (Regel 1, Kosten = 0)
- Ersetzen von „rl“ durch „d“ (Regel 2, Kosten = $\frac{1}{2}$)
- Übernehmen von „i“ (Regel 1, Kosten = 0)
- Übernehmen von „n“ (Regel 1, Kosten = 0)

Analyse

Die so definierte Abstandsfunktion $\delta_{\text{edit}''}$ ist im Allgemeinen weder symmetrisch noch erfüllt sie die Dreiecksungleichung und ist daher keine Metrik.

Die Laufzeit eines naiven Algorithmus' für die Berechnung von $\delta_{\text{edit}''}(s,t)$ ist $O(|s| \cdot |t| \cdot |R|)$, wobei $|R|$ die Summe der Längen der Zeichenketten in R ist [Nav01, S. 16, 17].

Eignung

Die erweiterte Editierdistanz ist auf Grund ihrer Allgemeinheit sehr flexibel einsetzbar. So kann beispielsweise mit entsprechenden Ersetzungsregeln auch die Transposition (Vertauschung) zweier aufeinander folgender Buchstaben modelliert werden. Dadurch ist sie gut geeignet, um Rechtschreib- und Tippfehler abzubilden. Auch OCR-Fehler lassen sich gut mit Hilfe der erweiterten Editierdistanz modellieren, weil beispielsweise auch Verschmelzungen

mehrerer aufeinander folgender Buchstaben dargestellt werden können (z. B. „rn“ → „m“). Um Verschmelzungen und Aufspaltungen von Zeichen zu modellieren gibt es im Allgemeinen Fall folgende Regeln:

$$R = \left\{(u \to w) \mid u \in \Sigma^2, w \in \Sigma\right\} \cup \left\{(w \to u) \mid u \in \Sigma^2, w \in \Sigma\right\}$$

Auch Synonyme könnten mit Hilfe der erweiterten Operationen behandelt werden, indem z. B. eine Regel „Dahlem“ → „Berlin“ eingefügt wird für den Fall, dass „Dahlem“ ein Synonym für „Berlin“ sein soll. Die Berechnung der erweiterten Editierdistanz ist jedoch recht aufwändig und die Laufzeit wächst linear mit der Anzahl und der Länge der Ersetzungsregeln. Dadurch ist es nicht effizient möglich alle Synonyme in dieser Art zu modellieren.[3]

4.2.6. Jaro-Ähnlichkeit

Definition

Das Jaro-Maß σ_{jaro} ist im Gegensatz zu den bisher betrachteten Maßen keine Abstands- sondern eine eine Ähnlichkeitsfunktion. Dabei werden in beiden Zeichenketten die Zeichen gezählt, zu denen es in der anderen Zeichenkette eine Entsprechung gibt.

Die Jaro-Ähnlichkeit ist in [CRF03] folgendermaßen definiert (aufbauend auf [Jar89]). Gegeben sind zwei Zeichenketten s und t. Ein Zeichen $s[i]$ heißt *passend*, wenn es ein $t[j] = s[i]$ gibt mit $|j - i| \leq l$, $l := \frac{1}{2}\min(|s|, |t|)$. $\bar{s}$ bezeichnet die Folge der passenden Zeichen aus s, wobei jedes Zeichen aus t nur einmal als Entsprechung vorkommen darf. Die Folge der passenden Zeichen in t ist analog als $\bar{t}$ definiert. (Anmerkung: $\bar{s}$ und $\bar{t}$ enthalten somit immer gleich viele Zeichen: $|\bar{s}| = |\bar{t}|$). Die Ähnlichkeit der Zeichenketten s und t lässt sich folgendermaßen berechnen:

$$\sigma_{\text{jaro}}(s,t) = \frac{1}{3} \cdot \left(\frac{|\bar{s}|}{|s|} + \frac{|\bar{t}|}{|t|} + \frac{|\bar{s}| - \frac{z}{2}}{|\bar{s}|} \right)$$

Dabei bezeichnet z die Anzahl der Stellen, an denen sich $\bar{s}$ und $\bar{t}$ unterscheiden, die Zeichen also in einer anderen Reihenfolge auftreten (sogenannte Transpositionen).

Beispiel

Um die Jaro-Ähnlichkeit von „Berlin“ und „Ballerina“ zu ermitteln, müssen also die Werte l, $\bar{s}$, $\bar{t}$ und z bestimmt werden. $l = \frac{1}{2}\min(|s|, |t|) = \frac{\max\{6,9\}}{2} = 4{,}5$.

Zur Bestimmung von $\bar{s}$ und $\bar{t}$ müssen jene Buchstaben von „Berlin“ ermittelt werden, zu denen es ein passendes Zeichen in „Ballerina“ gibt, das nicht weiter als $l = 4{,}5$ entfernt ist. Dies ist für alle Buchstaben von „Berlin“ der Fall, somit ist $\bar{s}$ = „Berlin“. Für $\bar{t}$ ergibt sich die Folge „Blerin“, weil es für die „a“s und das zweite „l“ keine Entsprechung in „Berlin“ gibt.

Um z zu bestimmen, muss die Anzahl der Stellen gezählt werden, an denen in $\bar{s}$ und $\bar{t}$ verschiedene Zeichen stehen:

```
z = 0 + 1 + 1 + 1 + 0 + 0 = 3
    B   e   r   l   i   n
    B   l   e   r   i   n
```

[3] Bei der Adresskorrektur gibt es beispielsweise für Ortsnamen insgesamt viele Tausend Synonyme.

$$\sigma_{\text{jaro}}(\text{„Berlin“}, \text{„Ballerina“}) = \frac{1}{3} \cdot \left(\frac{|\bar{s}|}{|s|} + \frac{|\bar{t}|}{|t|} + \frac{|\bar{s}| - \frac{z}{2}}{|\bar{s}|} \right) = \frac{1}{3} \cdot \left(\frac{6}{6} + \frac{6}{9} + \frac{6 - \frac{3}{2}}{6} \right) \approx 0{,}81$$

Analyse

Das Jaro-Ähnlichkeitsmaß σ_{jaro} ist symmetrisch. Weil jeder der drei Summanden eine Zahl zwischen 0 und 1 ist, gilt außerdem:

$$0 \leq \sigma_{\text{jaro}}(s, t) \leq 1$$

Das Jaro-Maß ist ein Ähnlichkeitsmaß, kein Abstandsmaß und ist daher auch keine Metrik.

Die Laufzeit der Berechnung von $\sigma_{\text{jaro}}(s, t)$ ist $O(|s| \cdot |t|)$, weil in linearer Zeit für jedes Zeichen der einen Zeichenkette die passenden Zeichen der anderen Zeichenkette bestimmen werden können.

Eignung

Das Jaro-Maß modelliert Rechtschreib-, Tipp- und OCR-Fehler nicht so direkt wie beispielsweise die Editierdistanz, liefert aber beispielsweise bei der Datensatzverknüpfung, also dem Suchen nach Duplikaten, sehr gute Ergebnisse (siehe [CRF03], [Bud91]).

Eine Erweiterung des Jaro-Maßes stammt von McLaughlin [PW97]. Dabei werden unterschiedliche aber in gewisser Weise ähnliche Zeichen nur mit einem Faktor von 0,3 gewichtet. Die Ähnlichkeit zweier Zeichen kann dabei kontextabhängig definiert werden: beispielsweise für Tippfehler basierend auf dem Abstand auf der Tastatur, oder für OCR-Fehler basierend auf der optischen Ähnlichkeit der Zeichen. Es ist auch denkbar diesen Wert mit Hilfe einer Trainingsmenge zu lernen.

4.2.7. Phonetische Verfahren

Definition

Es existieren eine Reihe phonetischer Verfahren, die einer Zeichenkette eine Kodierung zuordnen, die die Aussprache des Wortes repräsentieren soll. Ähnlich klingende Wörter sollen dabei die gleiche Kodierung erhalten. Ein solches phonetisches Verfahren stellt dabei keine Ähnlichkeits- oder Abstandsfunktion dar, sondern bestimmt eine Menge von Äquivalenzklassen, d. h. Mengen ähnlich klingender Wörter.

Beispielhaft wird hier der relativ einfache Soundex-Algorithmus vorgestellt, der eine Kodierung berechnet, die auf der (englischen) Aussprache beruht (vorgeschlagen in [Rus18]). Die Soundex-Kodierung eines Wortes besteht aus seinem Anfangsbuchstaben, sowie drei Ziffern. Die Ziffern repräsentieren die Konsonanten des Wortes; Vokale sowie die Buchstaben „H“, „W“ und „Y“ werden bei der Kodierung nicht betrachtet. Dabei wird jedem Konsonanten nach Tabelle 4.2 eine Ziffer zugeordnet; einige Konsonanten werden dabei auf die gleiche Ziffer abgebildet. Direkt aufeinander folgende Konsonanten mit der gleichen Kodierung werden dabei nur einmal verwendet.

Der Soundex-Algorithmus ordnet jeder Zeichenkette eine Kodierung zu; die zugehörige Abstandsfunktion ordnet zwei Zeichenketten eine 0 oder eine 1 zu, je nachdem, ob sie die

1	B, P, F, V
2	C, G, J, K, Q, X, Z
3	D, T
4	L
5	M, N
6	R

Tabelle 4.2.: Kodierung der Buchstaben beim SOUNDEX-Algorithmus

gleiche Kodierung haben oder nicht:

$$\delta_{\text{soundex}}(s,t) = \begin{cases} 0 & \text{falls } \text{SOUNDEX}(s) = \text{SOUNDEX}(t) \\ 1 & \text{sonst} \end{cases}$$

Beispiel

Die SOUNDEX-Kodierung von „Berlin“ ist „B645“, die von „Belgien“ ist „B425“. Weil die beiden Wörter eine unterschiedliche Kodierung haben, gelten sie im Sinne von SOUNDEX als *nicht ähnlich*. „Berolina“ hat hingegen die gleiche Kodierung wie „Berlin“ und ist damit in der gleichen Äquivalenzklasse.

Analyse

Weil die SOUNDEX-Kodierung alle Zeichenketten in Äquivalenzklassen einteilt, ist δ_{soundex} symmetrisch. Es werden jedoch nur die Werte 0 und 1 angenommen, so dass eine Betrachtung der Dreiecksungleichung hier nicht sinnvoll ist.

Die Laufzeit der Berechnung von $\delta_{\text{soundex}}(s,t)$ ist $O(|s| + |t|)$, weil in dieser Zeit für beide Zeichenketten die Kodierung bestimmt und verglichen werden kann.

Eignung

Weitere phonetische Verfahren neben SOUNDEX sind beispielsweise METAPHONE, DOUBLE METAPHONE [Phi00] und NYSIIS [Taf70]. Die zugehörigen Algorithmen sind etwas komplexer und liefern in der Praxis bessere Ergebnisse. Phonetische Verfahren orientieren sich allerdings sehr stark an der Aussprache der Wörter und sind sprachspezifisch [Bra05]. Deshalb eignen sie sich besonders gut, wenn Fehler vor allem auf Grund von ähnlicher Aussprache entstehen (z. B. bei der Schreibung von Eigennamen). Sie klassifizieren zwei Wörter als *gleich* bezüglich der Aussprache, beziehungsweise als *nicht gleich* und stellen jeweils keine Ähnlichkeitsfunktion dar. Die Fehler, die bei der optischen Zeichenerkennung auftreten, werden von phonetischen Verfahren nicht gut modelliert.

4.2.8. k-Gramme

Definition

Ein anderer Ansatz ist die Verwendung von k-Grammen, wobei $k \in \mathbb{N}$ eine Konstante ist (üblicherweise eine kleine natürliche Zahl). Ein k-Gramm einer Zeichenkette ist eine Teilzeichenkette mit genau k Buchstaben [Ukk92; Kuk92]. Die Menge der 2-Gramme von „Berlin“ ist beispielsweise {„Be“, „er“, „rl“, „li“, „in“}. Um die Ähnlichkeit von zwei Zeichenketten s

und t zu bestimmen, wird jeder Zeichenkette zunächst die Menge G_s beziehungsweise G_b der k-Gramme zugeordnet und die Ähnlichkeit dieser beiden Mengen mit einem beliebigen geeigneten Maß berechnet. Beispielsweise kann eine solche Menge G_s beziehungsweise G_t als Σ^k-dimensionaler 0-1-Vektor v_s beziehungsweise v_t repräsentiert werden, der genau an den Stellen eine 1 hat, die zu den enthaltenen k-Grammen gehören. Die k-Gramm-Ähnlichkeit von zwei Zeichenketten s und t kann darauf aufbauend beispielsweise folgendermaßen definiert werden [Kuk92; BHS07]:

- Kosinus-Ähnlichkeit $\sigma_{\text{kosi}}(s,t) = \frac{\langle v_s, v_t \rangle}{|v_s| \cdot |v_t|}$
- Jaccard-Ähnlichkeit $\sigma_{\text{jacc}}(s,t) = \frac{|G_s \cap G_t|}{|G_s \cup G_t|}$
- Dice-Ähnlichkeit $\sigma_{\text{dice}}(s,t) = \frac{2 \cdot |G_s \cap G_t|}{|G_s| + |G_t|}$

Beispiel

Für die Zeichenketten $s =$ „Berlin“, $t =$ „Belgien“ und $k = 2$ ergeben sich folgende Mengen:

$$G_s = \{\text{„Be“}, \text{„er“}, \text{„rl“}, \text{„li“}, \text{„in“}\} \text{ und}$$
$$G_t = \{\text{„Be“}, \text{„el“}, \text{„lg“}, \text{„ gi“}, \text{„ie“}, \text{„en“}\}.$$

Die Ähnlichkeit berechnet sich damit für die drei angegebenen Varianten folgendermaßen:

- Kosinus-Ähnlichkeit $\sigma_{\text{kosi}}(s,t) = \frac{\langle v_s, v_t \rangle}{|v_s| \cdot |v_t|} = \frac{1}{5 \cdot 6} = \frac{1}{30} \approx 0{,}03$
- Jaccard-Ähnlichkeit $\sigma_{\text{jacc}}(s,t) = \frac{|\{\text{„Be“}\}|}{|\{\text{„Be“}, \text{„er“}, \text{„rl“}, \text{„li“}, \text{„in“}, \text{„el“}, \text{„lg“}, \text{„ gi“}, \text{„ie“}, \text{„en“}\}|} = \frac{1}{10} = 0{,}1$
- Dice-Ähnlichkeit $\sigma_{\text{dice}}(s,t) = \frac{2 \cdot |G_s \cap G_t|}{|G_s| + |G_t|} = \frac{2 \cdot 1}{5+6} = \frac{2}{11} \approx 0{,}18$

Analyse

Die k-Gramm-Ähnlichkeit ist genau dann ein symmetrisch, wenn die zu Grunde liegende Ähnlichkeitsfunktion für Mengen symmetrisch ist. Dies ist für die vier oben genannten Funktionen der Fall. Sie sind jeweils Ähnlichkeitsmaße und daher keine Metriken.

Die Laufzeit der Berechnung von $\delta_{\text{kG}}(s,t)$ ist $O(|s| + |t|)$, weil für beide Zeichenketten in linearer Zeit die Menge der k-Gramme bestimmt werden kann. Die Berechnung der Ähnlichkeit zwischen den k-Gramm-Mengen benötigt für alle oben angegebenen Maße ebenfalls nur lineare Zeit.

Eignung

Die Verwendung von k-Grammen zur Bestimmung der Ähnlichkeit von zwei Zeichenketten ist deutlich weniger anschaulich als beispielsweise die Editierdistanz. Daher besteht die Gefahr, dass bei Verwendung dieses Ansatzes Zeichenketten als ähnlich gelten, die im intuitiven Sinne nicht ähnlich sind. Die bei der Editierdistanz modellierten Fehler wirken sich jedoch alle nur lokal aus, so dass sich auch die Menge der k-Gramme bei wenigen Editieroperationen nicht all zu stark ändert. Eine Ersetzung eines Buchstabens durch einen anderen wirkt sich beispielsweise höchstens auf k der k-Gramme aus (bei $k = 3$ wären also höchstens 3 Einträge des Vektors verändert). In [Ukk92] wird eine Verbindung zwischen der k-Gramm-Distanz und der Editierdistanz hergestellt.

Durch die Abbildung der Zeichenketten auf Vektoren ergibt sich die Möglichkeit, einen Index zu erstellen, in dem effizient ähnliche Zeichenketten gesucht werden können. Dieser Index kann beispielsweise dazu verwendet werden, eine Kandidatenmenge von Ergebnissen zu bestimmen, welche in einem zweiten Schritt mit Hilfe einer auf das konkrete Problem zugeschnittenen Ähnlichkeitsfunktion sortiert werden.

Diese drei vorgestellten Varianten haben unterschiedliche Charakteristika bezüglich der Bestimmung der Ähnlichkeit von Zeichenketten. Der Kosinus-Abstand eignet sich nach [Kuk92] am besten bei der Rechtschreibkorrektur zur Suche in einem Wörterbuch.

4.2.9. Zusammenfassung

In den vorangehenden Abschnitten wurden verschiedene Ähnlichkeits- und Abstandsmaße für Zeichenketten erläutert und die Eignung für die Anwendung bei der Adresskorrektur analysiert. Die Ergebnisse sind in Tabelle 4.3 zusammengefasst. Dabei ist für jede Fehlerart angegeben, ob sie mit Hilfe des jeweiligen Ähnlichkeitsmaßes gut modelliert (●), teilweise modelliert (◐) oder nicht modelliert (○) werden kann. Diese Abstufung ist jedoch nicht sehr fein und kann nur einen groben Überblick geben. Die Spalte *erlernbar* gibt an, ob das Ähnlichkeitsmaß mit einer Trainingsmenge trainiert und damit verbessert werden kann. Die Spalte *Laufzeit* gibt die Laufzeit zur Berechnung des Ähnlichkeitsmaßes im ungünstigsten Fall (engl.: worst case) an. Dabei ist hier aber insbesondere auch die durch die asymptotische Schreibweise verborgene Konstante wichtig, weil die Zeichenketten in der Regel nicht länger als 30 Zeichen sind.

	Synonyme	Rechtschreib.	Tipp-Fehler	OCR-Fehler	Erlernbar	Laufzeit
Hamming-Distanz	○	◐	◐	◐	○	$\|s\| + \|t\|$
Schreibmaschinend.	○	◐	◐	◐	○	$\|s\| + \|t\|$
Editierdistanz	○	◐	◐	◐	○	$\|s\| \cdot \|t\|$
Editierd. m. Gewichten	○	●	●	●	●	$\|s\| \cdot \|t\|$
Editierd. m. erw. Op.	◐	●	●	●	●	$\|s\| \cdot \|t\| \cdot \|R\|$
Jaro-Distanz	○	●	●	◐	●	$\|s\| \cdot \|t\|$
Phonetische Verfahren	○	●	◐	○	○	$\|s\| + \|t\|$
k-Gramme	○	◐	◐	◐	○	$\|s\| + \|t\|$

Tabelle 4.3.: Ähnlichkeitsmaße

Die *Hamming-* und die *Schreibmaschinendistanz* sind sehr simpel und können keinen Einfügungen, Löschungen, Verschmelzungen und Aufspaltungen von Buchstaben modellieren. Sie sind daher nicht gut für die Modellierung von Rechtschreib-, Tipp- und OCR-Fehlern geeignet. Die *phonetischen Verfahren* eignen sich nur dann, wenn die Aussprache der Wörter nicht verändert wird und sind deshalb nicht bei OCR-Fehlern anwendbar. Diese drei Ähnlichkeitsmaße kommen daher für eine Verwendung bei der Adresskorrektur nicht in Frage.

Das *Jaro-Ähnlichkeitsmaß* kann die hier zu erwartenden Fehler gut modellieren. Mir ist jedoch kein Verfahren zur Indizierung einer Menge von Zeichenketten unter Verwendung der Jaro-Ähnlichkeit bekannt.

Das *k-Gramm*-Ähnlichkeitsmaß ist mit Hilfe von invertierten Listen besonders gut indizierbar [Ukk92]. Es bietet jedoch keine Möglichkeit, die Häufigkeiten der Fehler zu erlernen und so die Qualität zu verbessern.

Die Gruppe der *Editierdistanzen* eignet sich sehr gut zur Modellierung von Rechtschreib-, Tipp- und OCR-Fehlern, weil die Fehler direkt modelliert werden können. Diese Operationen entsprechen genau den Fehlern, die in Abschnitt 2.5.2 angesprochen wurden. Auch beispielsweise Verschmelzungen und Vertauschungen von Buchstaben können abgebildet werden. Bei der Verwendung einer gewichteten Editierdistanz können den Operationen verschiedene Gewichte zugeordnet werden, wodurch auch die Häufigkeiten der Fehler modelliert werden können.

Die Editierdistanz wird sowohl bei der Datensatzverknüpfung, als auch bei der Zeichenkettensuche, der Rechtschreibkorrektur und der OCR-Nachkorrektur vielfach empfohlen und eingesetzt [BM03; ZD95; WM91; BYN98; NBYST01; CN02; BCP02; KCG90; RY98; BM00; CB04; MS04; BHS07; TE96; KR02; LH03; Rin03; Sch03; LW75] (siehe auch Kapitel 3).

Eine Menge von Zeichenketten kann mit Hilfe eines *Tries* bezüglich der Editierdistanz indiziert werden (siehe dazu auch Abschnitt 5.2). Auf Grund der genannten Vorteile verwende ich in dieser Arbeit die Editierdistanz. Auf eine selbst durchgeführte experimentelle Evaluation der Eignung der Ähnlichkeitsmaße wird hier verzichtet. Die einfache Editierdistanz wird zur Suche im Index und eine gewichtete Editierdistanz mit erweiterten Operationen bei der anschließenden Sortierung der Ergebniskandidaten eingesetzt werden. Als erweiterte Operationen wird hier das Aufteilen und die Verschmelzung und benachbarten Buchstaben implementiert. Diese Operationen treten relativ häufig bei der optischen Zeichenerkennung auf (siehe Abschnitt 8.3). Die Gewichte der Editieroperationen werden mit Hilfe einer Trainingsmenge erlernt, um auch die Häufigkeiten der Fehler abbilden zu können (Abschnitt 7.1).

Bemerkung Bei der Editierdistanz darf jeder Buchstabe nur einmal verändert werden. Dies ist eine kleine Einschränkung, weil in der Realität die Ersetzungen nacheinander ausgeführt werden: Beispielsweise kann zunächst ein Tippfehler geschehen und dieser falsch getippte Buchstabe wird anschließend von einer OCR-Software nicht korrekt erkannt. Es wäre wünschenswert, auch diese kombiniert hintereinander auftretenden Ersetzungen modellieren zu können. Eine solche Editierdistanz wäre dann jedoch nicht mehr effizient berechenbar, im Allgemeinen sogar nicht mehr berechenbar [Nav01, S. 37]. Weil zu erwarten ist, dass die Systematik solcher kombinierten Fehler auch nur schwer erkennbar ist, wird hier darauf verzichtet.

4.3. Ähnlichkeit von Datensätzen

In diesem Abschnitt wird untersucht, wie die Ähnlichkeit von Datensätzen definiert werden kann. Wünschenswert ist ein Verfahren, das flexible Gewichtungen der Datensatz-Attribute zulässt, aber nicht aufwändig manuell konfiguriert werden muss.

Ein Ähnlichkeitsmaß für Adressen sollte diese beispielsweise als sehr ähnlich bewerten, wenn sie in den Attribute *ort*, *plz* und *str* jeweils sehr ähnlich sind. Eine Abweichung von einem Zeichen beim Attribut *plz* sollte dabei jedoch mit höheren Kosten verbunden sein, als eine Abweichung von einem Zeichen beim Attribut *str*.

In Abschnitt 3.1.1 wurden im Rahmen der Datensatzverknüpfung Methoden zur Definition der Klassifikation von Datensätzen als Duplikate erläutert. Ein Ansatz verwendet Entscheidungsbäume, die zwei gegebene Datensätzen als *ähnlich* oder *nicht ähnlich* klassifizieren. Diese werden in [CCGK07] mit Hilfe einer Trainingsmenge von manuell klassifizierten Datensätzen trainiert. Anschließend muss ein Experte die Konfiguration des Entscheidungsbaumes

übernehmen, um gute Resultate zu erhalten. Dieses Verfahren erlaubt jedoch ausschließlich die binäre Klassifikation der Datensätze und ist nicht in der Lage die Ähnlichkeit von Datensätzen in Form eines Ähnlichkeits- oder Abstandsmaßes zu bestimmen.

In dieser Arbeit wird daher ein anderes Verfahren verwendet, das ebenfalls bei der Datensatzverknüpfung zum Einsatz kommt [BM02; Man07] und gute Ergebnisse erzielt [BM03]. Die Ähnlichkeit von Datensätzen wird dabei auf die Klassifikation von Vektoren im euklidischen Raum zurückführt. Diese Klassifikation geschieht mit Hilfe einer Klassifikationsfunktion die von einer *Support Vector Machines (SVM)* anhand einer Trainingsmenge erlernt wird. Dies wird in Abschnitt 7.2 genauer beschrieben. Im Folgenden wird zunächst ein einfacherer Ansatz als Motivation und dann die Abbildung von Datensatzpaaren auf Vektoren erläutert.

4.3.1. Einfacher Ansatz

Dieser Abschnitt beschreibt einen einfachen Ansatz zur Definition der Ähnlichkeit von Datensätzen. Dies wird anschließend verallgemeinert. Der Abstand zweier Datensätze basiert dabei auf dem paarweisen Abstand der Attribute. Die Gewichtung der einzelnen Attribute wird dabei von einer Funktion $f\colon \mathbb{R}^m \to \mathbb{R}$ bestimmt:

$$\delta_{\text{einfach}}(q, r) = f\left(\delta_{\text{einfach}}(q.a_1, r.a_1), \ldots, \delta_{\text{einfach}}(q.a_m, r.a_m)\right)$$

Eine sinnvolle Wahl für die Funktion f ist beispielsweise die gewichtete Summe der Argumente (mit $g_i \in \mathbb{R}$):

$$f(x_1, \ldots, x_m) = g_1 \cdot x_1 + \ldots + g_m \cdot x_m$$

Die Wahl der Parameter g_i erfordert jedoch einen Experten, der den einzelnen Attributen sinnvolle Gewichtungen zuweist. Beim folgenden allgemeineren Ansatz können diese Gewichte mit Hilfe einer Trainingsmenge erlernt werden.

4.3.2. Rückführung auf Klassifikationsproblem

In diesem Abschnitt wird beschrieben wie die Definition der Ähnlichkeit von zwei Datensätzen q und r auf das Problem der Klassifikation von Datensatzpaaren (q, r) als *ähnlich* beziehungsweise *nicht ähnlich* zurückgeführt werden kann.

Allgemeines Klassifikationsproblem

Beim binären Klassifikationsproblem von Vektoren gibt es die Klassen $+1$ und -1 und eine Menge von Trainingsvektoren, die jeweils einer der beiden Klassen zugeordnet sind. Für einen unbekannten Testvektor x_i soll bestimmt werden, in welcher der beiden Klassen $c(x_i)$ er ist. Dies geschieht bei Support Vector Machines beispielsweise mit Hilfe einer Funktion $f\colon \mathbb{R}^m \to \mathbb{R}$, die einen Vektor auf eine reelle Zahl abbildet. Diese Zahl soll positiv sein, wenn der Vektor x_i zur Klasse $+1$, und negativ, wenn er zur Klasse -1 gehört: $c(x_i) = \operatorname{sgn}(f(x_i))$. Bei Support Vector Machines gilt darüber hinaus, dass ein höherer Betrag des Funktionswertes $f(x_i)$ eine höhere Sicherheit der Zugehörigkeit des Vektors x_i zu dieser Klasse bedeutet.

Anwendung

Dieses Klassifikationsverfahren kann auch zur Bestimmung der Ähnlichkeit von Datensätzen eingesetzt werden. Die Objekte sind Paare von Datensätzen und die Klassen $+1$ und -1

bedeuten, dass die beiden Datensätze *ähnlich* beziehungsweise *nicht ähnlich* sind. Um ein solches Klassifikationsverfahren anwenden zu können, müssen die Datensatzpaare noch auf Vektoren abgebildet werden.

Ein Datensatzpaar (q, r) wird mit Hilfe der Funktion $v_\delta \colon U \times U \to \mathbb{R}^m$ auf einen Vektor abgebildet. Ich habe mich dazu entschieden, diese Abbildung ähnlich wie in [BM03] durchzuführen. Die Komponenten dieses Vektors sind dabei die attributweisen Abstände der beiden Datensätze[4] (siehe auch Abbildung 4.2):

$$v_\delta(q, r) = \begin{pmatrix} \delta\left(q.a_1, r.a_1\right) \\ \vdots \\ \delta\left(q.a_m, r.a_m\right) \end{pmatrix}$$

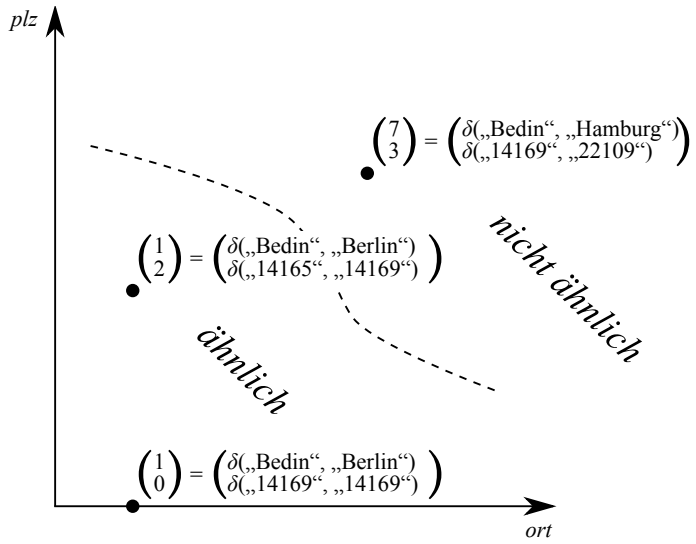

Abbildung 4.2.: Abbildung von Datensatzpaaren auf Vektoren

Die Entscheidungsfunktion f der Klassifizierung kann mit Hilfe einer Trainingsmenge erlernt werden. Dies wird in Abschnitt 7.2 genauer beschrieben.

Um ausgehend von einer solchen Entscheidungsfunktion den Abstand eines Datensatzpaares (q, r) zu bestimmen, wird der zugehörige Vektor $v_\delta\left(q, r\right)$ konstruiert und darauf die Funktion f angewendet. Der resultierende Wert kann als Zugehörigkeitsmaß zu den Klassen *ähnlich* beziehungsweise *nicht ähnlich* interpretiert werden. Er kann daher als Abstandsmaß verwendet werden:[5]

$$\delta_{\text{svm}}(q, r) = f\left(v_\delta\left(q, r\right)\right) + z$$

[4] Diese Art der Abbildung von Datensatzpaaren auf Vektoren ist nicht zwingend. Es wäre auch denkbar, andere Komponenten zu verwenden.

[5] Dabei wird eine genügend große Konstante $z \in \mathbb{R}^+$ addiert, um ausschließlich positive Werte zu erhalten.

4.3.3. Zusammenfassung

Als Abstandsfunktion für Datensätze wurden hier zwei Verfahren in Erwägung gezogen. Zum einen die gewichtete Summe der attributweisen Abstände, die auf Grund der effizienten Berechenbarkeit in der Indexstruktur zur Generierung von Ergebniskandidaten verwendet werden wird (Kapitel 5). Zum anderen wurde der Abstand von Datensätzen auf ein Klassifikationsproblem mit trainierbarer Entscheidungsfunktion zurückgeführt. Um den Abstand von zwei Datensätzen zu berechnen, werden sie auf einen Vektor von reellen Zahlen abgebildet, der auf Zugehörigkeit zu den Klassen der ähnlichen beziehungsweise nicht ähnlichen Datensätze geprüft wird. Dieses Verfahren verspricht bessere Ergebnisse als der einfache Ansatz zu liefern und wird daher bei der Sortierung der Kandidaten eingesetzt werden. Das Training der Klassifikationsfunktion wird in Abschnitt 7.2 beschrieben.

4.4. Zusammenfassung

In diesem Kapitel wurden Ähnlichkeitsmaße für Zeichenketten und Datensätze erläutert. Davon ausgehend können Antworten auf die ersten beiden in der Einleitung (Abschnitt 1.4) aufgeworfenen Fragen gegeben werden:

1. *Wie kann die Ähnlichkeit von Zeichenketten definiert werden, so dass die Fehler (hier unter anderem die Fehler der optischen Zeichenerkennung) gut abgebildet werden?*

   Für Zeichenketten hat sich zur Modellierung der bei der Adresskorrektur zu erwartenden Fehler die Editierdistanz als geeignet herausgestellt. Sie ist mit Tries indizierbar und kann daher bei der Implementierung zur Erzeugung von Ergebniskandidaten eingesetzt werden. Bei der anschließenden Sortierung der Kandidaten kann die gewichtete Editierdistanz mit den erweiterten Operationen zum Verschmelzen und Auftrennen von Buchstaben eingesetzt werden. Sie modelliert sehr gut die bei der optischen Zeichenerkennung zu erwartenden Fehler, insbesondere wenn die Gewichte mit Hilfe einer Trainingsmenge erlernt werden.

2. *Wie kann die Ähnlichkeit von Datensätzen definiert werden, so dass die Fehler in den einzelnen Attributen der Datensätze gut abgebildet werden?*

   Ein Abstandsmaß für Datensätze, das eine unterschiedliche Gewichtung der Attribute erlaubt, ist die gewichtete Summe der attributweisen Abstände. Dieses ist effizient berechenbar und kann daher bei der Kandidatengenerierung eingesetzt werden. Flexibler ist hingegen das Ähnlichkeitsmaß, das auf einer Klassifizierungsfunktion basiert, die von einer Support Vector Machine mit Hilfe von Trainingsdaten erlernt werden kann.

Im folgenden Kapitel wird beschrieben, mit welchen Datenstrukturen die ähnlichsten Zeichenketten beziehungsweise Datensätze bezüglich der genannten Ähnlichkeitsmaße effizient gefunden werden können.

5. Indexstrukturen

In diesem Kapitel wird eine Datenstruktur präsentiert, mit der eine effiziente Ähnlichkeitssuche in einer Menge von Datensätzen möglich ist. Die Ähnlichkeit von Datensätzen (Abschnitt 4.3) wurde basierend auf der Ähnlichkeit der Attributwerte definiert, insbesondere der Ähnlichkeit von Zeichenketten (Abschnitt 4.2). In ähnlicher Weise basiert auch der hier von mir vorgeschlagene Index für Datensätze auf Indexstrukturen der einzelnen Attribute, wieder mit einem besonderen Augenmerk auf Zeichenketten (siehe Abbildung 5.1). [1]

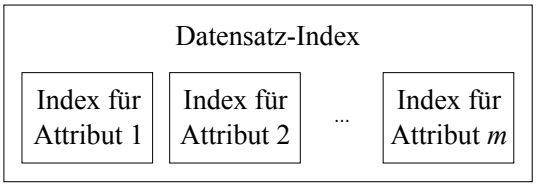

Abbildung 5.1.: Indexstrukturen-Architektur

In den folgenden Abschnitten werden zunächst die Rahmenbedingungen erläutert. Im Anschluss daran wird der *Trie* als Index für Zeichenketten dargestellt und der FRI als Index für Datensätzen vorgeschlagen.

5.1. Rahmenbedingungen

An ein Verfahren zum Suchen von ähnlichen Datensätzen werden verschiedene Anforderungen gestellt, die im folgenden Abschnitt erläutert werden. Anschließend werden begünstigende Gegebenheiten aufgeführt, die die Konstruktion eines Indexes gegebenenfalls erleichtern.

5.1.1. Anforderungen

Die Anforderungen beziehen sich auf die Laufzeit des Aufbaus des Indexes, den Speicherbedarf und vor allem die Laufzeit der Anfragebeantwortung. Damit Aktualisierungen der Indexstruktur möglich sind, sollte die Aufbauzeit des Indexes nicht zu hoch sein. Der Speicherbedarf sollte nur höchstens linear mit der Anzahl der Datensätze wachsen, um auch bei größeren Datenmengen anwendbar zu sein. Die Laufzeit einer Suchanfrage sublinear sein und möglichst wenig von der Anzahl der gespeicherten Datensätze abhängen, um eine effiziente Durchführung von Ähnlichkeitssuchen auch bei größeren Datenbanken zu gewährleisten.

[1] Diese Architektur ist nicht zwingend, es wäre auch ein Index denkbar, der nicht Attributwerte, sondern komplette Datensätze indiziert (z. B. metrische Bäume, siehe Abschnitt 3.2.1).

Adresskorrektur

Bei der Adresskorrektur stehen für die Suche der ähnlichsten Adresse auf Grund der Laufzeit der Briefe nur ungefähr 100 ms zur Verfügung (siehe Abschnitt 1.3). Die Indexstruktur sollte die Größe des Arbeitsspeichers (2 Gigabyte) nicht überschreiten, damit bei der Anfragebeantwortung keine Festplattenzugriffe notwendig sind. Solange der Index in den Arbeitsspeicher passt, spielt der Speicherbedarf bei der Adresskorrektur jedoch keine große Rolle. Für die Anwendung bei der Adresskorrektur sollte es möglich sein, für die Attributwerte auch Synonyme in der Indexstruktur verwalten zu können (siehe Abschnitt 2.5).

5.1.2. Begünstigende Gegebenheiten

Bestimmte Rahmenbedingungen können den Aufbau einer Indexstruktur zur Suche von ähnlichen Zeichenketten beziehungsweise Datensätzen begünstigen. So gibt es in der geschriebenen natürlichen Sprache in der Regel Gesetzmäßigkeiten, die ausgenutzt werden können (beispielsweise viele gemeinsame Präfixe der Wörter). Außerdem ist die Menge der natürlichsprachlichen Wörter im allgemeinen recht dünn besetzt.[2] So gibt es zu einem gegebenen Wort in der Regel nicht viele Wörter, die innerhalb eines kleinen Editierabstandes liegen (es gibt z. B. kein deutsches Wort s mit $\delta_{\text{edit}}(\text{„Kleeblatt“}, s) < 3$). Dadurch ist es möglicherweise leichter, das ähnlichste Wort zu einer gegebenen Zeichenkette zu finden.

Auch Datensätze haben oftmals einen relativ hohen Abstand und sind z. B. nicht durch die Änderung von nur einem Attributwert ineinander überführbar. Gegebenenfalls kann weiterhin ausgenutzt werden, dass manche Attribute in der Regel nicht verändert werden und beim Anfrage- und Ergebnisdatensatz übereinstimmen.

Adresskorrektur

Bei der Adresskorrektur kann ausgenutzt werden, dass alle korrekten Adressen bekannt sind, in der Sprechweise der Rechtschreibkorrektur also ein sogenanntes *perfektes Lexikon* vorliegt [Rin03, S. 91]. Weiterhin sind die Ortsnamen und auch die Straßennamen relativ dünn besetzt, so dass es möglicherweise einfacher ist, den korrekten Orts- und Straßennamen zu einer Anfrage-Adresse zu finden. Die Postleitzahlen in Deutschland sind relativ dicht besetzt (es gibt knapp 28.000 Postleitzahlen von 100.000 möglichen fünfstelligen Zahlen [Deu08]). Jedoch treten bei der Postleitzahl nur sehr wenig Fehler durch die optische Zeichenerkennung auf, weil dort die Menge der in Frage kommenden Zeichen kleiner ist. Ein Algorithmus zur Suche von ähnlichen Adressen könnte also davon profitieren, dass die Postleitzahl nur in geringem Maße fehlerhaft ist.

5.2. Indexstruktur für Zeichenketten: Trie

Wie in der Einleitung des Kapitels erwähnt, basiert die Indexstruktur für Datensätze auf Indexstrukturen der einzelnen Attribute. Diese sollen zu einem Anfragewert effizient die Suche nach den ähnlichsten Attributwerten unterstützen. Für numerische Daten und sonstige Daten, die bezüglich der jeweiligen Ähnlichkeitsfunktion linear geordnet werden können, gibt es eine Reihe verschiedener effizienter Indexstrukturen (z. B. B-Bäume [BM72]).

Zeichenketten lassen sich *nicht* bezüglich von Ähnlichkeitsmaßen wie der Editierdistanz linear anordnen und können deshalb auch nicht mit Hilfe solcher Strukturen indiziert werden.

[2] Eine Menge von Zeichenketten D heißt „dünn besetzt“, wenn ein Polynom p existiert, so dass für alle $j \in \mathbb{N}$ gilt: $|\{x \in D \text{ mit } |x| \leq j\}| \leq p(j)$ [HHR97]

Bei den verwandten Arbeiten in Abschnitt 3.2 wurden bereits einige für Zeichenketten in Frage kommende Indexstrukturen erwähnt.

Ich habe für die Implementierung *Tries* als Indexstrukturen für Zeichenketten gewählt. Tries bieten eine direkte Unterstützung der Ähnlichkeitssuche bezüglich der Editierdistanz. Dies ist beispielsweise ein großer Vorteil gegenüber der Verwendung einer Indizierung von k-Grammen mit inverierten Listen. Die Ähnlichkeitssuche ist bei Tries auch relativ effizient implementierbar, so dass sie deutlich schneller ist als etwa in metrischen Bäumen. Weil sie nur moderaten Speicherplatz benötigen, sind sie außerdem gut geeignet, wenn viele Zeichenketten verwaltet werden müssen.

In den folgenden Absätzen wird zunächst die Datenstruktur und dann ein Algorithmus zur Ähnlichkeitssuche vorgestellt. Außerdem wird der Zeit- und Platzbedarf des Verfahrens analysiert und anschließend eine Erweiterung zur Verbesserung der Suchlaufzeit erläutert (sogenannte *Ternary Search Tries*).

5.2.1. Datenstruktur

Ein Trie ist eine Datenstruktur zur Verwaltung einer Menge von Zeichenketten, gebildet über einem Alphabet Σ, in sortierter Reihenfolge. Tries bieten effiziente Unterstützung für die exakte sowie für die approximative Suche. Ein Trie besteht aus einer Baumstruktur, bei der jeder Knoten bis zu $|\Sigma|$ Kinder haben kann (siehe Abbildung 5.2). Der Wurzelknoten repräsentiert die leere Zeichenkette ϵ. Jede von einem Knoten ausgehende Kante steht für einen Buchstaben. Jeder Knoten repräsentiert implizit die Zeichenkette, die durch die Kanten auf dem Weg von der Wurzel bis zum Knoten gegeben ist. Die Blattknoten repräsentieren die tatsächlich gespeicherten Zeichenketten.[3]

Ein Trie kann als azyklischer deterministischer endlicher Automat (DEA, engl.: deterministic finite automaton, DFA) betrachtet werden. Dieser Automat akzeptiert genau die Wörter, die im Trie gespeichert sind [Rin03, S. 94].

In den Blattknoten können zu jeder Zeichenkette noch weitere Daten gespeichert sein, so dass Tries sich auch sehr gut für Abbildungen eignen, bei denen der Schlüssel eine Zeichenkette ist.

Die Implementierung eines Tries muss nicht zwangsläufig mit Hilfe von expliziten Verweisen auf die Kindknoten realisiert werden. In [Aoe89] wird eine Speicherung mit Hilfe von Arrays vorgeschlagen.

5.2.2. Algorithmus

Exakte Suche

Bei der exakten Suche soll bestimmt werden, ob eine Anfragezeichenkette in einem Trie enthalten ist. Gegebenenfalls sollen die mit der Zeichenkette verknüpften Daten zurückgegeben werden. Der Algorithmus zur einfachen Suche geht die Anfragezeichenkette Buchstabe für Buchstabe durch und steigt dabei im Baum Knoten für Knoten ab. Es wird immer der Kindknoten ausgewählt, der dem aktuellen Buchstaben entspricht. Wenn es keinen solchen Knoten gibt, ist die Zeichenkette nicht im Trie enthalten. Wenn das Ende der Anfragezeichenkette erreicht wurde, ist sie enthalten und die zugehörigen Daten können zurückgegeben werden.

[3] Damit auch Zeichenketten gespeichert werden können, die ein Präfix einer anderen gespeicherten Zeichenkette sind, werden alle Wörter mit einem zusätzlichen Zeichen „$“ abgeschlossen.

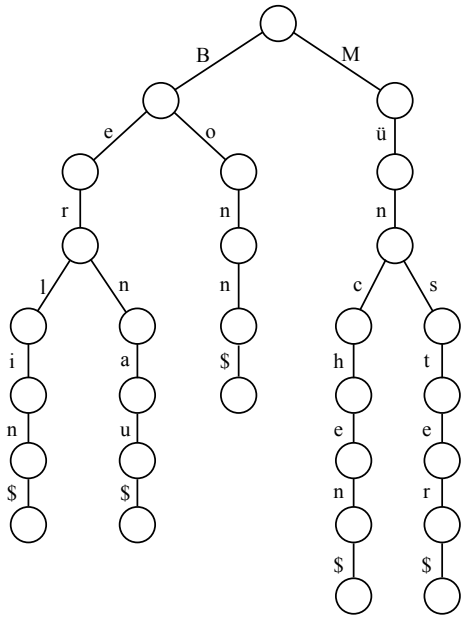

Abbildung 5.2.: Trie mit fünf Zeichenketten

Ähnlichkeitssuche

Bei der Ähnlichkeitssuche sollen in einem Trie zu einer Anfragezeichenkette alle Zeichenketten ermittelt werden, die innerhalb eines gewissen Editierabstands e liegen.[4] Eine solche Anfrage, bei der die Ergebnisse innerhalb einer gewissen Toleranz zur Anfrage liegen, wird als *Bereichs-Anfrage* bezeichnet. Gegebenenfalls sollen auch die jeweils zugehörigen Daten zurückgegeben werden. Diese Art der Suche wird von Tries unterstützt, dazu muss nur der Algorithmus der exakten Suche ein wenig modifiziert werden.

Beim Absteigen im Baum werden auch Knoten zugelassen, die nicht dem aktuellen Buchstaben der Anfragezeichenkette entsprechen, um Ersetzungen eines Buchstabens durch einen anderen zuzulassen. Zusätzlich wird das Lesen eines Buchstabens der Anfrage ohne gleichzeitiges Absteigen im Baum und umgekehrt auch das Absteigen im Baum ohne Lesen eines Buchstabens zugelassen, um auch Einfügungen und Löschungen von Buchstaben zu erlauben. Solche „falschen Abzweigungen“ werden so oft toleriert, bis die gegebene Suchtoleranz e erschöpft ist (siehe Abbildung 5.3). Alle so gefundenen Endknoten repräsentieren die Zeichenketten, deren Editierabstand $\leq e$ ist. Eine genauere Erläuterung dieses Suchalgorithmus' findet sich in [SM96]. Anstatt den Editierabstand für jede Zeichenkette einzeln zu berechnen, kann in Tries also der Editierabstand von gemeinsamen Präfixen zusammen berechnet werden.

Eine Top-k-Anfrage, also die Suche nach den k ähnlichsten Zeichenketten zu einer Anfragezeichenkette, kann in einem Trie ebenfalls durchgeführt werden. Dazu wird die Suchtoleranz so lange schrittweise erhöht, bis insgesamt k Ergebnisse gefunden wurden.[5]

[4] Als Ähnlichkeitsfunktion kann hier beispielsweise die einfache oder die gewichtete Editierdistanz verwendet werden.

[5] Der hier angegebene Algorithmus bezieht sich auf die einfache Editierdistanz und muss bei Verwendung einer Variante der Editierdistanz leicht angepasst werden.

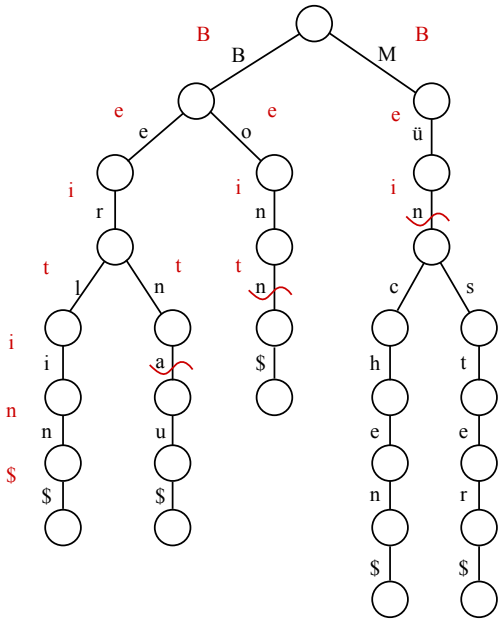

Abbildung 5.3.: Ähnlichkeitssuche in einem Trie mit q = „Beitin“ und $e = 2$

Algorithmus 5.1: Top-k-Anfrage in einem Trie

```
1 TrieSucheTopK(Trie T, q, k) begin
2       for e = 0 to ∞ do
3             ergebnis = TrieSuche(T, q, e)
4             if |ergebnis| >= k then
5                   return ergebnis
6       end for
7 end
```

5.2.3. Analyse

Speicher

Der Speicherbedarf von Tries in Abhängigkeit von der Anzahl der Zeichenketten n und der Länge der längsten Zeichenkette m ist $O(n \cdot m)$. Für jede Zeichenkette gibt es höchstens m neue Knoten. In der Praxis ist der Speicherbedarf von Tries jedoch geringer, weil gemeinsame Präfixe nur einmal abgespeichert werden. Wenn natürlichsprachliche Worte gespeichert werden, sind in der Tat viele Präfixe gemeinsam, so dass der Speicherplatz sublinear ist. Somit kann sich sogar eine Kompression der Daten ergeben [SM96]. Eine exakte Abschätzung ist jedoch nicht trivial und hängt unter anderem von der konkreten Verteilung der Daten ab (beispielsweise den Übergangswahrscheinlichkeiten der Buchstaben).

Aufbauzeit

Der Aufbau eines Tries mit n Zeichenketten der maximalen Länge $m_{\max}$ benötigt eine Laufzeit von $O(n \cdot m_{\max})$, weil das Einfügen einer Zeichenkette in $O(m)$ bewerkstelligt werden kann.

Anfragezeit

Die exakte Suche nach einer Zeichenkette q mit $m = |q|$ in einem Trie benötigt $O(m)$ Zeit, unabhängig von der Anzahl der Elemente im Trie. Dies ist asymptotisch sogar schneller als eine imperfekte Hash-Tabelle, die im schlechtesten Fall $O(n)$ benötigt.

Eine genaue Abschätzung der Anfragezeit $T_{\text{Trie}}(n, e)$ einer Ähnlichkeitssuche mit Toleranz e in Tries ist sehr komplex und hängt wie auch der Speicherbedarf von der konkreten Verteilung der Daten ab. Eine Betrachtung unter der Voraussetzung von verschiedenen Wahrscheinlichkeitsmodellen für die Buchstabenverteilung findet sich in [Cle01]. Eine experimentelle Untersuchung von Tries findet sich in Abschnitt 8.4.4.

Die Anfragezeit einer ungenauen Suche hängt exponentiell von der verwendeten Suchtoleranz ab [SM96],[6] so dass in einem Trie nicht mit einer hohen Toleranz gesucht werden sollte, wenn die Effizienz eine entscheidende Rolle spielt.

5.2.4. Erweiterung: Ternary Search Tries

Eine Variante von Tries sind die Ternary Search Tries (TST). Bei diesen werden die Kindknoten nicht in einem Array oder in einer Liste, sondern konzeptionell wie in einem Binärbaum verwaltet. Der Baum erhält dadurch folgende Gestalt: Jeder Knoten speichert einen Buchstaben u und nicht mehr $|\Sigma|$ sondern drei Zeiger *links*, *gleich* und *rechts* (siehe Abbildung 5.4). Der Zeiger *gleich* zeigt auf den Kindknoten, der dem gespeicherten Buchstaben c entspricht. Um die Kindknoten bezüglich eines anderen Buchstabens w zu erreichen, muss den Zeigern *links* beziehungsweise *rechts* gefolgt werden, je nachdem ob w lexikographisch vor oder nach u steht. Die Wahl, welcher Buchstabe in welchem Knoten gespeichert wird, sollte möglichst einen balancierten Baum sicher stellen. Dies ist möglich, wenn die Menge der Zeichenketten statisch vorgegeben ist [BS98, S. 4].

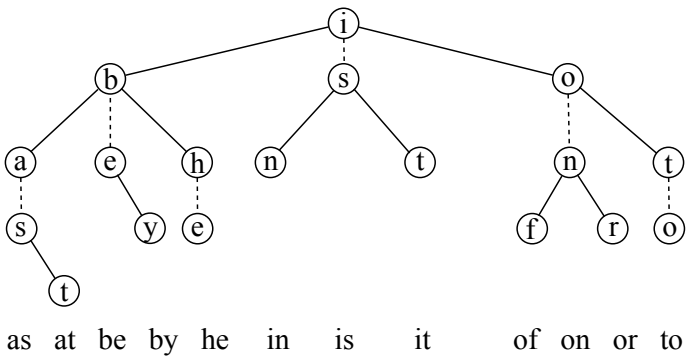

Abbildung 5.4.: Ternary Search Trie mit 12 zwei-buchstabigen Wörtern [BS97]

Die Algorithmen zum Aufbau und zur Suche entsprechen denen eines gewöhnlichen Tries, mit Modifikationen an den Stellen, wo ein Kind eines Knotens ausgewählt wird.

Ternary Search Tries bieten einen Kompromiss zwischen der speichersparenden Verwaltung der Kindknoten als Liste und der schnelleren Verwaltung als Array.

[6] Diese exponentielle Abhängigkeit wurde auch in den experimentellen Untersuchungen bestätigt, siehe Abschnitt 8.4.4.

5.2.5. Bewertung

Tries sind eine Datenstruktur zum Verwalten großer Mengen von Zeichenketten. Sie eignen sich für eine effiziente Beantwortung sowohl von exakten als auch von unscharfen Anfragen. Als Ähnlichkeitsfunktion kann dabei je nach Anwendung eine Variante der Editierdistanz verwendet werden. Der Speicherbedarf ist nur moderat, weil gemeinsame Präfixe der Zeichenketten nur einmal abgespeichert werden. Dadurch kann davon profitiert werden, dass es in der natürlichen Sprache (und auch beispielsweise bei Orts- und Straßennamen) viele gemeinsame Präfixe gibt. Die Anfragezeit hängt kaum von der Datenbankgröße, aber exponentiell von der Suchtoleranz ab. Suchen mit einer großen Toleranz sollten deshalb vermieden werden.

Für die Implementierung wurden in dieser Arbeit Ternary Search Tries verwendet. Diese Datenstruktur ist als Bibliothek für die Programmiersprache C++ verfügbar [Ekm07].

5.3. Indexstruktur für Datensätze: FRI

In diesem Abschnitt schlage ich den Fri (Fast Record Index)[7] als Index für Datensätze vor, der zu einem gegebenen Anfragedatensatz die ähnlichsten Datensätze in einer Datenbank ermittelt.

Es gibt verschiedene Möglichkeiten, einen Index zur Suche von ähnlichen Datensätzen zu konstruieren. Eine Möglichkeit sind die bei den verwandten Arbeiten in Abschnitt 3.2.1 angesprochenen metrischen Bäume. Diese bieten eine generische Unterstützung für verschiedene Abstandsmaße. Die einzige Anforderung ist dabei, dass das Abstandsmaß eine Metrik ist. Durch die relativ hohe Laufzeit der unscharfen Anfragen sind metrische Bäume jedoch beim heutigen Stand der Technik nicht für die Adresskorrektur einsetzbar. Die Anfragen benötigen für 338.000 Adressen bei einer Suchtoleranz von 5 bezogen auf die einfache Editierdistanz über eine Sekunde (siehe Abschnitt 3.2.1). Bei der Adresskorrektur sollen auch Datensätze mit größerem Abstände innerhalb von 100 ms gefunden werden. Wahrscheinlich kann die Suchlaufzeit von metrischen Bäume hier auch nicht durch einfache Optimierungen auf einen akzeptablen Wert gebracht werden.

Alternativ könnten die Datensätze auch auf Vektoren abgebildet und dann beispielsweise mit Hilfe von k-d-Bäumen, R-Bäumen oder Bereichsbäumen indiziert werden. Die Ähnlichkeit von Zeichenketten lässt sich jedoch kaum, beziehungsweise nur in einer sehr hohen Dimension auf Vektoren abbilden. Die genannten Baumstrukturen können Suchanfragen jedoch bei einer solche hohen Dimensionalität nicht effizient beantworten, so dass sie hier nicht verwendet werden können.

Eine weitere Möglichkeit zur Konstruktion eines Indexes zur Suche von ähnlichen Datensätzen ist es, die Datensätze als Zeichenketten zu kodieren und dann in einem Trie zu indizieren. Dieser einfache Ansatz funktioniert prinzipiell recht gut, hat jedoch auch einige Nachteile. Er wird im folgenden Abschnitt vorgestellt. In den darauf folgenden Abschnitten wird eine komplexere, aber effizientere Indexstruktur mit dem zugehörigen Algorithmus zur Suche von ähnlichen Datensätzen vorgeschlagen.

5.3.1. Einfacher Ansatz

Ein erster Ansatz zur Konstruktion eines Indexes für Datensätze basierend auf einem Index für Zeichenketten ist es, alle Datensätze als Zeichenketten zu kodieren und diese dann zu

[7] Der Name Fri ist ein Akronym von Fast Record Index (deutsch: Schneller Datensatz-Index) und an den Namen der *Trie*-Datenstruktur angelehnt, die von ihm verwendet wird. Außerdem ist *fri* das schwedische Wort für *frei* und referenziert damit den Namen der Freien Universität Berlin.

indizieren. Eine solche Kodierung kann beispielweise die Konkatenation der Attributwerte sein, separiert durch ein Trennzeichen. Die Kodierung eines Datensatzes r mit den Attributwerten $r.a_1$, $r.a_2$, ..., $r.a_m$ und dem Trennzeichen „#“ ist dann „$r.a_1\#r.a_2\#\ldots\#r.a_m$“.

Dies ist ein sinnvoller Ansatz, weil es mit diesem Verfahren relativ einfach möglich ist, ähnliche Datensätze zu einem gegebenen Anfragedatensatz zu finden. Dazu wird der Anfragedatensatz ebenfalls kodiert und mit dieser Kodierung im Index nach ähnlichen Kodierungen gesucht.[8] Das Ergebnis sind dann die jeweils zugehörigen Datensätze.

Bei diesem Verfahren können teilweise die Vorteile von Tries übernommen werden, unter anderem, dass gemeinsame Präfixe auch gemeinsam abgespeichert werden. Dabei spielt hier allerdings die Reihenfolge der Attribute eine große Rolle, weil im Allgemeinen nur die Präfixe des ersten Attributs gemeinsam gespeichert werden können.[9] Der Einfluss der Reihenfolge der Attribute wird in Abschnitt 8.4.4 untersucht.

Allerdings ist der Begriff der *Ähnlichkeit* bei diesem einfachen Ansatz sehr eingeschränkt. Auf alle Attribute wird dieselbe Ähnlichkeitsfunktion angewendet, so dass es nicht möglich ist, verschiedene Ähnlichkeitsmaße für verschiedene Attribute zu verwenden. Insbesondere muss dies eine Ähnlichkeitsfunktion für *Zeichenketten* sein und kann beispielsweise keine numerische Ähnlichkeitsfunktion sein. Außerdem gehen alle Attribute mit demselben Gewicht ein. Dadurch ist es nicht möglich, Abweichungen in einem für die Ähnlichkeit besonders wichtigen Attribut höher zu gewichten.

Des Weiteren können bei diesem Verfahren die Synonyme für Attributwerte nicht effizient integriert werden. Für jedes Synonym eines Attributwertes müsste ein weiterer Datensatz eingefügt werden. Tauchen Synonyme bei mehreren Attributen eines Datensatzes auf, müssen alle Kombinationen im Index gespeichert werden, was mit einem hohen Platzbedarf verbunden ist.

Ein weiterer Nachteil dieser Art der Indizierung ist die relativ hohe Laufzeit für eine Suchanfrage. Wie oben erläutert hängt die Laufzeit exponentiell von der Suchtoleranz ab. Bei diesem Verfahren ist es jedoch häufig nötig, mit einer hohen Suchtoleranz zu suchen, weil die Abweichungen in allen Attribute in *einer* Suche behandelt werden müssen. Das Ziel beim nachfolgend vorgeschlagenen Algorithmus ist es, anstatt *einer aufwändigen* Suche, *mehrere einfache* Suchen durchzuführen.

5.3.2. Idee

Um die genannten Nachteile der naiven Indizierung zu vermeiden, schlage ich hier eine neue Indexstruktur für Datensätze vor. Insbesondere soll bei diesem Verfahren die langsame Suche mit einer hohen Suchtoleranz vermieden werden, um eine kürzere Anfragezeit bei der Suche nach ähnlichen Datensätzen zu erreichen. Die Datenstruktur besteht nicht (wie beim einfachen Ansatz) aus einem großen Trie, sondern aus vielen kleineren Tries. Anstatt einer Suche mit einer großen Suchtoleranz gibt es dann mehrere Anfragen mit einer kleineren Toleranz.

Die Idee dabei ist, dass die Datensätze bei einer Ähnlichkeitssuche nacheinander Attribut für Attribut mit Werten belegt werden. Für die Bestimmung des ersten Attributs muss nur mit einer geringen Suchtoleranz gesucht werden. Die verbleibenden Attribute werden dann jeweils ausgehend von den bisher schon ausgewählten Attributwerten bestimmt. Dabei muss nur noch jeweils ein kleiner Teil der Datenbank durchsucht werden. In den nächsten

[8] Dabei sollte sichergestellt werden, dass Editieroperationen sich nicht über die Trennzeichen hinweg erstrecken.

[9] Nur wenn Datensätze im ersten Attribut übereinstimmen, werden auch die Präfixe des zweiten Attributs gemeinsam gespeichert.

beiden Abschnitten wird zunächst die verwendete Datenstruktur und anschließend der Suchalgorithmus dargestellt.

5.3.3. Datenstruktur

Die Datenstruktur des Indexes besteht grundsätzlich aus zwei Teilen:

1. Für jedes Attribut A_i gibt es einen Index I_i, der es ermöglicht zu einem gegebenen Anfragewert x alle ähnlichen Werte y des Attributs A_i in der Datenbank effizient zu finden.

2. Zu jedem Attributwert $x \in A_j$ gibt es für jedes Attribut A_i mit $i \neq j$ einen Index I_i^x. Ausgehend von einem Attributwert x können damit zu einem Attribut A_i effizient die Werte y gefunden werden, die gemeinsam in einem Datensatz vorkommen, d. h. $(\ldots, x, \ldots, y, \ldots) \in D$.

Jeder Wert x eines Attributes wird dabei durch ein Objekt W_{x} repräsentiert. Die Indizes sind Tries und jeweils eine Abbildung von Zeichenketten x auf die zugehörigen Repräsentanten W_{x}. Ein Beispiel veranschaulicht diese Architektur im Folgenden. Gegeben ist eine Datenbanktabelle mit den drei Attributen *ort*, *plz* und *str*:

ort	*plz*	*str*
Berlin	10963	Möckernstr.
Berlin	10963	Yorckstr.
Berlin	14169	Clayallee
München	80637	Yorckstr.

Die Indizes der Kategorie 1 zu dieser Datenbank sind:

$$I_{ort} = \{\text{„Berlin“} \to W_{\text{Berlin}}, \text{„München“} \to W_{\text{München}}\}$$
$$I_{plz} = \{\text{„10963“} \to W_{10963}, \text{„14169“} \to W_{14169}, \text{„80637“} \to W_{80637}\}$$
$$I_{str} = \{\text{„Clayallee“} \to W_{\text{Clayallee}}, \text{„Möckernstr.“} \to W_{\text{Möckernstr.}}, \text{„Yorckstr.“} \to W_{\text{Yorckstr.}}\}$$

Außerdem gibt es folgende Indizes der Kategorie 2:

$$I_{plz}^{\text{Berlin}} = \{\text{„10963“} \to W_{10963}, \text{„14169“} \to W_{14169}\}$$
$$I_{str}^{\text{Berlin}} = \{\text{„Clayallee“} \to W_{\text{Clayallee}}, \text{„Möckernstr.“} \to W_{\text{Möckernstr.}}, \text{„Yorckstr.“} \to W_{\text{Yorckstr.}}\}$$
$$I_{plz}^{\text{München}} = \{\text{„80637“} \to W_{80637}\}$$
$$I_{str}^{\text{München}} = \{\text{„Yorckstr.“} \to W_{\text{Yorckstr.}}\}$$
$$I_{ort}^{10963} = \{\text{„Berlin“} \to W_{\text{Berlin}}\}$$
$$I_{str}^{10963} = \{\text{„Möckernstr.“} \to W_{\text{Möckernstr.}}, \text{„Yorckstr.“} \to W_{\text{Yorckstr.}}\}$$
$$I_{ort}^{14169} = \{\text{„Berlin“} \to W_{\text{Berlin}}\}$$
$$I_{str}^{14169} = \{\text{„Clayallee“} \to W_{\text{Clayallee}}\}$$
$$I_{ort}^{80637} = \{\text{„München“} \to W_{\text{München}}\}$$
$$I_{str}^{80637} = \{\text{„Yorckstr.“} \to W_{\text{Yorckstr.}}$$

$$
\begin{aligned}
I_{plz}^{\text{Clayallee}} &= \{\text{„14169“} \rightarrow W_{14169}\} \\
I_{ort}^{\text{Clayallee}} &= \{\text{„Berlin“} \rightarrow W_{\text{Berlin}}\} \\
I_{plz}^{\text{Yorckstr.}} &= \{\text{„10963“} \rightarrow W_{10963}, \text{„80637“} \rightarrow W_{80637}\} \\
I_{ort}^{\text{Yorckstr.}} &= \{\text{„Berlin“} \rightarrow W_{\text{Berlin}}, \text{„München“} \rightarrow W_{\text{München}}\} \\
I_{plz}^{\text{Möckernstr.}} &= \{\text{„10963“} \rightarrow W_{10963}\} \\
I_{ort}^{\text{Möckernstr.}} &= \{\text{„Berlin“} \rightarrow W_{\text{Berlin}}\}
\end{aligned}
$$

Ein Ausschnitt dieser Datenstruktur ist in Abbildung 5.5 dargestellt. Die Kästen stehen jeweils für einen Index, die Bezeichnung des Indexes steht links daneben. (Von einem Repräsentanten W_{x} gibt es jeweils Verweise auf die zugehörigen Indizes I_i^x.)

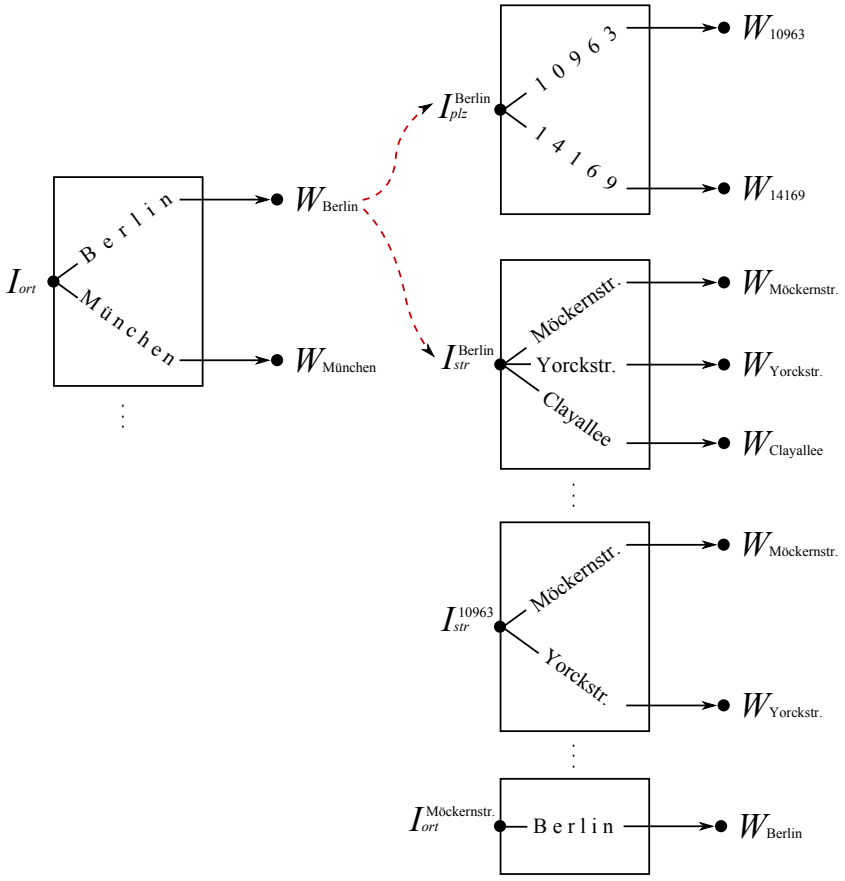

Abbildung 5.5.: Datenstruktur beim FRI (Ausschnitt des Beispiels)

Im Allgemeinen gibt es m Indizes der ersten Kategorie und $|A_1|*(m-1)+...+|A_m|*(m-1)$ Indizes der zweiten Kategorie. Dabei bezeichnet $|A_i|$ die Anzahl der unterschiedlichen Attributwerte von A_i in der Datenbank.

Die Indizes der Kategorie 1 ermöglichen es, zu einem gegebenen Anfragedatensatz eine Kandidatenmenge zu bestimmen, indem eine Suche bezüglich eines beliebigen Attributs durchgeführt wird. Als Attribut kann beispielsweise ein solches gewählt werden, das mit einer hohen Wahrscheinlichkeit unverändert in der Datenbank enthalten ist. Gibt es bei der Suche nach diesem Attribut ein Ergebnis, kann zunächst diese sehr kleine Teilmenge der Datenbank mit Hilfe der Indizes der Kategorie 2 weiter durchsucht werden. Der genaue Algorithmus wird in Abschnitt 5.3.5 erläutert.

Synonyme

In diese Datenstruktur können auch Synonyme für Attributwerte integriert werden. Dazu muss für jedes Synonym y eines Wertes x in jedem betroffenen Index nur ein weiteres Element $y \rightarrow W_{\text{x}}$ hinzugefügt werden. Ein Beispiel erläutert dies im Folgenden kurz. Dabei wird das oben genannte Beispiel um folgende Synonyme erweitert:

$$\begin{aligned}
&\text{Synonyme}_{ort} = \{\text{„Dahlem“} \rightarrow \text{„Berlin“}\} \\
&\text{Synonyme}_{plz} = \{\} \\
&\text{Synonyme}_{str} = \{\text{„Möckernstraße“} \rightarrow \text{„Möckernstr.“}, \text{„Yorckstraße“} \rightarrow \text{„Yorckstr.“}\}
\end{aligned}$$

Die Indizes der ersten Kategorie sind unter Berücksichtigung der Synonyme dann folgendermaßen aufgebaut (analog für die Indizes der zweiten Kategorie):

$$\begin{aligned}
I_{ort} &= \{\text{„Berlin“} \rightarrow W_{\text{Berlin}}, \text{„Dahlem“} \rightarrow W_{\text{Berlin,}}\ \text{„München“} \rightarrow W_{\text{München}}\} \\
I_{plz} &= \{\text{„10963“} \rightarrow W_{10963}, \text{„14169“} \rightarrow W_{14169}, \text{„80637“} \rightarrow W_{80637}\} \\
I_{str} &= \{\text{„Clayallee“} \rightarrow W_{\text{Clayallee}}, \\
&\qquad \text{„Möckernstr.“} \rightarrow W_{\text{Möckernstr.}}, \text{„Möckernstraße“} \rightarrow W_{\text{Möckernstr.}}, \\
&\qquad \text{„Yorckstr.“} \rightarrow W_{\text{Yorckstr.}}, \text{„Yorckstraße“} \rightarrow W_{\text{Yorckstr.}}\}
\end{aligned}$$

Bemerkung Falls es Werte unter den Synonymen gibt, die auf mehrere verschiedene Werte abgebildet werden (formal: $\exists y, x_1, x_2 : (y \rightarrow x_1), (y \rightarrow x_2) \in \text{Synonyme}$ mit $x_1 \neq x_2$), müssen die Tries jeweils Multimengen verwalten.

5.3.4. Aufbau

Beim Aufbau der Indexstruktur werden alle Datensätze der Datenbank sequentiell durchlaufen (siehe Algorithmus 5.2). Jeder Attributwert $r.a_i$ eines Datensatzes wird in die die jeweilige Indexstruktur eingefügt. Außerdem wird er in die Indexstrukturen der zweiten Kategorie für jeden anderen Attributwert $r.a_j$ des Datensatzes eingefügt. Die Operation $X \Leftarrow x$ bedeutet im Folgenden, dass der Wert x in die Datenstruktur X eingefügt wird.

Algorithmus 5.2: Aufbau der FRI-Datenstruktur

```
1 FriAufbau(Datenbank D, Synonyme[1], ..., Synonyme[m]) begin
2     for r in D do
3         for i = 0 to m do
4             // Einfügen in den Index der Kategorie 1
5             I_i ⇐ r.a_i
```

```
6                   füge s als Verweis auf r.a_i ein, für (s → r.a_i) ∈ Synonyme[i]
7
8                   // Einfügen in den Index der Kategorie 2
9                   for j = 0 to m, j ≠ i do
10                      x := r.a_j
11                      I_i^x  ⇐  r.a_i
12                      füge s als Verweis auf r.a_i ein, für (s → r.a_i) ∈ Synonyme[i]
13                  end for
14              end for
15          end for
16  end
```

5.3.5. Algorithmus

Dieser Abschnitt erläutert den Algorithmus zur Ähnlichkeitssuche in der vorgeschlagenen Indexstruktur. Dabei sollen bei Eingabe eines Anfragedatensatzes diejenigen Datensätze gefunden werden, die den geringsten Abstand zum Anfragedatensatz haben. Der Abstand zweier Datensätze q und r ist hier definiert als die gewichtete Summe der attributweisen Distanzen:

$$\delta(q, r) = g_1 \cdot \delta_1(q.a_i, r.a_i) + \ldots + g_m \cdot \delta_{\mathrm{m}}(q.a_m, r.a_m)$$

Im Folgenden wird der Abstand der Einfachheit halber als ungewichtete Summe der attributweisen einfachen Editierdistanzen dargestellt (also $\forall_i : g_i = 1, \delta_{\mathrm{i}} = \delta_{\mathrm{edit}}$). Implementiert wurde das Verfahren hingegen mit unterschiedlichen Gewichten für die einzelnen Attribute, und auch beispielsweise die gewichtete Editierdistanz mit erweiterten Operationen als Ähnlichkeitsmaß ist denkbar.

Um die Suche effizient zu gestalten, sollen Suchen in den Indizes mit hoher Toleranz vermieden werden. Insbesondere soll in den Indizes der Kategorie 1 nicht mit hoher Toleranz gesucht werden, weil diese jeweils alle Werte bezüglich eines Attributes der Datenbank enthalten.

Bei der Suche werden die Attribute eines Ergebnisdatensatzes schrittweise nacheinander mit Werten belegt. Dadurch kann ausgenutzt werden, dass in der Regel nicht alle Attributwerte des ähnlichsten Datensatzes einen hohen Abstand zum Anfragedatensatz haben. Der Algorithmus belegt erst eins der wenig veränderten Attribute mit einem Wert und bestimmt davon ausgehend passende Werte für die anderen Attribute. Dadurch muss in der gesamten Datenbank nur mit einer geringen Toleranz gesucht werden und die Suchen mit hoher Toleranz finden nur in kleineren Teilmengen statt.

In den folgenden Abschnitten wird zunächst das Problem genauer erläutert und der Algorithmus als Entscheidungsbaum modelliert. Anschließend wird die Suche in den Indizes und die Auswahl des besten Weges im Entscheidungsbaum erläutert. Zur Beschreibung des Algorithmus' ist zunächst folgende Definition hilfreich.

Definition 5.1 (Partieller Datensatz)**.** *Ein Datensatz r heißt* partiell definiert*, wenn beliebig viele seiner Attribute den Wert* undefiniert *(?) haben. Er hat im Allgemeinen folgende Gestalt:* $r = (?, \ldots, a_{i_1}, ?, \ldots, a_{i_2}, ?, \ldots, a_{i_k}, ?, \ldots)$.

Ein Datensatz, bei dem kein Attribut mit einem Wert belegt ist, heißt undefiniert *und wird mit* $(?, \ldots, ?)$ *notiert.*

Ein Datensatz, bei dem alle Attribute mit einem Wert belegt sind, heißt vollständig definiert.

Die Anzahl k der definierten Attribute wird als Rang *von r bezeichnet.*

Problem

Der Algorithmus beginnt mit dem undefinierten Datensatz $(?, \ldots, ?)$ und belegt die Attribute schrittweise mit einem Wert. Wenn alle Attribute gefüllt sind, ist der ähnlichste Datensatz zum gegebenen Anfragedatensatz gefunden. Die Reihenfolge, in der die Attribute mit Werten belegt werden, spielt dabei hinsichtlich der Effizienz eine entscheidende Rolle. Der Algorithmus zur Ähnlichkeitssuche sollte dabei möglichst eine Reihenfolge finden, bei der nur wenig Suchen in den Indizes mit einer hohen Suchtoleranz nötig sind. Für den Algorithmus zur Suche von ähnlichen Datensätzen im FRI gelten folgende Rahmenbedingungen:

Ziel: Bestimmung der Kombination von Attributwerten, die die geringste Distanz zum Anfragedatensatz hat.

Optimierungskriterium: Vermeidung von Suchen in den Indizes mit einer hohen Suchtoleranz.

Invariante: Ein Zwischenergebnis r_i im Schritt i hat mindestens den gleichen erwarteten Abstand wie das Zwischenergebnis r_{i-1} im Schritt $i-1$.[10]

Abbruchkriterium: Beispielsweise „Alle Ergebnisse mit Toleranz e wurden gefunden“ oder „Es wurden k Ergebnisse gefunden“.

Im Laufe des Algorithmus' müssen an verschiedenen Stellen Entscheidungen getroffen werden. Zunächst muss bestimmt werden, mit welchem Attribut A_i begonnen wird. Dann muss die Suchtoleranz für diese Suche auf diesem Attribut bestimmt werden. Wenn die Suche mit dieser Toleranz kein Ergebnis liefert, muss eine höhere Suchtoleranz oder aber ein anderes Attribut ausgewählt werden. Wenn die Suche ein nicht-leeres Ergebnis hat, muss ein Attributwert x aus dieser Menge ausgewählt werden, um ausgehend davon die Suche fortzusetzen. Anschließend muss das nächste Attribut A_j (mit $j \neq i$) gewählt werden und so weiter. Insgesamt müssen im Lauf des Algorithmus' mehrere Entscheidungen folgenden Typs getroffen werden:

1. Welches Attribut A_i soll als nächstes mit einem Wert belegt werden?
2. Mit welcher Suchtoleranz e soll bezüglich dieses Attributs gesucht werden?
3. Welcher Ergebniswert x einer Suche soll verwendet werden?

Entscheidungsbaum

Diese Wahlmöglichkeiten werden hier mit einer Art Entscheidungsbaum modelliert (siehe Abbildung 5.6). Jede getroffene Entscheidung wird durch einen Knoten repräsentiert. Damit gibt es neben dem *Wurzelknoten* folgende drei Typen von Knoten: *Attributknoten*, *Suchknoten* und *Elementknoten*.

- Der *Wurzelknoten* stellt selbst keine Entscheidung dar, sondern dient lediglich als Startpunkt des Algorithmus'. Zu Beginn muss entschieden werden, welches Attribut des undefinierten Datensatzes $(?, \ldots, ?)$ als erstes bestimmt werden soll. Der Wurzelknoten hat also für jedes der m Attribute einen Attributknoten als Kind.

[10] Bemerkung: Der Abstand ist außerdem höchstens um 1 höher, wenn die einfache Editierdistanz verwendet wird und die Attribute alle ein Gewicht von 1 haben.

- Die *Attributknoten* repräsentieren die Entscheidung, welches Attribut eines Datensatzes bestimmt werden soll. Sie sind 2-Tupel (r, i) bestehend aus einem partiell definierten Datensatz r sowie einem Index i des zu bestimmenden Attributs.[11] Die Kinder von Attributknoten sind Suchknoten.

- *Suchknoten* repräsentieren die Entscheidung, mit welcher Suchtoleranz gesucht werden soll. Sie sind 3-Tupel (r, i, e), bestehend aus einem partiell definierten Datensatz r, dem Index i eines Attributs sowie einer Suchtoleranz e. In einem Suchknoten werden die Werte der Datenbank für das Attribut A_i bestimmt, die einen Abstand von e zum Wert des Anfragedatensatzes haben. Hier findet also die Suche in den Indizes statt, die in Abschnitt 5.3.5 noch detaillierter erläutert wird. Die Kinder von Suchknoten sind Elementknoten, jeweils ein Knoten für jedes Suchergebnis.

- *Elementknoten* repräsentieren die Entscheidung, welcher konkrete Wert für ein Attribut aus einer möglicherweise mehrelementigen Ergebnismenge einer Suche ausgewählt wird. Elementknoten sind 3-Tupel (r, i, x) und bestehen aus einem partiell definierten Datensatz, dem Index i eines Attributs sowie einem Wert x für dieses Attribut. In einem Elementknoten wird ein neuer partieller Datensatz r' gespeichert, der r entspricht, aber bei dem zusätzlich das Attribut $r'.a_i$ mit x belegt ist. Die Kinder von Elementknoten sind Attributknoten und zwar jeweils einer für jedes noch nicht definierte Attribut. Wenn ein Elementknoten ein Blattknoten ist (das heißt alle Attribut mit einem Wert belegt sind), ist ein vollständig definierter Datensatz gefunden.

Die Entscheidungen werden bewertet und der Algorithmus setzt die Ausführung immer an der jeweils besten Stelle im Entscheidungsbaum fort. Die Bewertung der Entscheidungen, also die Reihenfolge, in der die Knoten bearbeitet werden, ergibt sich aus dem insgesamt mindestens erwarteten Abstand des partiellen Datensatzes r zum Anfragedatensatz q. Eine genauere Erläuterung, wie diese Priorität berechnet wird, folgt im nächsten Abschnitt.

Die Knoten des Entscheidungsbaumes werden in einer Prioritätswarteschlange gehalten und der Reihe nach abgearbeitet. Dabei wird immer der Knoten betrachtet, dessen Datensatz den geringsten insgesamt zu erwartenden Abstand zum Anfragetupel hat. In der Warteschlange ist zu Beginn nur der Wurzelknoten, weitere Knoten werden während der Ausführung der Knoten hinzugefügt. Der Suchalgorithmus für die Ähnlichkeitssuche in einem FRI ist in Algorithmus 5.3 dargestellt:

Algorithmus 5.3: Ähnlichkeitssuche im FRI

```
1  FriSuche(Datensatz q, Abbruchkriterium a) begin
2      ergebnisse := {}
3      schlange := {}
4      schlange ⇐ Wurzelknoten
5      do
6          v := schlange.pop
7          Auswerten(v)
8          r := partiellerDatensatz(v)
9          if r ist vollständig definiert then
10             ergebnisse ⇐ r
11         if Abbruchkriterium a trifft zu then
12             break
13     loop
```

[11] *Index* ist hier im Sinne einer natürlichen Zahl und nicht einer Datenstruktur gemeint.

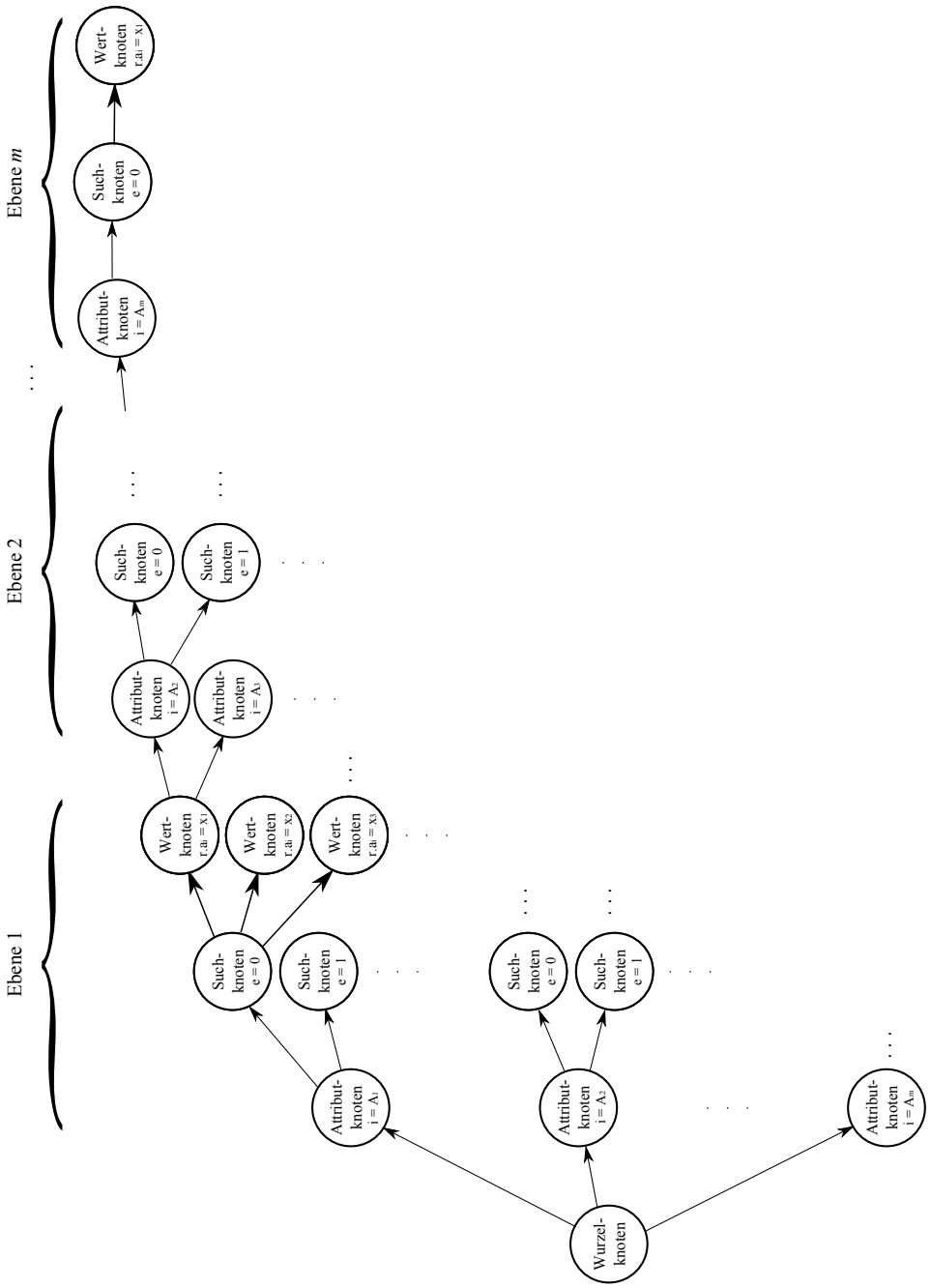

Abbildung 5.6.: Suchalgorithmus im FRI als Entscheidungsbaum

```
14      return ergebnisse
15  end
```

Dies ist der zu Grunde liegende Algorithmus zur Suche der ähnlichsten Datensätze. Dabei ist jedoch noch offen, was bei der Auswertung der Knoten geschieht und nach welchem Kriterium die Prioritätswarteschlange sortiert ist. Zunächst werden die Algorithmen zur Auswertung der verschiedenen Knotentypen angegeben. Die Algorithmen entsprechend jeweils genau der Beschreibung der Knoten zu Beginn dieses Abschnitts.

Algorithmus 5.4: Auswertung der Knoten des Entscheidungsbaumes

```
1   Auswerten(Wurzelknoten) begin
2       r := (?, ..., ?)
3       for j = 0 to m do
4           schlange ⇐ Attributknoten(r, j)
5       end for
6   end
7
8   Auswerten(Attributknoten(r, i)) begin
9       schlange ⇐ Suchknoten(r, i, 0)      // [12]
10  end
11
12  Auswerten(Suchknoten(r, i, e)) begin
13      menge := FindeAttributWerte(q, r, i, e)
14      for x in menge do
15          schlange ⇐ Elementknoten(r, i, x)
16      end for
17      schlange ⇐ Suchknoten(r, i, e + 1)
18  end
19
20  Auswerten(Elementknoten(r, i, x)) begin
21      r.a_i := x
22      for j = 0 to m do
23          if r.a_j ist noch undefiniert then
24              schlange ⇐ Attributknoten(r, j)
25          end if
26      end for
27  end
```

Im Folgenden ist noch die Funktion FindeAttributWerte() zur eigentlichen Suche in den Indizes zu definieren.

Suche in den Indizes

In einem Suchknoten (r, i, e) sollen für den partiellen Datensatz r diejenigen Werte des Attributs $r.a_i$ gefunden werden, die zum Wert $q.a_i$ des Anfragedatensatzes einen Abstand von e haben.

Wenn der Datensatz r noch vollständig undefiniert ist, muss im Index I_i gesucht werden, der alle Werte der Datenbank bezüglich dieses Attributs enthält. Wenn schon Attribute des Datensatzes r definiert sind, muss nicht in der gesamten Datenbank gesucht werden, sondern nur in den Indizes I_i^y bezüglich der schon belegten Attributwerte y.

[12] Die Suchtoleranz muss nicht bei 0 begonnen werden, wenn aus Berechnungen in anderen Teilen des Baumes schon bekannt ist, dass es kein Ergebnis mit Toleranz 0 gibt. Die wird im Beispiel am Ende dieses Abschnitts erläutert.

Algorithmus 5.5: Suche in den Indizes

```
1  FindeAttributWerte(q, r, i, e) begin
2        if r = (?, ..., ?) then
3              return TrieSuche(I_i, q.a_i, e)
4        else
5              ergebnis := A_i
6              for j = 0 to m do
7                    y := r.a_j
8                    ergebnis = ergebnis ∩ TrieSuche(I_i^y, q.a_i, e)
9              end for
10             return ergebnis
11       end if
12 end
```

In diesem zweiten Fall muss außerdem sicher gestellt werden, dass eine Belegung des Attributs $r.a_i$ zu einem Datensatz führt, der eine Fortsetzung in der Datenbank hat. (Ein partieller Datensatz hat eine *Fortsetzung* in der Datenbank, wenn es eine Belegung der undefinierten Attribute gibt, so dass der entstehende Datensatz in der Datenbank ist.)

Um dies sicher zu stellen, wird in den Indizes für jedes schon definierte Attribut die Ergebnismenge ermittelt und die Schnittmenge gebildet. Jeder Wert in dieser Schnittmenge kommt also mit jedem anderen schon definierten Attributwert gemeinsam in einem Datensatz der Datenbank vor.

In der Implementierung wird die Schnittmenge dadurch gebildet, dass die Ähnlichkeitssuche für eins der schon belegten Attribute durchgeführt wird und anschließend für jeden so erhaltenen Wert überprüft wird, ob er auch in den Indizes der anderen Attribute enthalten ist. Dadurch muss insgesamt nur *eine* Ähnlichkeitssuche durchgeführt werden. Bei der Bildung dieser Schnittmenge kann der Aufwand durch eine geschickte Wahl der Reihenfolge der Mengen noch verringert werden. In der bisherigen Implementierung wird die Schnittmenge jedoch einfach in der Reihenfolge der Attribute gebildet.

Bemerkung Somit ist jedoch im Allgemeinen noch nicht sicher gestellt, dass auch *alle* Werte gemeinsam in einem Datensatz vorkommen. Dies ist dann gewährleistet, wenn auch ein Schlüsselattribut der Relation selektiert wird, weil in diesem Fall jedes Attribut gemeinsam mit diesem Schlüssel auftreten muss. Beim konkreten Fall der Suche von ähnlichen Adressen mit den Attributen *ort*, *plz* und *str* ist aber keines der Attribute ein Schlüssel. Jedoch bestimmt immer entweder die Postleitzahl den Ort oder der Ort die Postleitzahl. Damit ist hier Korrektheit sicher gestellt, auch wenn kein Schlüsselattribut selektiert wird.

Auswahl des besten Knotens

Der Algorithmus ist nun beinahe vollständig angegeben, es fehlt noch die Bestimmung der Priorität der Knoten in der Warteschlange. Es soll immer der Knoten als erstes bearbeitet werden, für dessen partiellen Datensatz die geringste Distanz zum Anfragedatensatz zu erwarten ist. Die minimal zu erwartende Distanz δ_{min} ist die Summe aus der tatsächlichen bisherigen Distanz δ_{bisher} und der zu erwartenden zukünftigen Distanz $\delta_{\text{künftig}}$:

$$\delta_{\text{min}}(v) = \delta_{\text{bisher}}(v) + \delta_{\text{künftig}}(v)$$

Die bisherige Distanz δ_{bisher} ist die Summe der Distanzen der schon definierten Attribute von r mit den Werten von q.

$$\delta_{\text{bisher}}(v) = \sum_{r.a_i \text{ ist definiert}} \delta(q.a_i, r.a_i)$$

Die zukünftig mindestens erwartete Distanz $\delta_{\text{künftig}}$ wird anhand der noch nicht definierten Attribute berechnet:

$$\delta_{\text{künftig}}(v) = d_{\text{me1}}(v, \{i \mid r.a_i \text{ ist undefiniert}\})$$

Die Berechnung der mindestens noch zu erwartenden Distanz d_{me1} für eine Menge von noch undefinierten Attributen wird im folgenden Abschnitt genauer erläutert.

Zunächst wird kurz diskutiert, wie verfahren werden kann, falls mehrere Knoten den gleichen erwarteten Abstand haben. In diesem Fall kann aus diesen Knoten prinzipiell ein beliebiger ausgewählt werden ohne das Ergebnis zu verändern. Um die Effizienz des Algorithmus' zu erhöhen, bietet sich jedoch an, gezielt einen Knoten zu wählen. Es kann beispielsweise derjenige Knoten als nächster bearbeitet werden, der mit den geringsten lokalen Berechnungskosten verbunden ist oder bei dem schon möglichst viele Attribute definiert sind. Außerdem kann es sinnvoll sein, Knoten, die viele Geschwister haben mit eher geringer Priorität auszuführen und wenn möglich an einer anderen Stelle weiter zu suchen. Ich habe verschiedene Heuristiken implementiert und in Abschnitt 8.4.2 untersucht.

Berechnung der erwarteten Distanz

In diesem Abschnitt wird erläutert, wie die mindestens zu erwartende Distanz d_{me1} einer Menge von Attributen in einem Knoten des Entscheidungsbaumes berechnet werden kann. Sie ergibt sich jeweils aus Berechnungen in anderen Teilen des Baumes.

Die mindestens erwartete Distanz eines Knotens sollte möglichst gut von unten abgeschätzt werden, so dass im Algorithmus keine Knoten ausgeführt werden, bei denen noch eine hohe Distanz erwartet wird. Somit wird sichergestellt, dass keine Pfade im Baum verfolgt werden, die wahrscheinlich nicht zum Finden des ähnlichsten Datensatzes beitragen. Das Konzept der mindestens noch zu erwartenden Distanz wird zunächst mit drei Beispielen motiviert.

- Wenn auf der ersten Ebene[13] im Entscheidungsbaum (also in der gesamten Datenbank) für ein Attribut A_i mit einer Suchtoleranz von e kein Wert gefunden wurde, kann es auch an keiner anderen Stelle im Baum ein Wert für dieses Attribut mit einem geringeren Abstand als e geben. Es gilt also: $d_{\text{me1}}(\text{Wurzelknoten } v, \{i\}) = e$.
- Wenn zu einem Elementknoten v (und damit zu einem partiell definierten Datensatz r) für ein Attribut A_i mit einer Suchtoleranz von e kein Wert gefunden wurde, kann es auch in keinem Unterbaum (und damit einer Fortsetzung von r) von v ein Wert für dieses Attribut mit einem geringeren Abstand als e geben. Es gilt also: $d_{\text{me1}}(\text{Wertknoten } v, \{i\}) = e$.
- Wenn unterhalb eines Elementknotens v für zwei Attribute A_i und A_j alle möglichen Kombinationen von Suchen mit den Toleranzen e_i, e_j mit $e_i + e_j \leq e$ ergebnislos waren,

[13] Siehe Abbildung 5.6

so haben die beiden Attribute in jedem Unterbaum von v zusammen einen Abstand von mindestens e: $d_{\text{me1}}(\text{Wertknoten } v, \{i, j\}) = e$.

Allgemeiner formuliert basiert die Berechnung der mindestens erwartete Distanz d_{me1} einer Menge von Attributen unter anderem auf folgenden Beobachtungen: Wenn alle Kinder eines Knotens v bezüglich einer Menge von Attributen eine mindestens erwartete Distanz von e haben, so ist die Distanz in v mindestens ebenso groß:

$$d_{\text{me1}}(v, M) \geq \min_{k \in \text{Kinder}(v)} d_{\text{me1}}(k, M)$$

Wenn der Vater eines Knotens v bezüglich einer Menge von Attributen eine mindestens erwartete Distanz von e hat, so ist die Distanz in v mindestens ebenso groß.

$$d_{\text{me1}}(v, M) \geq d_{\text{me1}}(\text{Vater}(v), M)$$

Dies sind die prinzipiellen Ideen der Berechnung der mindestens erwarteten Distanz. Davon ausgehend gelten für die vier verschiedenen Typen von Knoten jeweils leicht abgewandelte Berechnungen. Im Folgenden werden die entsprechenden Formeln der Berechnung einer Hilfsfunktion d_{me2} für jeden Knotentyp aufgeführt. Die eigentliche Funktion d_{me1} wird nachfolgend basierend auf der Funktion d_{me2} angegeben.

- Beim *Wurzelknoten* v kann auf die Berechnung des Vaterknotens verzichtet werden. Außerdem müssen nur die Kinder (welche vom Typ Attributknoten sind) betrachtet werden, deren gewähltes i in M enthalten ist. Die anderen Attribute können nicht zur Abschätzung beitragen.

$$d_{\text{me2}}(\text{Wurzelknoten } v, M) = \min_{k \in \text{Kinder}(v) \wedge attr(k) \in M} d_{\text{me1}}(k, M)$$

- Bei *Attributknoten* $v = (r, i)$ gelten die oben angegebenen Formeln ohne Modifikationen.

$$d_{\text{me2}}(\text{Attributknoten } v, M) = \max \left\{ \begin{array}{l} d_{\text{me1}}(\text{Vater}(v), M), \\ \min_{k \in \text{Kinder}(v)} d_{\text{me1}}(k, M) \end{array} \right\}$$

- Bei *Suchknoten* $v = (r, i, e)$ gibt es zwei kleine Modifikationen gegenüber der oben angegebenen Formel. Die erste betrifft das Betrachten der Kinder: Im darüber liegenden Suchknoten wurde ein Attribut i ausgewählt. Im Kindknoten des Suchknotens (welcher ein Elementknoten ist) wird dieses ausgewählte Attribut mit einem konkreten Wert belegt. Bei der rekursiven Berechnung der mindestens zu erwartenden Distanz in diesem Kindknoten wird daher also das schon definierte Attribut i aus der Menge heraus genommen und stattdessen die Distanz bezüglich dieses Attributs addiert.

$$d_{\text{me2}}(\text{Suchknoten } v, M) = \max \left\{ \begin{array}{l} d_{\text{me1}}(\text{Vater}(v), M), \\ \min_{k \in \text{Kinder}(v)} d_{\text{me1}}(k, M \setminus \{i\}) + e, \\ \omega \end{array} \right\}$$

Die zweite Modifikation ist die Einführung des Wertes ω. In einem Suchknoten ist die mindestens erwartete Distanz von jeder Menge von Attributen immer größer oder gleich der Suchtoleranz e, weil die Werte in allen Kindknoten mindestens diesen Abstand haben. Falls eine Suche in diesem Knoten ohne Ergebnis (also mit einer leeren Ergebnismenge) durchgeführt wurde, so ist die mindestens erwartete Distanz ∞, weil dieser Knoten eine „Sackgasse“ ist.

$$\text{mit } \omega = \begin{cases} \infty & \text{falls die Suche in diesem Knoten kein Ergebnis geliefert hat} \\ e & \text{sonst} \end{cases}$$

- Bei *Elementknoten* $v = (r, i, x)$ gibt es ebenfalls kleine Modifikationen gegenüber der allgemeinen Formel: Um möglichst viele Informationen in anderen Teilen des Entscheidungsbaumes ausnutzen zu können, wird die mindestens erwarteten Distanz mit Hilfe des Vaterknotens auf zwei Arten abgeschätzt. Einerseits mit der Menge M selbst und andererseits auch mit der Menge M erweitert um das aktuell ausgewählte Attribut i. Dabei muss dann jedoch die für dieses Attribut verwendete Suchtoleranz wieder subtrahiert werden (umgekehrt zur Vorgehensweise beim Übergang von Suchknoten zum Elementknoten).
  Außerdem müssen wie schon beim Wurzelknoten nur die Kinder (welche vom Typ Attributknoten sind) betrachtet werden, deren gewähltes Attribut j in M enthalten ist. Die anderen Attribute können nicht zur Abschätzung beitragen.

$$d_{\text{me2}}(\text{Elementknoten } v, M) = \max \left\{ \begin{array}{l} d_{\text{me1}}\left(\text{Vater}\left(v\right), M\right), \\ d_{\text{me1}}\left(\text{Vater}\left(v\right), M \cup \{i\}\right) - e, \\ \min_{k=(r,j) \in \text{Kinder}(v) \,\wedge\, j \in M} d_{\text{me1}}\left(k, M\right) \end{array} \right\}$$

In diesem Absatz wird die Berechnung der eigentlichen Funktion d_{me1} zur Berechnung der mindestens erwarteten Distanz basierend auf der Hilfsfunktion d_{me2} angegeben. Um die Distanz einer Menge M von Attributen möglichst gut abzuschätzen, werden alle Zerlegungen Z dieser Menge betrachtet. Für jede Zerlegung wird die mindestens zu erwartende Distanz als Summe der Distanzen der Teilmengen berechnet. Von all diesen Zerlegungen wird das Maximum verwendet. Durch dieses Betrachten aller Zerlegungen wird sicher gestellt, dass die Informationen aus den anderen Teilbäumen ausgenutzt werden, die sich jeweils auf verschiedene Mengen von Attributen beziehen können.

$$d_{\text{me1}}(v, M) = \max_{Z \text{ ist Partition von } M} \sum_{M' \in Z} d_{\text{me2}}\left(v, M'\right)$$

Durch die Betrachtung der Zerlegungen ist es beispielsweise bei Berechnung der mindestens erwarteten Distanz einer zwei-elementigen Menge $M = \{i, j\}$ möglich, eine Abschätzung für das Attribut i mit Hilfe des Vaterknotens und für das Attribut j aus einem Unterbaum zu erhalten.

Die Berechnung der mindestens erwarteten Distanz wurde in diesem Abschnitt funktional angegeben. In der Implementierung wird nach jeder Iteration (Algorithmus 5.3) in den Knoten jeweils die für den jeweiligen Unterbaum bekannte minimale Distanz gespeichert.

Beispiel

Dieser Abschnitt veranschaulicht den Algorithmus zur Ähnlichkeitssuche mit Hilfe eines Beispiels. Das Beispiel ist dabei relativ klein gewählt und deckt beispielsweise nicht alle

angeführten Fälle zur Berechnung der mindestens erwarteten Distanz ab. Es soll jedoch die zu Grunde liegenden Ideen verdeutlichen. Gegeben ist die in Abschnitt 5.3.3 angegebene Datenbank, allerdings reduziert auf die Attribute *ort* und *plz*:

ort	*plz*
Berlin	10963
Berlin	14169
München	80637

Außerdem ist der Anfragedatensatz $q = (\text{„Bedin“}, \text{„10963“})$ gegeben. Gesucht ist der ähnlichste Datensatz in der Datenbank.

Im Folgenden wird für jeden Schritt von Algorithmus 5.3 angegeben, was bei der Auswertung der Knoten stattfindet. Der Entscheidungsbaum ist in den Abbildungen 5.7, 5.8 und 5.9 visualisiert. Dabei ist rechts oberhalb von jedem Knoten die Ordnungsnummer in der Reihenfolge der Auswertung vermerkt. In der Prioritätswarteschlange befinden sich jeweils die erzeugten, aber noch nicht ausgewerteten Knoten; diese sind im Bild hell dargestellt. Wenn mehrere Knoten die gleiche mindestens erwartete Distanz haben, wird hier der Knoten mit der geringeren Suchtoleranz bevorzugt. Das Symbol $\emptyset$ bedeutet, dass bei einer Suche kein Ergebnis gefunden wurde. Zu den folgenden Schritten des Algorithmus' wird jeweils stichpunktartig die Auswertung des Knotens beschrieben. Außerdem wird jeweils für ausgewählte Knoten angegeben, wenn sich die mindestens erwartete Distanz für eine Attributmenge verändert.

1. Erzeugen neuer Kind-Attributknoten der Wurzel für *ort* und *plz*.
2. Erzeugen des neuen Kind-Suchknotens für das Attribut *ort* und die Toleranz $e = 0$.
3. Die Suche nach „Bedin“ mit Toleranz 0 liefert kein Ergebnis.
   $\Rightarrow d_{\text{me1}}(\text{Wurzel}, \{ort\}) = 1$
4. Erzeugen des neuen Kind-Suchknotens für das Attribut *plz* und die Toleranz $e = 0$.
5. Die Suche nach „10963“ mit Toleranz 0 liefert als Ergebnis die Menge $\{10963\}$.
   Erzeugen des neuen Kind-Wertknotens für $r = (?, \text{„10963“})$
6. Erzeugen des neuen Kind-Attributknotens für *ort*.
7. Erzeugen des neuen Kind-Suchknotens für das Attribut *ort* und die Toleranz $e = 1$.[14]
8. Die Suche nach „Bedin“ mit Toleranz 1 liefert kein Ergebnis.
   $\Rightarrow d_{\text{me1}}(\text{Wurzel} \to plz \to 0 \to 10963, \{ort\}) = 2$, $d_{\text{me1}}(\text{Wurzel}, \{ort, plz\}) = 1$
9. Die Suche nach „Bedin“ mit Toleranz 1 liefert kein Ergebnis.
   $\Rightarrow d_{\text{me1}}(\text{Wurzel}, \{ort\}) = 2$
10. Die Suche nach „Bedin“ mit Toleranz 2 liefert als Ergebnis die Menge $\{Berlin\}$.
    Erzeugen des neuen Kind-Wertknotens für $r = (\text{„Berlin“}, \text{„10963“})$.
11. Der Datensatz $r = (\text{„Berlin“}, \text{„10963“})$ ist vollständig definiert. Fertig.

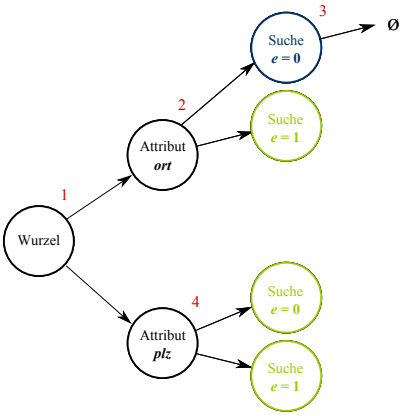

Abbildung 5.7.: Beispiel (Schritte 1–4)

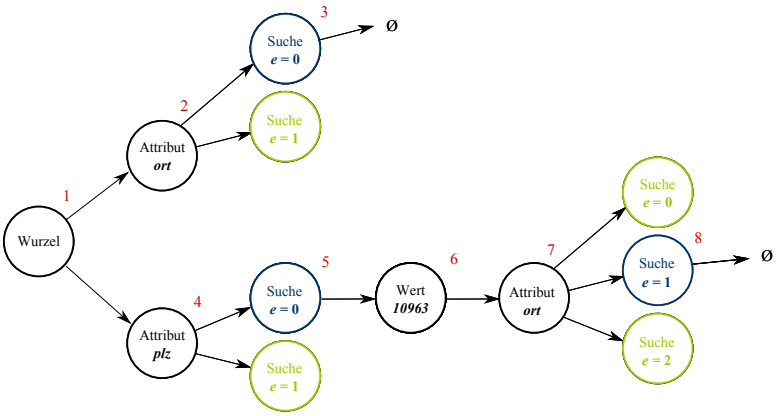

Abbildung 5.8.: Beispiel (Schritte 5–8)

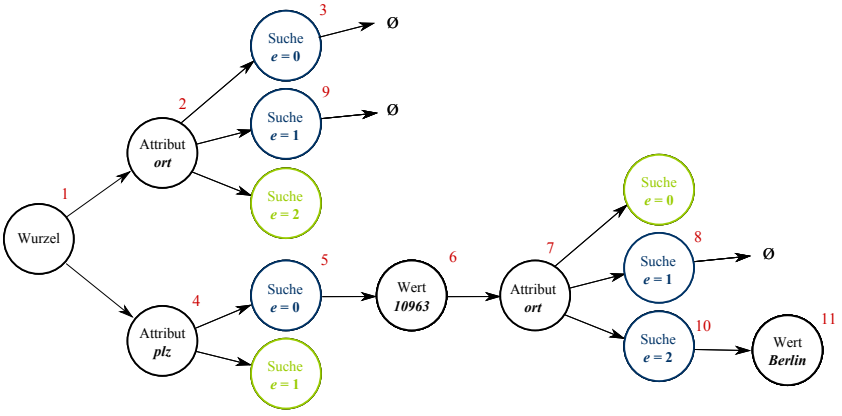

Abbildung 5.9.: Beispiel (Schritte 9–11)

Wenn nicht nur der ähnlichste, sondern mehrere ähnliche Datensätze gesucht werden sollen, würde die Suche fortgesetzt werden. Hier ist jedoch nur der ähnlichste Datensatz zum Anfragedatensatz $q = (\text{„Bedin“}, \text{„10963“})$ gesucht und das ist in diesem Beispiel der Datensatz $r = (\text{„Berlin“}, \text{„10963“})$.

An diesem Beispiel ist auch zu sehen, dass wenig Suchanfragen mit einer hohen Suchtoleranz durchgeführt werden. In den Indizes der ersten Kategorie wird zwei mal mit Toleranz 0 und einmal mit Toleranz 1 gesucht. In den Indizes der zweiten Kategorie wird einmal mit Toleranz 1 und einmal mit Toleranz 2 gesucht. Die Indizes der zweiten Kategorie sind jedoch nicht nur in diesem Beispiel, sondern auch in der Praxis oftmals sehr klein, so dass eine Suche dort auch mit einer höheren Toleranz noch recht effizient durchführbar ist.

5.3.6. Analyse

In diesem Abschnitt wird die vorgeschlagene Indexstruktur analysiert und dabei der Speicherbedarf, die Aufbauzeit und die Anfragezeit betrachtet. Die Anzahl der Datensätze wird mit n bezeichnet, die Anzahl der Attribute eines Datensatzes mit m und die maximale Anzahl an Synonymen für einen Attributwert mit S.

Speicherbedarf

Für jedes der m Attribute A_i gibt es einen Index I_i. Die Größe eines solchen Tries hängt von der konkreten Verteilung der Werte in der Datenbank für dieses Attribut ab, wächst jedoch im schlechtesten Fall linear mit der Anzahl der enthaltenen Werte. Die Anzahl der Werte ist hier durch die Anzahl der Datensätze n multipliziert mit der maximalen Anzahl an Synonymen S nach oben beschränkt. Ein Index I_i benötigt also einen Speicherplatz von $O(n \cdot S)$, die m Indizes zusammen benötigen damit $O(n \cdot S \cdot m)$.

Außerdem gibt es die Indizes I_i^x, deren Größe im Folgenden abgeschätzt wird. Für jeden Datensatz $r \in D$ wird jeder Attributwert $r.a_j$ in die $m-1$ Indizes I_i^x der anderen Attributwerte $x = r.a_i$ mit $i \neq j$ eingefügt. Außerdem können jeweils im schlechtesten Falle S weitere Werte für die Synonyme eingefügt werden. Pro Datensatz kommen also bis zu $m \cdot (m-1) \cdot (1+S)$ Werte hinzu. (Wenn ein Attributwert schon in einem Index vorhanden ist, wird er nicht erneut hinzugefügt; der angegebene Wert ist somit eine obere Schranke.) Insgesamt ergibt sich also ein Speicherbedarf für die Indizes der zweiten Kategorie von $O(n \cdot m^2 \cdot (1+S))$.

Insgesamt ergibt sich damit ein Speicherbedarf von $O(n \cdot m^2 \cdot (1+S))$, was insbesondere linear in der Anzahl der Datensätze der Datenbank ist. Eine experimentelle Untersuchung der Größe der Indexstruktur folgt in Kapitel 8.

Aufbauzeit

Die Aufbauzeit der Indexstruktur ist ebenfalls $O(n \cdot m^2 \cdot S)$. Im Algorithmus 5.2 gibt es drei ineinander verschachtelte Schleifen mit jeweils n, m und $m-1$ Durchläufen, und bei jedem Durchlauf werden bis zu S Werte eingefügt.

Anfragezeit

In diesem Abschnitt wird die Zeit der Anfragebeantwortung einer Ähnlichkeitssuche in der FRI-Datenstruktur untersucht. Es wird eine Ähnlichkeitssuche betrachtet, bei welcher der

[14] Beim Attribut *ort* muss nicht mehr mit Toleranz 0 gesucht werden, weil die mindestens erwartete Distanz für dieses Attribut 1 ist. Der erste Suchknoten hat hier daher schon die Suchtoleranz 1. (Dies wurde bei Algorithmus 5.4 bereits kurz angesprochen.)

ähnlichste Datensatz r zu einem Anfragedatensatz q gefunden werden soll. Die Zeit wird in Abhängigkeit von der Anzahl der Datensätze n, der Anzahl der Attribute m und des Abstands $e := \delta(q, r)$ abgeschätzt und mit $T_{\text{FRI}}(n, m, e)$ bezeichnet. Sie wird basierend auf der Laufzeit $T_{\text{Trie}}(n, e')$ für eine Ähnlichkeitssuche mit Toleranz e' in einem Trie mit n Elementen angegeben.

Diese Zeit ist für den allgemeinen Fall nur schwer analytisch bestimmbar, weil sie (wie schon bei Tries) sehr stark von der konkreten Verteilung der Werte in der Datenbank abhängt. Die asymptotische Laufzeit wird hier daher unter zwei vereinfachenden Annahmen bestimmt:

1. Alle m Attributwerte von r haben den gleichen Abstand: $\forall_i$: $\delta(q.a_i, r.a_i) = \frac{e}{m}$.[15]
2. Jeder Attributwert des ähnlichsten Datensatzes r ist auch einzeln betrachtet schon der ähnlichste Wert in der Datenbank: $\forall_i \forall_x$: $\delta(q.a_i, x.a_i) \geq \frac{e}{m}$.

Die zweite Annahme hat sich in der Praxis als sehr zutreffend herausgestellt. Bei der Adresskorrektur beispielsweise ist in den meisten Fällen der ähnlichste Wert für Attribut auch schon der eigentlich korrekte Wert. Die erste Annahme dient vor allem einer einfacheren Analyse. Wenn die Abstände ungleich auf die Attribute verteilt sind, sollte der Algorithmus sogar schneller den korrekten Datensatz finden, weil dann mindestens ein Attribut mit geringerem Abstand als $\frac{e}{m}$ gibt.

Im Folgenden wird die Laufzeit der gesamten Suchanfrage über die Laufzeit der Suchen in den Indexstrukturen abgeschätzt. In den Indizes der ersten Kategorie wird dann für jedes Attribut jeweils mit Toleranz 1, 2 usw. bist $\frac{e}{m}$ gesucht. (Bis auf die letzte Suche gibt es jeweils kein Suchergebnis.) Die Laufzeit von diesen Suchen wird asymptotisch von der Suche mit der größten Toleranz dominiert. Die Anfragezeit dieser Suche mit Toleranz $\frac{e}{m}$ kann mit der Laufzeit der Suche in einem Trie $T\left(n, \frac{e}{m}\right)$ abgeschätzt werden.[16] Die Größe eines Indexes ist beim FRI durch die Anzahl der Elemente in der Datenbank nach oben begrenzt. Die Suche bezüglich der m Attribute in den Indizes der ersten Kategorie benötigt also im ungünstigsten Fall $O\left(m \cdot T_{\text{Trie}}\left(n, \frac{e}{m}\right)\right)$ Zeit.

Sobald ein Attribut mit Toleranz $\frac{e}{m}$ durchsucht wird, wird ein Wert gefunden und dieser Wert ist genau der Wert des ähnlichsten Datensatzes.[17] Zu diesem Wert wird dann in einem Index der zweiten Kategorie ebenfalls mit Toleranz $\frac{e}{m}$ gesucht und dabei ein weiterer Wert des Ergebnisdatensatzes bestimmt. (Mit einer geringeren Suchtoleranz muss hier nicht gesucht werden, weil durch die mindestens erwartete Distanz ausgeschlossen werden kann, dass ein solcher existiert.) Die Suche wird fortgesetzt, der nächste Wert bestimmt und so weiter bis der Datensatz vollständig definiert ist. In den Indizes der zweiten Kategorie finden also $m-1$ Suchen jeweils mit Toleranz $\frac{e}{m}$ statt. Die Suche in den Indizes der zweiten Kategorie benötigt im ungünstigsten Fall also $O\left(m \cdot T_{\text{Trie}}\left(n, \frac{e}{m}\right)\right)$ Zeit. Dies entspricht der asymptotischen Laufzeit der Suche in den Indizes der ersten Kategorie.

Insgesamt ergibt sich für die Suche des ähnlichsten Datensatzes (unter den beiden genannten vereinfachenden Annahmen) also eine asymptotische Laufzeit von:

$$T_{\text{FRI}}(n, m, e) = O\left(m \cdot T_{\text{Trie}}\left(n, \frac{e}{m}\right)\right)$$

In Abschnitt 8.4 wird die Laufzeit der Datenstruktur in der Praxis untersucht.

[15] Hier wird zur einfachen Notation vorausgesetzt, dass die Toleranz e ganzzahlig durch die Anzahl der Attribute m teilbar ist. An der asymptotischen Laufzeit ändert diese Annahme nichts.

[16] $T_{\text{Trie}\left(n, \frac{e}{m}\right)}$ bezeichnet die Anfragezeit eines Tries mit n Elementen und Suchtoleranz e (siehe Abschnitt 5.2.3).

[17] Zu diesem Zeitpunkt ist die insgesamt mindestens erwartete Distanz für einen Ergebnisdatensatz gleich e. In den Indizes der ersten Kategorie wird nun nicht mehr gesucht, weil der nächste Suchknoten der ersten Ebene eine Suchtoleranz größer als $\frac{e}{m}$ hätte, wodurch die mindestens erwartete Distanz größer als e ist.

5.3.7. Erweiterung

In diesem Abschnitt schlage ich eine Modifikation der Datenstruktur vor, die den Speicherplatzbedarf reduzieren und auch zu einer Beschleunigung der Aufbau- und Anfragezeit führen soll. Das Prinzip ist hier ähnlich der Idee von Burst-Tries, siehe dazu Abschnitt 3.2.3 oder auch [HZW02].

Die Trie-Datenstruktur eignet sich gut, wenn große Mengen von Zeichenketten gespeichert und durchsucht werden sollen. Bei kleinen Datenmengen sind sie jedoch auf Grund der Verwaltung der Baumstruktur mit einem relativ hohen Speicherbedarf verbunden. Auch die Anfragezeit kann für sehr kleine Mengen bei einem Trie höher sein, als wenn man alle Elemente linear durchläuft. (Ich habe beispielsweise experimentell ermittelt, dass bei $n \leq 40$ Elementen und einer Suchtoleranz von $e \geq 4$ eine Suche in einer verketteten Liste schneller ist, als in einem Trie.)

Die in Abschnitt 5.3.3 vorgeschlagene Datenstruktur verwendet unter anderem Indizes I_i^x, die für einen Wert x die zugehörigen Werte für das Attribut A_i speichern. Diese Indizes umfassen teilweise nur recht wenige Elemente.[18]

Ich schlage also folgende Modifikation vor: Falls ein Index weniger als $minSizeTrie$ Elemente enthält, wird er nicht als Trie, sondern beispielsweise als Liste der Werte gespeichert. (Falls die Werte ohnehin an einer anderen Stelle explizit gespeichert sind, genügt eine Liste von Verweisen auf diese Werte.) Implementiert wurde dieses Verfahren mit einer Datenstruktur, die automatisch von einer Liste auf einen Trie erweitert wird, falls mehr als $minSizeTrie$ Elemente eingefügt werden.

Bei der Verwendung einer Liste anstatt eines Tries muss der oben angegebene Algorithmus an einigen Stellen modifiziert werden. Anstatt die Indexstruktur mit schrittweise höherer Toleranz zu durchsuchen, können bei kleinen Datenmengen sofort alle Elemente betrachtet werden. Auch die Berechnung der mindestens erwarteten Distanz (Abschnitt 5.3.5) muss leicht modifiziert werden. Auf die Details wird hier jedoch nicht genauer eingegangen. Eine Evaluation im Vergleich zur herkömmlichen Variante findet sich in Abschnitt 8.4.2.

Berechnung der Editierdistanz

Die Editierdistanz wurde mittels dynamischer Programmierung implementiert, die sich auf Grund der rekursiven Definition gut eignet und auch das Standardverfahren zur Berechnung ist. Für die Berechnung der einfachen Editierdistanz habe ich jedoch drei Optimierungen vorgenommen, um die Effizienz zu erhöhen.

Die Anfragen der Indexstruktur suchen zu einer gegebenen Zeichenkette s alle solchen Zeichenketten t mit $\delta_{\text{edit}}(s,t) \leq e$. Wenn sich die Länge der Zeichenketten um mehr als e unterscheidet, kann auf die Berechnung der exakten Editierdistanz verzichtet werden, weil mindestens $||s| - |t||$ Operationen nötig sind, um die Längendifferenz auszugleichen. Somit können insbesondere bei einer geringen Suchtoleranz e viele Zeichenketten schon von vornherein verworfen werden.

Eine weitere Abschätzung geschieht mit der Hamming-Distanz. Dabei soll auf die Berechnung der Editierdistanz verzichtet werden, wenn die Zeichenketten ohnehin sehr ähnlich sind. Wenn $\delta_{\text{ham}}(s,t) \leq e$, dann ist auch $\delta_{\text{edit}}(s,t) \leq e$. Durch Ausnutzung dieser Regel kann insbesondere für $s = t$ oder wenn nur einzelne Buchstaben vertauscht sind, auf die aufwändigere dynamische Programmierung verzichtet werden.

[18] Bei der Adresskorrektur enthalten viele Indizes nur einen Wert, so enthält beispielsweise I_{ort}^{14165} nur „Berlin“ und nur ungefähr 1 % der Orte besitzt mehr als eine Postleitzahl.

Auch die dynamische Programmierung zur Berechnung der Editierdistanz kann effizienter gestaltet werden. Die Idee stammt von Ukkonnen [Ukk85] und bewirkt, dass nicht die gesamte Matrix berechnet wird. Bei der spaltenweisen Iteration durch die Matrix wird die Berechnung abgebrochen, wenn ein gewisser Schwellwert erreicht ist (engl.: cutoff). Das Verfahren wird in [SM96] detaillierter erläutert.

Die Auswirkungen dieser drei Optimierungen auf die Laufzeit der Berechnung werden in Abschnitt 8.4.5 evaluiert.

5.3.8. Parameter

Die FRI-Datenstruktur besitzt einige Parameter, die zum einen die Ergebnisse und zum anderen die Effizienz der Berechnung beeinflussen. Diese Parameter werden in diesem Abschnitt zusammenfassend beschrieben.

Basis des Indexes ist eine Datenbank D von Datensätzen der Form $(a_1, \ldots, a_m)$. Außerdem gibt es für jedes Attribut A_i gegebenenfalls Synonyme, die mit Synonyme_{A_i} bezeichnet werden.

Für jedes Attribut A_i kann prinzipiell ein eigenes Abstandsmaß δ_i verwendet werden. Dies muss jedoch von der zu Grunde liegenden Trie-Datenstruktur unterstützt werden. Momentan sind dies unter anderem die Hamming-Distanz (Abschnitt 4.2.1) und die einfache Editierdistanz (Abschnitt 4.2.3).

Das Abstandsmaß für Datensätze kann mit Hilfe von Gewichten g_i für jedes Attribut A_i bestimmt werden.

Der Parameter a bei der Ähnlichkeitssuche gibt an, wann die Suche abgebrochen werden soll. Der Algorithmus kann beispielsweise abgebrochen werden, wenn genügend Datensätze gefunden wurden (Top-k-Anfrage): $a =$ „$|\text{ergebnisse}| \geq k$“ mit $k \in \mathbb{N}$. Der Algorithmus kann jedoch auch für eine Bereichsanfrage verwendet werden. Dann werden alle Datensätze gesucht, die innerhalb einer gewissen Suchtoleranz liegen: $a =$ „$\delta_{\min}(q, r) \geq e_{\max}$“ mit $e_{\max} \in \mathbb{R}$.

Außerdem gibt es zwei Parameter, die nicht die Ergebnisse, sondern die Effizienz der Berechnung beeinflussen: Der Parameter $minSizeTrie \in \mathbb{N}$ bestimmt die Grenze, ab der ein Trie anstatt als einfache Liste als Indexstruktur verwendet wird. Er beeinflusst damit sowohl den Speicherbedarf als auch die Anfragezeit.

Der Parameter *prio* bestimmt, wie die Priorität der Knoten im Entscheidungsbaum berechnet wird, wenn zwei Knoten den gleichen mindestens zu erwartenden Abstand haben. Er beeinflusst nur die Anfragezeit bei der Ähnlichkeitssuche.

5.3.9. Zusammenfassung

In den vorangegangenen Abschnitten wurde der FRI als eine Indexstruktur für Datensätze vorgeschlagen. Er besteht aus mehreren Tries. Ziel bei der Ähnlichkeitssuche in dieser Indexstruktur ist es, die Suche mit einer hohen Suchtoleranz zu vermeiden und stattdessen mehrere Suchen mit einer geringeren Toleranz durchzuführen. Dazu werden die Attribute der Ergebnisdatensätze nacheinander mit Werten belegt. Der Suchalgorithmus arbeitet mit einem Entscheidungsbaum und die Suche wird immer an der Stelle im Baum fortgesetzt, an der die geringste Distanz für den Ergebnisdatensatz zu erwarten ist. Bei der Berechnung der minimal zu erwartenden Distanz werden auch die Ergebnisse der Suchen in anderen Teilen des Baums berücksichtigt.

Die Laufzeit der Ähnlichkeitssuche eines Datensatzes mit Abstand e wurde unter zwei vereinfachenden Annahmen mit $O\left(m \cdot T_{\text{Trie}}\left(n, \frac{e}{m}\right)\right)$ abgeschätzt.

Es wurde eine Erweiterung vorgeschlagen, bei der kleine Datenmengen durch einfache Listen anstatt durch Tries indiziert werden. Die Berechnung der Editierdistanz für die Elemente dieser Listen wird durch Abschätzungen anhand der Länge und der Hamming-Distanz sowie durch den Ukkonen-Cutoff optimiert.

Die FRI-Datenstruktur ist flexibel bezüglich des Abstandsmaßes der einzelnen Attribute. Es ist beispielsweise die einfache Editierdistanz, die normierte Editierdistanz oder eine gewichtete Editierdistanz denkbar. Dabei müssen nicht alle Attribute das gleiche Abstandsmaß verwenden.

Die Gewichtung der Attribute, also die Ähnlichkeitsfunktion für Datensätze, kann mit Hilfe von Gewichten eingestellt werden. Somit kann ausgenutzt werden, dass es in vielen Anwendungsfällen Attribute gibt, die weniger mit Fehlern behaftet sind als andere.

Der Suchalgorithmus ist außerdem flexibel bezüglich des Abbruchkriteriums der Suche und kann somit sowohl für Top-k-Anfragen als auch für Bereichsanfragen verwendet werden.

5.4. Zusammenfassung

Dieses Kapitel stellte Indexstrukturen für Zeichenketten und Datensätze vor. Zunächst wurden dabei die Anforderungen aufgeführt und Gegebenheiten, welche die Konstruktion solcher Indizes erleichtern können. Abschließend können Antworten auf die in der Einleitung (Abschnitt 1.4) aufgeworfenen Fragen 3 und 4 gegeben werden:

3. *Mit welcher Datenstruktur kann effizient nach ähnlichen Zeichenketten gesucht werden?*

   Ein Trie ist eine geeignete Datenstruktur, wenn große Mengen von Zeichenketten abgespeichert werden sollen. Er eignet sich auch insbesondere dann, wenn eine effiziente Ähnlichkeitssuche in den Zeichenketten möglich sein soll. Das Ähnlichkeitsmaß kann beispielsweise durch die Editierdistanz gegeben sein. Suchen mit einer hohen Toleranz sollten jedoch vermieden werden, weil die Laufzeit der Suche exponentiell mit der Suchtoleranz steigt.

4. *Mit welcher Datenstruktur kann effizient nach ähnlichen Datensätzen gesucht werden?*

   Der im Rahmen dieser Arbeit einwickelte FRI ist eine Datenstruktur um eine Ähnlichkeitssuche in Datensätzen durchzuführen. Die Attributwerte werden dabei in mehreren Tries indiziert, die dann bei einer Ähnlichkeitssuche verwendet werden können. Die vorgeschlagene Indexstruktur ist flexibel bezüglich des eingesetzten Ähnlichkeitsmaßes und der Gewichtung der Attribute und ermöglicht die effiziente Beantwortung von Bereichs- und Top-k-Anfragen.

6. Adresskorrektur

Dieses Kapitel beschreibt das im Rahmen dieser Arbeit entstandene System zur Adresskorrektur. Dabei wird die von der OCR gelesene Zeichenkette zunächst vorverarbeitet und in die Adressbestandteile zerlegt. Zu dieser möglicherweise fehlerhaften Adresse wird mit Hilfe der FRI-Datenstruktur eine Menge von ähnlichen Adressen bestimmt. Die erhaltenen Ergebniskandidaten werden anschließend nach einer feineren Ähnlichkeitsfunktion sortiert. Das Adresskorrektur-Verfahren ist in Abbildung 6.1 dargestellt (siehe dazu auch Abbildung 1.3 auf Seite 11).

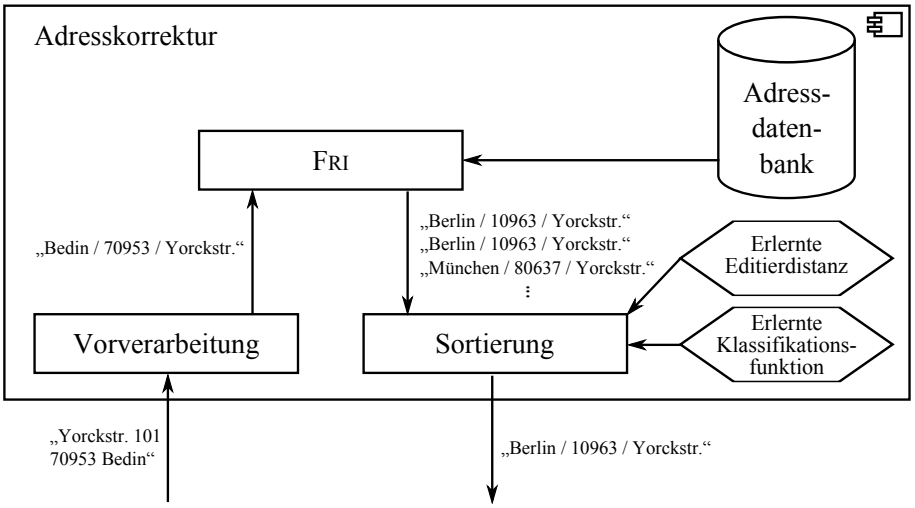

Abbildung 6.1.: Systemarchitektur der Adresskorrektur

Zunächst werden jedoch die verwendete Datenbasis und die Generierung der Datensätze und Synonyme beschrieben. Anschließend wird das Verfahren zur Adresskorrektur detaillierter erläutert.

6.1. Datenbank

Die Grundlage, um eine Korrektur von Adressen durchführen zu können, ist eine Datenbasis, die die korrekten Adressen enthält. An diese Datenbank werden einige Anforderungen gestellt, die im Folgenden kurz aufgeführt werden. Die Daten sollten möglichst vollständig sein, so dass nur in wenigen Fällen eine Adresse verworfen wird, weil sie nicht enthalten ist. Neben Straßen sollten auch Großempfänger und Postfächer verzeichnet sein, denen jeweils eigene Postleitzahlbereiche zugeordnet sind. Außerdem sollte die Datenbank regelmäßig aktualisiert werden, damit beispielsweise auch Umbenennungen von Straßen oder Zusammenlegungen von Orten im Adresskorrektur-System nachvollzogen werden können. Die Firma POSTCON

ist momentan nur in Deutschland tätig, so dass sich auch die Datenbank auf den deutschen Raum beschränken kann.

Zum Testen habe ich zunächst die frei verfügbare Datenbank OpenGeoDB [Hop05] verwendet, die die meisten deutschen Städte und Gemeinden mit ihren Geo-Koordinaten enthält. Sie enthält jedoch keine Straßennamen und eignete sich daher nur anfangs zum Testen.

Im fertigen System zur Adresskorrektur verwende ich die Datenbank Datafactory. Diese kann von der Deutschen Post erworben werden und wird quartalsweise aktualisiert. Sie umfasst alle Orte, Postleitzahlen und Straßen sowie Großempfänger und Postfachbereiche in Deutschland. Unter anderem sind dabei die folgenden Daten gespeichert: Zu jedem *Ort* sind der Name, gegebenenfalls ein Ortszusatz und ein verkürzter Ortsname sowie veraltete Ortsnamen und die Namen von Ortsteilen gespeichert. Zu jeder *Postleitzahl* ist der Ort gespeichert. Zu jeder *Straße* sind der Name, der Ort und die Postleitzahl sowie gegebenenfalls noch eine verkürzte Schreibweise des Straßennamens gespeichert. Zu jedem *Großempfänger* sind der Name, der Ort und die Postleitzahl gespeichert.[1] Zu jedem *Postfachbereich* sind die Postleitzahl und der Ort gespeichert.

6.1.1. Erzeugung der Datensätze

Dieser Abschnitt erläutert, wie aus dieser Datenbasis eine Datenbanktabelle mit den Attributen *ort*, *plz* und *str* sowie die Synonyme für die Attribute *ort* und *str* erzeugt werden. Diese Datenbank wird dann mit Hilfe der Fri-Datenstruktur indiziert und bei der Adresskorrektur verwendet.

- Die Straßen mit den Attributen Ortsname *ort*, Postleitzahl *plz* und Straßenname *str* können unverändert in die Datenbank übernommen werden.
- Die Postfächer besitzen nur die Attribute *ort* und *plz*. Für den Wert des Attributs *str* wird bei den Postfächern „Postfach“ verwendet, so dass sie im weiteren Verfahren genau wie normale Straßen behandelt werden können.
- Die Großempfänger erfordern jedoch eine separate Behandlung, weil Briefe an Großempfänger oftmals keinen Straßennamen haben und auch der Name des Empfängers stark variieren kann.[2] Für Großempfänger wird deshalb die Platzhalterzeichenkette „Großempfänger“ für das Attribut *str* verwendet und bei der Suche im Index dann separat behandelt (siehe Abschnitt 6.2.2).

6.1.2. Erzeugung der Synonyme

Zusätzlich zu den eigentlichen Datensätzen werden auch Synonyme für die Attribute *ort* und *str* erzeugt, um eine Adresse auch dann korrigieren zu können, wenn beispielsweise eine veraltete Bezeichnung für einen Ort verwendet wurde. Neben veralteten Bezeichnungen verwende ich auch Abkürzungen von Orts- und Straßennamen als Synonyme. Dadurch steigt zwar der Speicherbedarf der Indexstruktur, aber gleichzeitig sinkt die Anfragezeit, weil nur das Synonym und nicht der eigentliche Wert des Attributs in der Datenbank gefunden werden muss. In den folgenden beiden Absätzen wird erläutert, wie diese Mengen Synonyme_{ort} und Synonyme_{str} mit Hilfe der Datenbasis erzeugt wurden.[3]

[1] Beispielsweise Name = „Freie Universität Berlin“, Ort = „Berlin“, Postleitzahl = „14195“

[2] Beispielsweise wird „FU Berlin“ oder „Institut für Informatik“ anstatt von „Freie Universität Berlin“ auf den Brief geschrieben.

[3] Die Menge Synonyme_{plz} ist leer, weil es keine Synonyme für Postleitzahlen gibt.

Orte

Die Synonyme für Ortsnamen werden wie folgt generiert:

- Wenn der Ortsname ein Trennzeichen („/“, „(“, „)“, „ ,“, „-“, „&“, „.“) enthält, wird er in Teile mit Länge ≥ 5 aufgeteilt.
- Wenn der Ortsname Abkürzungen enthält, werden diese einmal abgekürzt und einmal ausgeschrieben.[4] Dafür wird eine Liste von 105 möglichen Abkürzungen verwendet (siehe Anhang A.3).
- Wenn zu dem Ort ein Ortszusatz[5] vermerkt ist, wird dieser Zusatz einmal an den Ortsnamen angehängt und einmal nicht.
- Wenn zu dem Ortsnamen veraltete Bezeichnungen vermerkt sind, werden diese als Synonym hinzugefügt.

Diese vier Verfahren zur Erzeugung von Synonymen werden kombiniert eingesetzt. Damit enthält die Menge Synonyme_{ort} beispielsweise folgende Synonyme für den Ort namens „Neukirchen-Adorf“ mit dem Ortszusatz „Erzgebirge“:

$$
\begin{aligned}
\text{Synonyme}_{ort} = \{ & \\
& \vdots \\
& \text{„Neukirchen-Adorf, Erzgebirge“} \leftarrow \text{„Adorf“} \\
& \text{„Neukirchen-Adorf, Erzgebirge“} \leftarrow \text{„Adorf, Erzgeb.“} \\
& \text{„Neukirchen-Adorf, Erzgebirge“} \leftarrow \text{„Adorf, Erzgebirge“} \\
& \text{„Neukirchen-Adorf, Erzgebirge“} \leftarrow \text{„Neukirchen“} \\
& \text{„Neukirchen-Adorf, Erzgebirge“} \leftarrow \text{„Neukirchen, Erzgeb.“} \\
& \text{„Neukirchen-Adorf, Erzgebirge“} \leftarrow \text{„Neukirchen, Erzgebirge“} \\
& \text{„Neukirchen-Adorf, Erzgebirge“} \leftarrow \text{„Neukirchen-Adorf“} \\
& \text{„Neukirchen-Adorf, Erzgebirge“} \leftarrow \text{„Neukirchen-Adorf, Erzgeb.“} \\
& \text{„Neukirchen-Adorf, Erzgebirge“} \leftarrow \text{„Neukirchen-Adorf, Erzgebirge“} \\
& \vdots \\
& \}
\end{aligned}
$$

Straßen

Auch für Straßen werden Synonyme generiert, wobei das Verfahren jedoch einfacher als bei Orten ist. Bei Straßen werden Abkürzungen ausgeschrieben und ein gegebenenfalls vorhandener verkürzter Straßenname verwendet. Die Synonyme für Straßennamen werden wie folgt generiert:

- Wenn der Straßenname Abkürzungen enthält, werden diese einmal abgekürzt und einmal ausgeschrieben.[6] Dafür wird eine Liste von 26 möglichen Abkürzungen verwendet (siehe Anhang A.3).

[4] Beispielsweise „St.“ und „Sankt“
[5] Beispielsweise „im Taunus“
[6] Beispielsweise „Str.“ und „Straße“ sowie „Fr.“ und „Friedrich“

- Wenn zu dem Ort ein kürzerer Straßenname[7] vermerkt ist, wird dieser als Synonym hinzugefügt.

Damit enthält die Menge Synonyme_{str} beispielsweise folgende Synonyme für die Straße namens „Prof.-Wilhelm-Thielmann-Str.“ mit dem verkürzten Namen „Prof.-W.-Thielmann-Str.“:

$$
\begin{aligned}
\text{Synonyme}_{str} = \{ & \\
& \vdots \\
& \text{„Prof.-Wilhelm-Thielmann-Str.“} \leftarrow \text{„Prof.-Wilhelm-Thielmann-Str.“} \\
& \text{„Prof.-Wilhelm-Thielmann-Str.“} \leftarrow \text{„Prof.-Wilhelm-Thielmann-Straße“} \\
& \text{„Prof.-Wilhelm-Thielmann-Str.“} \leftarrow \text{„Professor-Wilhelm-Thielmann-Str.“} \\
& \text{„Prof.-Wilhelm-Thielmann-Str.“} \leftarrow \text{„Professor-Wilhelm-Thielmann-Straße“} \\
& \text{„Prof.-Wilhelm-Thielmann-Str.“} \leftarrow \text{„Prof.-W.-Thielmann-Str.“} \\
& \text{„Prof.-Wilhelm-Thielmann-Str.“} \leftarrow \text{„Prof.-W.-Thielmann-Straße“} \\
& \vdots \\
& \}
\end{aligned}
$$

6.2. Verfahren

Für die OCR-Nachkorrektur von Adressen wird hier ein zweistufiges Verfahren vorgeschlagen, bei dem die FRI-Indexstruktur eine grobe Ähnlichkeitsfunktion verwendet und Ergebniskandidaten liefert. Diese werden in einem zweiten Schritt bezüglich einer genaueren Ähnlichkeitsfunktion sortiert. (Dieses Vorgehen ist analog zu dem bei der unscharfen Datenbankanfrage, wo ebenfalls zunächst Ergebniskandidaten erzeugt und dann bezüglich einer Ähnlichkeitsfunktion sortiert wurden. Siehe Abschnitt 2.3.)

Der Algorithmus zur Adresskorrektur erhält als Eingabe k Zeichenketten, die die Zeilen einer per optischer Zeichenerkennung gelesenen Adresse repräsentieren. Zunächst müssen die Adressbestandteile *ort*, *plz* und *str* aus dieser Zeichenkette extrahiert werden. Mit dem so konstruierten Datensatz (ort, plz, str) wird dann in der Datenbank mit Hilfe der FRI-Datenstruktur gesucht. Als Ähnlichkeitsfunktion zur Suche im Index wird eine gewichtete Summe der attributweisen einfachen Editierdistanz verwendet, weil diese relativ effizient berechenbar ist und von der gewählten Trie-Implementierung unterstützt wird. Anschließend werden die Ergebniskandidaten bezüglich der genaueren Ähnlichkeitsfunktion sortiert. Als feineres Ähnlichkeitsmaß kommt hier die mit Hilfe einer Support Vector Machine trainierte Klassifikationsfunktion zum Einsatz. Diese basiert auf der attributweisen gewichteten Editierdistanz mit den erweiterten Operationen zum Aufteilen und Verschmelzen von zwei Zeichen. Die Ausgabe des Algorithmus' ist dann die ähnlichste Adresse.

Algorithmus 6.1: Algorithmus zur Adresskorrektur

```
1 Adresskorrektur(s_1, ..., s_k) begin
2       q = ExtrahiereAdresse(s_1, ..., s_k)
3       kandidaten = FriSuche(q, Abbruchkriterium a)
4       if kandidaten = ∅
```

[7] Beispielsweise „Prof.-W.-Thielmann-Str.“ für „Prof.-Wilhelm.-Thielmann-Str.“

```
        error „Die Adresse kann nicht korrigiert werden.“
    end if
    sortiere die Elemente r ∈ kandidaten absteigend nach δ_svm(q, r)
    return erstes Element von kandidaten
end
```

Das Abbruchkriterium wird in Abschnitt 6.2.2 beschrieben. In den folgenden Abschnitten wird zunächst die Extraktion der Adressbestandteile *ort*, *plz* und *str* aus der gelesenen OCR-Zeichenkette erläutert. Außerdem wird die Verwendung der Datensatz-Indexstruktur für die Adresskorrektur beschrieben. Anschließend wird der Prozess der Sortierung der Ergebnisse bezüglich der genaueren Ähnlichkeitsfunktion dargestellt.

6.2.1. Vorverarbeitung

Das Ergebnis der optischen Zeichenerkennung ist die Repräsentation einer Adresse als Zeichenkette, die aus mehreren Zeilen besteht. Um damit eine Suche im Index durchführen zu können, müssen aus dieser Zeichenkette die einzelnen Adressbestandteile (insbesondere Ort, Postleitzahl und Straße) extrahiert werden. In diesem Abschnitt wird ein Algorithmus für dieses Problem angegeben.

Für die Suche im Index werden nur die Attribute *ort*, *plz* und *str* verwendet, der Vollständigkeit halber sind hier jedoch auch die Attribute *land*, *hausnr* und $empfaenger$ aufgeführt. Im Algorithmus wird zunächst die Zeile bestimmt, die die Postleitzahl enthält; ausgehend davon werden auch die übrigen Attribute extrahiert.

Algorithmus 6.2: Extraktion der Adressbestandteile

```
ExtrahiereAdresse(s_1, ..., s_k) begin
    for i = k down to 1 do
        if s_i enthält 5 aufeinerander folgende Zeichen, von denen mind. 3 Ziffern sind then
            break
        end if
    end for
    // i bezeichnet die Nummer der Zeile, die die Postleitzahl enthält

    if i = 1 then
        error „Die Adresse kann nicht zerlegt werden.“
    end if

    r := leere Adresse
    if i ≠ k then r.land  = s_{i+1}
    r.plz       = extrahiere PLZ aus s_i        // die gefundenen 5 Zeichen
    r.ort       = extrahiere Ort aus s_i        // der Rest dieser Zeile
    r.hausnr    = extrahiere Hausnr. aus s_{i-1} // im Wesentlichen die letzte Zifferngruppe
    r.str       = extrahiere Straße aus s_{i-1}  // der Rest dieser Zeile
    r.empfaenger = s_1 ∘ ... ∘ s_{i-2}           // die restlichen Zeilen am Anfang
    return r
end
```

Im Folgenden wird der Algorithmus kurz mit einem Beispiel erläutert. Die Eingabe der optischen Zeichenerkennung ist in Abbildung 6.2 sichtbar, das Ergebnis sind die folgenden fünf Zeilen.

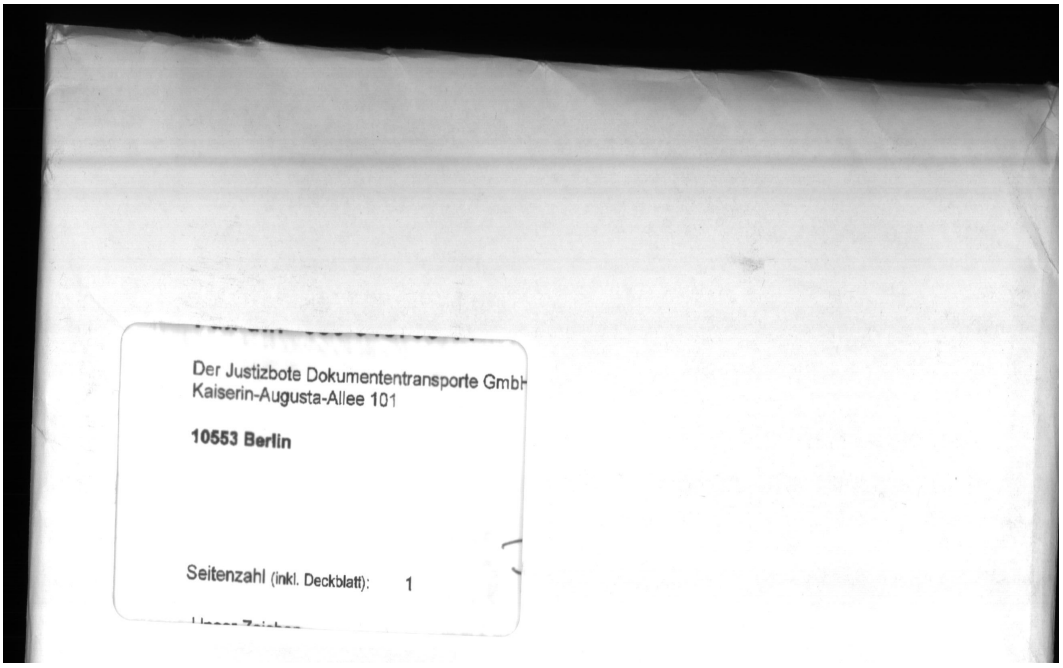

Abbildung 6.2.: Brief

$$
\begin{aligned}
s_1 &= \text{„Der Jusübote Dokumententransporte Gmb}\sim\text{“} \\
s_2 &= \text{„Kaisedn-Augusta-Allee 101“} \\
s_3 &= \text{„10663 Bedin“} \\
s_4 &= \text{„“} \\
s_5 &= \text{„Seitenzahl (inkl. DeckblaH): 1“}
\end{aligned}
$$

Der Algorithmus extrahiert dazu folgende Bestandteile:

$$
\begin{aligned}
r.land &= \text{„“} \\
r.plz &= \text{„10663“} \\
r.ort &= \text{„Bedin“} \\
r.hausnr &= \text{„101“} \\
r.str &= \text{„Kaisedn-Augusta-Allee“} \\
r.empfaenger &= \text{„Der Jusübote Dokumententransporte Gmb}\sim\text{“}
\end{aligned}
$$

Die Extraktion der Adressbestandteile funktioniert in diesem Beispiel korrekt. Es gibt jedoch Fälle, in denen die Zerlegung nicht korrekt funktioniert, beispielsweise, wenn der

Ortsname auf einer anderen Zeile als die Postleitzahl geschrieben ist. In Abschnitt 8.5 wird anhand von Testdaten untersucht, wie häufig die Bestandteile richtig extrahiert werden.

Manchmal enthält der so bestimmte Ortsname auch noch weitere Zusätze wie beispielsweise die Angabe eines Ortsteils (z. B. „Berlin-Zehlendorf“). Diese zusammengesetzte Zeichenkette kann so nicht direkt in der Datenbank gefunden werden, weil in der Datenbank nur „Berlin“ und „Zehlendorf“, nicht aber die Kombination abgespeichert ist. Bei längeren Zeichenketten für den Ortsnamen wird deshalb auch mit Teilen dieser Zeichenkette im Index gesucht. Dabei wird die Zeichenkette an Trennzeichen („/“, „(“, „␣“, „-“) zerteilt und die Suche auch mit den Teilen durchgeführt. Dieses Verfahren wird nicht nur für Ortsnamen, sondern auch für Straßennamen durchgeführt, damit Schreibweisen wie „Fliederweg 3b (Gartenhaus)“ erkannt werden können.

Anschließend wird gegebenenfalls die Groß- und Kleinschreibung der Adressbestandteile angepasst. Im Index sind die Daten mit der normalen Schreibweise[8] abgespeichert. Eine Anfrage in Versalschrift[9] könnte somit nicht korrigiert werden, weil jeder Buchstabe als falsch angesehen würde. Es wird daher folgendermaßen vorgegangen: Wenn mehr als die Hälfte der Zeichen Großbuchstaben sind, werden alle bis auf den jeweils ersten Buchstaben eines Wortes in die entsprechenden Kleinbuchstaben umgewandelt (andernfalls wird die Zeichenkette unverändert gelassen). Die gegebene Anfragezeichenkette würde also umgewandelt werden in „Sonthofen Im Allgäu“ und könnte somit leicht im Index gefunden werden.

Bemerkung Eine alternative Möglichkeit wäre es, von vornherein alle Zeichenketten im Index und alle Anfragezeichenketten in Kleinbuchstaben umzuwandeln. Jedoch kann bei dieser Vorgehensweise die korrekte Groß- und Kleinschreibung der Daten nicht mehr rekonstruiert werden.

6.2.2. Verwendung der Indexstruktur für Datensätze

In diesem Abschnitt wird beschrieben, wie die Fri-Datenstruktur bei der Adresskorrektur eingesetzt wird. Die Datensätze und Synonyme werden dabei wie in Abschnitt 6.1 beschrieben erzeugt.

Als Ähnlichkeitsmaß für die drei Attribute wird die einfache Editierdistanz verwendet: $\delta_{\text{ort}} = \delta_{\text{plz}} = \delta_{\text{str}} = \delta_{\text{edit}}$. Das Ähnlichkeitsmaß für Datensätze ist im Fri als Summe der attributweisen Distanz vorgegeben, wobei die drei Attribute jedoch unterschiedlich gewichtet werden können. Für Ort und Straße wird ein Gewicht von 1, für die Postleitzahl jedoch ein Gewicht von 2 verwendet. Damit kann ausgenutzt werden, dass beispielsweise die Postleitzahl nur in seltenen Fällen falsch ist, weil sie nur Ziffern enthält (siehe Abschnitt 8.3): $g_{ort} = 1$, $g_{plz} = 2$, $g_{str} = 1$. Diese Werte haben sich für die Generierung der Ergebniskandidaten als brauchbar herausgestellt. Eine genauere Gewichtung der Attribute findet ohnehin bei der anschließenden Sortierung mit Hilfe der Support Vector Machine statt.

Die eigentliche Datenstruktur muss bei der Adresskorrektur kaum verändert werden. Einzig die Behandlung der Großempfänger erfordert eine kleine Modifikation. Weil bei Großempfängern als Straßennamen lediglich die Zeichenkette „Großempfänger“ gespeichert ist, wird für eine Großempfänger-Postleitzahl jeder Straßenname bei einer Suchtoleranz von 5 automatisch akzeptiert. Ein zu kleiner Wert führt dazu, dass andere Adressen fälschlicherweise als Großempfänger erkannt werden, bei einem zu großen Wert kann der Datensatz nicht

[8] Im Wesentlichen werden nur die jeweils ersten Buchstaben eines Wortes groß geschrieben (z. B. „Sonthofen im Allgäu“).

[9] Alle Zeichen sind Großbuchstaben (z. B. „SONTHOFEN IM ALLGÄU“).

innerhalb der zur Verfügung stehenden Zeit korrigiert werden. Ich habe verschiedene Werte getestet und ein Wert von 5 liefert im Allgemeinen zufriedenstellende Ergebnisse.

Es kommen zwei Abbruchkriterien bei der Suche im FRI zum Einsatz. Das erste Abbruchkriterium bezieht sich auf den aktuell mindestens zu erwartenden Abstand für ein Ergebnisdatensatz. Wenn dieser den Abstand des ersten gefundenen Ergebnisdatensatzes genügend weit übersteigt, wird die Suche abgebrochen. Dadurch wird verhindert, dass Ergebniskandidaten generiert werden, die einen sehr hohen Abstand zum Anfragedatensatz haben. Als zweites Abbruchkriterium der Suche im FRI wird bei der Adresskorrektur ein Zeitlimit verwendet. Die Suche wird nach 100 ms abgebrochen, unabhängig davon wie viele Ergebniskandidaten gefunden wurden. Dies ist notwendig, um Echtzeitfähigkeit zu garantieren. Wenn bis zu diese Zeitpunkt keine Ergebnisse gefunden wurden, wird die Eingabeadresse als *falsch* verworfen. Wenn ein oder mehrere Ergebniskandidaten gefunden wurden, werden diese anschließend sortiert und der ähnlichste Datensatz als Ergebnis verwendet (siehe nächster Abschnitt). Die Sortierung kann sehr schnell durchgeführt werden, so dass das Zeitlimit durch die Sortierung nicht überschritten wird.

6.2.3. Sortierung

Anschließend an die Suche im Index werden die gefundenen Ergebnis-Kandidaten bezüglich eines genaueren Ähnlichkeitsmaßes sortiert. Dieses Ähnlichkeitsmaß ist die in Abschnitt 4.3 beschriebene Funktion δ_{svm}, die mit Hilfe einer Support-Vektor-Machine trainiert wurde (siehe Abschnitt 7.2). Als attributweises Ähnlichkeitsmaß wird dabei die gewichtete Editierdistanz mit erweiterten Operationen zur Verschmelzung und Aufspaltung von Zeichen verwendet. Sie kann eine genauere Aussage über die Ähnlichkeit von zwei Zeichenketten liefern als die einfache Editierdistanz. Die Gewichte der Editieroperationen werden dabei erlernt, so dass häufige Operationen mit einem geringeren Gewicht verbunden sind als seltene. Das Lernverfahren wird in Abschnitt 7.1 dargestellt. Die Auswirkung dieser Sortierung auf die Qualität der Ergebnisse der Adresskorrektur wird in Abschnitt 8.5.1 untersucht.

Auch bei der Sortierung gibt es einige kleine Anpassungen an die für die Adresskorrektur spezifischen Gegebenheiten. Unter anderem müssen Großempfänger wieder separat behandelt werden. Die Details werden hier jedoch nicht ausgeführt.

6.3. Zusammenfassung

In diesem Kapitel wurde das im Rahmen der Arbeit entwickelte System zur Adresskorrektur dargestellt. Die zur Korrektur verwendeten Daten stammen aus einer Datenbasis der DEUTSCHEN POST und enthalten alle in Deutschland gültigen Adressen. Bei der Korrektur einer Adresse werden zunächst die Adressbestandteile extrahiert und damit eine Suche in der FRI-Datenstruktur durchgeführt. Anschließend werden die so erhaltenen Ergebniskandidaten bezüglich einer feineren Ähnlichkeitsfunktion sortiert. Wie diese Ähnlichkeitsfunktion trainiert werden kann, beschreibt das folgende Kapitel .

7. Lernverfahren

In Kapitel 4 wurden die Ähnlichkeitsmaße von Zeichenketten und Datensätzen vorgestellt, die zur Sortierung der Ergebnisse bei der Adresskorrektur verwendet werden (Abschnitt 6.2.3). Für Zeichenketten ist dies die gewichtete Editierdistanz mit den erweiterten Operationen zum Aufteilen und Verschmelzen von Zeichen. Das Ähnlichkeitsmaß für Datensätze basiert auf einer Klassifikationsfunktion von Datensatzpaaren in die Klassen *ähnlich* und *nicht ähnlich*.

In diesem Zusammenhang wurde jedoch noch offen gelassen, wie die Gewichte der Editierdistanz gewonnen werden und welche Klassifikationsfunktion für die Datensätze verwendet wird. In diesem Kapitel wird erläutert, wie die Editiergewichte und die Datensatz-Klassifikationsfunktion jeweils mit Hilfe von Trainingsdaten erlernt werden können. Sie sollen dabei so optimiert werden, dass sie die Fehler und deren Häufigkeiten gut modellieren. Die Ähnlichkeitsmaße werden jeweils abgespeichert (siehe Abbildung 7.1) und dann bei der Sortierung der Ergebnisse verwendet (siehe dazu auch Abbildung 6.1 auf Seite 86).

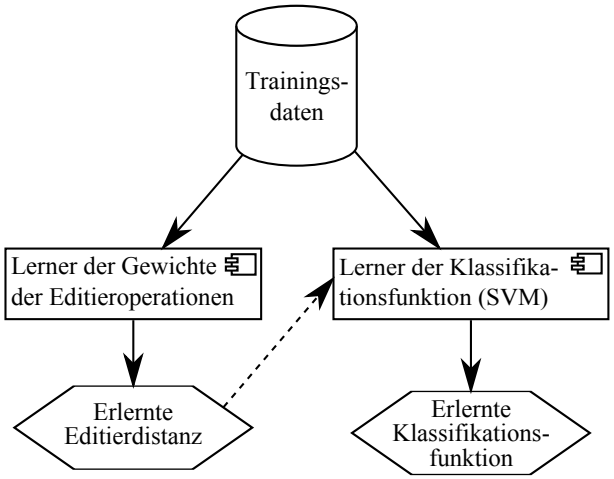

Abbildung 7.1.: Lernprozess

7.1. Lernen der Ähnlichkeit von Zeichenketten

Als Ähnlichkeitsmaß bei der Sortierung der Ergebniskandidaten der Adresskorrektur wird die gewichtete Editierdistanz mit den erweiterten Operationen zur Aufspaltung und Verschmelzung von Zeichenketten verwendet. Bisher wurde noch nicht erläutert, woher die Gewichte der Editieroperationen stammen. Ziel ist es, mit diesen Gewichten die Häufigkeiten der Fehler (insbesondere der OCR-Fehler, Abschnitt 2.5.2) gut zu modellieren. Die Gewichte werden dabei für jedes Attribut einzeln bestimmt. Im Folgenden wird beschrieben, wie die Gewichte

für *ein* Attribut erlernt werden können. Die Gewichte werden hier basierend auf Wahrscheinlichkeiten der elementaren Editieroperationen definiert. Diese Wahrscheinlichkeiten werden mit Hilfe von Trainingsdaten erlernt.

7.1.1. Wahrscheinlichkeit der korrekten Zeichenkette

Zu einer Anfragezeichenkette $s \in \Sigma^*$ für ein Attribut A_i soll der Wert $t \in A_i$ gefunden werden, der mit möglichst großer Wahrscheinlichkeit der korrekte Wert ist. Diese Wahrscheinlichkeit wird dabei häufig mit der Methode von Bayes modelliert [TE96; KR02; Sch03; Rin03; LH03]:

$$\mathrm{P}(t \mid s) = \frac{\mathrm{P}(s \mid t) \cdot \mathrm{P}(t)}{\mathrm{P}(s)}$$

- $\mathrm{P}(t \mid s)$ ist die Wahrscheinlichkeit, dass die Zeichenkette t korrekt ist, wenn s als OCR-Ergebnis geliefert wurde. Dies ist die Zielgröße, die maximiert werden soll.
- $\mathrm{P}(s \mid t)$ ist die Wahrscheinlichkeit, die Zeichenkette s als OCR-Ergebnis zu erhalten, wenn die Eingabe t war. Diese Wahrscheinlichkeiten können beispielsweise mit Hilfe einer Trainingsmenge erlernt werden. Dieses Lernverfahren wird im folgenden Abschnitt genauer beschrieben.
- $\mathrm{P}(t)$ ist die A-Priori-Wahrscheinlichkeit, dass die Zeichenkette t geschrieben wird. Diese Informationen kann beispielsweise mit Hilfe von Frequenzinformationen aus einer Trainingsmenge gewonnen werden.
- $\mathrm{P}(s)$ ist die A-Priori-Wahrscheinlichkeit, dass die Zeichenkette s gelesen wird. Dieser Term ist für alle Werte t der Datenbank konstant, weil er nur von der Anfragezeichenkette s abhängt. Er kann also ignoriert werden, wenn die Wahrscheinlichkeit von $\mathrm{P}(t \mid s)$ nicht konkret berechnet, sondern nur maximiert werden soll.

Bemerkung Die A-Priori-Wahrscheinlichkeiten $\mathrm{P}(t)$ für das Auftreten einer Zeichenkette t basieren auf den beobachteten Häufigkeiten in einer Trainingsmenge. Weil die hier verfügbare Trainingsmenge recht klein ist (ungefähr 4.000 Datensätze, siehe Abschnitt 8.2), kommen also nur wenige Zeichenketten (hier also Orte, Postleitzahlen und Straßen) überhaupt vor. Der Großteil der ungefähr 1,2 Mio Straßen wäre nicht von diesen Statistiken erfasst. Ich verzichte daher auf die Verwendung der A-Priori-Wahrscheinlichkeiten.

7.1.2. Wahrscheinlichkeit der OCR-Zeichenkette

Die Wahrscheinlichkeit, dass die Zeichenkette t korrekt ist, wenn die Zeichenkette s gelesen wurde, wird also mit der Wahrscheinlichkeit abgeschätzt, dass t bei der optischen Zeichenerkennung als s erkannt wurde.

Die Berechnung der Wahrscheinlichkeit $\mathrm{P}(s \mid t)$ kann auf Basis von Wahrscheinlichkeiten für die elementaren Editieroperationen (Ersetzung, Einfügung und Löschung von einzelnen Buchstaben) geschehen.[1] Diese Wahrscheinlichkeiten können beispielsweise mit einem Korpus von Trainingsdaten erlernt werden. Wenn in der Trainingsmenge zu jeder von der OCR-Software gelesenen Zeichenkette die eigentlich korrekte Zeichenkette bekannt ist, kann bestimmt werden, welche Fehler bei dieser OCR-Software wie häufig auftreten.

[1] Der Einfachheit halber werden die erweiterten Editieroperationen zur Verschmelzung und Aufspaltung von Buchstaben in den folgenden Absätzen nicht erwähnt. Sie können jedoch genau analog integriert werden.

Für die Berechnung der Wahrscheinlichkeit $\mathrm{P}(s \mid t)$, dass eine Zeichenkette t als s erkannt wird, wird häufig folgendes Verfahren verwendet [TE96; KR02; LH03; RY98; Rin03]. Die Wahrscheinlichkeit, dass anstatt der ersten j Buchstaben von t, die ersten i Buchstaben von s erkannt wurden, wird mit $\mathrm{P}(s_{[1..i]} \mid t_{[1..j]})$ bezeichnet und kann folgendermaßen angenähert werden:

$$\mathrm{P}(s_{[1..i]} \mid t_{[1..j]}) \approx \max \begin{cases} \mathrm{P}(s_{[1..i-1]} \mid t_{[1..j-1]}) \cdot \mathrm{P}(s_{[i]} \mid t_{[j]}) & \text{Substitution}[2] \\ \mathrm{P}(s_{[1..i]} \mid t_{[1..j-1]}) \cdot \mathrm{P}(\epsilon \mid t_{[j]}) & \text{Einfügung} \\ \mathrm{P}(s_{[1..i-1]} \mid t_{[1..j]}) \cdot \mathrm{P}(s_{[i]} \mid \epsilon) & \text{Löschung} \end{cases}$$

Dies ist aus drei Gründen nur eine Annäherung an die tatsächliche Wahrscheinlichkeit.

1. Es werden nur die definierten Editieroperationen betrachtet und darüber hinausgehende Fehler werden nicht erfasst.[3]

2. Es wird nur die wahrscheinlichste Sequenz von Operationen betrachtet, obwohl der Übergang von t nach s durch mehrere verschiedene Folgen entstanden sein kann.[4]

3. Die Wahrscheinlichkeit für das Eintreten mehrerer Editieroperationen wird als Produkt der einzelnen Wahrscheinlichkeiten berechnet. Dies setzt voraus, dass die Editieroperationen unabhängig voneinander auftreten [Sch03]. Dies ist unter anderem jedoch deshalb nicht der Fall, weil sich einige Fehler (engl.: burst errors) auf mehrere Buchstaben erstrecken.

Es besteht jedoch Grund zur Annahme, dass bei dieser Annäherung kein großer Fehler gemacht wird. Es sind nur wenige Fehler zu erwarten, die über die fünf definierten Editieroperationen (Substitution, Einfügung, Löschung, Verschmelzung, Auftrennen) hinausgehen. Auch die Betrachtung der wahrscheinlichsten Operationsfolge (und nicht aller möglichen Operationsfolgen) sollte keinen großen Fehler zur Folge haben, weil die Wahrscheinlichkeit der wahrscheinlichsten Operationsfolge die der anderen Folgen dominiert.

In diesem Abschnitt wurde erläutert, wie die Wahrscheinlichkeit eine Zeichenkette s anstatt einer Zeichenkette t zu erkennen, basierend auf den Wahrscheinlichkeiten der einzelnen Editieroperationen abgeschätzt werden kann. Wie Wahrscheinlichkeiten der Editieroperationen anhand der Häufigkeiten in einer Trainingsmenge erlernt werden können, wird im folgenden Abschnitt dargestellt.

7.1.3. Lernen der Wahrscheinlichkeiten der Editieroperationen

Gegeben ist eine Trainingsmenge von Paaren von Zeichenketten (s, t), dabei ist s die von der OCR-Software gelesene Zeichenkette und t der zugehörige korrekte Wert. Die Erzeugung einer solchen Trainingsmenge wird in Abschnitt 8.2 beschrieben.

Gesucht sind die Wahrscheinlichkeiten der elementaren Editieroperationen. Die Wahrscheinlichkeit $\mathrm{P}(u \mid w)$ der Ersetzung eines Buchstabens w durch u kann durch Betrachtung der Häufigkeiten in der Trainingsmenge bestimmt werden (die Berechnung erfolgt analog für

[2] Das Übernehmen eines Zeichens kann als Ersetzung durch sich selbst betrachtet werden und muss deshalb nicht separat behandelt werden.

[3] Beispielsweise das Aufteilen eines Buchstabens in drei Buchstaben: „W“ → „iii“

[4] Beispielsweise kann der Übergang „Berlin“ → „Bedin“ unter anderem durch folgende drei Operationsfolgen entstanden sein: („rl“ → „d“) oder („r“ → ϵ, „l“ → „d“) oder („r“ → „d“, „l“ → ϵ).

die anderen Operationen) [Rin03; KCG90; TE96]:

$$\mathrm{P}(u \mid w) = \frac{\#\text{Vorkommen der Editieroperation } w \to u}{\#\text{Vorkommen von } w \text{ in den korrekten Zeichenketten}}$$

Der Nenner des Bruchs, also die Anzahl der Vorkommen eines Buchstaben w in den Zeichenketten der Trainingsmenge, kann einfach abgezählt werden. Zur Bestimmung des Zählers, also der Häufigkeiten der einzelnen Editieroperationen, wird die Trainingsmenge einmal durchlaufen. Dabei wird für jedes Paar von Zeichenketten die Folge von Editieroperationen bestimmt. Dies kann unter anderem mit dem *Viterbi-Algorithmus* geschehen [RY98; PCAAL00; Sch03] ein dynamischer Programmierungs-Algorithmus, der in einem Verborgenen-Markow-Modell (engl.: hidden Markov model) die Folge der wahrscheinlichsten Operationen findet. Ich habe mich jedoch für ein einfacheres Verfahren entschieden und verwende hier die Folge der Operationen, die die geringste ungewichtete Editierdistanz hat. Dies ist eine Vereinfachung des Verfahrens, die dennoch brauchbare Ergebnisse liefert (siehe Abschnitt 8.5.1). Für jede der auftretenden Operationen wird ein Zähler um eins erhöht. Dieses Verfahren wird auch in folgenden Algorithmus dargestellt:

Algorithmus 7.1: Bestimmung der Wahrscheinlichkeiten der Editieroperationen

```
1  WahrscheinlichkeitenEditieroperationen(T) begin
2      var Buchstaben[]      // Abbildung Σ_ε → ℕ
3      var EditOps[]         // Abbildung Editieroperation → ℕ
4      var P[]               // Abbildung Editieroperation → [0, 1]
5
6      for (s, t) ∈ T do // s ist die OCR−, t die korrekte Zeichenkette
7          Ops = Folge der Editieroperationen von t → s
8          for op ∈ Ops do
9              EditOps[op]++ // Zählen der Häufigkeiten der Editieroperationen
10         end for
11         for i = 0 to |t| do
12             Buchstaben[t_[i]]++ // Zählen der Häufigkeiten der Buchstaben
13         end for
14     end for
15
16     // Wahrscheinlichkeiten der Editieroperationen berechnen
17     for (u → w) ∈ EditOps do
18         P[u → w] = EditOps[u → w] / Buchstaben[u]
19     end for
20
21     return P
22 end
```

Wenn die Trainingsmenge nicht sehr groß ist, werden einige Editieroperationen nicht beobachtet, obwohl sie in der Realität vorkommen. Damit diesen Operationen nicht die Wahrscheinlichkeit 0 haben, wird ihnen eine kleine Wahrscheinlichkeit α zugeordnet.[5] Damit sind die Werte aus wahrscheinlichkeitstheoretischer Sicht nicht mehr konsistent. Durch diese Modifikation ist das System jedoch auch einsetzbar, wenn die Trainingsmenge nicht sehr groß ist.

Die erlernten Editieroperationen sind mit den daraus resultierenden Wahrscheinlichkeiten in Abschnitt 8.3 beschrieben und in Anhang A.5 aufgelistet.

[5] Als Wert für diesen Parameter α habe ich 0.00001 gewählt.

7.1.4. Verbindung zum Editierabstand

In Abschnitt 7.1.2 wurde erläutert, wie die Wahrscheinlichkeit des Auftretens einer OCR-Zeichenkette s anstatt der eigentlich korrekten Zeichenkette t rekursiv auf Basis der Wahrscheinlichkeiten der Editieroperationen berechnet werden kann. In diesem Abschnitt wird erläutert, wie diese Berechnung mit Hilfe der gewichteten Editierdistanz vorgenommen werden kann (siehe beispielsweise [RY98]). Aufbauend auf den Ersetzungswahrscheinlichkeiten kann beispielsweise folgendermaßen ein Abstandsmaß für Zeichenketten definiert werden:

$$\delta_{\text{prob}}(s,t) := -\log(\text{P}(s \mid t))$$

Die Funktion δ_{prob} erfüllt trivialerweise die Eigenschaften eines Abstandsmaßes (Abschnitt 2.2). □

Das Abstandsmaß δ_{prob} kann folgendermaßen ausformuliert werden:

$$\delta_{\text{prob}}(s_{[1..i]}, t_{[1..j]}) = \min \begin{cases} \delta_{\text{prob}}(s_{[1..i-1]}, t_{[1..j-1]}) - \log(\text{P}(s_{[i]} \mid t_{[j]})) & \text{Substitution} \\ \delta_{\text{prob}}(s_{[1..i]}, t_{[1..j-1]}) - \log(\text{P}(\epsilon \mid t_{[j]})) & \text{Einfügung} \\ \delta_{\text{prob}}(s_{[1..i-1]}, t_{[1..j]}) - \log(\text{P}(s_{[i]} \mid \epsilon)) & \text{Löschung} \end{cases}$$

Diese Struktur entspricht genau der gewichteten Editierdistanz. Die Gewichte der Editieroperation $(u \to w)$ werden daher folgendermaßen gewählt:

$$d(u,w) := -\log(\text{P}(u \mid w))$$

Dadurch lässt sich die Wahrscheinlichkeit des Auftretens der Zeichenkette s anstatt der korrekten Zeichenkette t folgendermaßen mit Hilfe der gewichteten Editierdistanz berechnen:

$$\text{P}(s \mid t) = e^{-\delta_{\text{prob}}(s,t)}$$

Die Berechnung auf diese Weise ist auch effizienter als die direkte Berechnung der Wahrscheinlichkeiten (Abschnitt 7.1.2), weil hier nur addiert und nicht multipliziert werden muss. Dies kann auf Grund der quadratisch von der Länge der Zeichenketten abhängenden Laufzeit in der Praxis durchaus einen Unterschied machen, wurde hier jedoch nicht evaluiert.

7.1.5. Durchführung

Die Gewichte der Editieroperationen werden bei der Adresskorrektur für Ort und Straße einerseits sowie für die Postleitzahl andererseits separat gelernt. Dadurch kann berücksichtigt werden, dass bei der Postleitzahl wahrscheinlich andersartige Operationen und Häufigkeiten der Fehler auftreten, als bei Orts- und Straßennamen. Die Gewichte wurden mit der Trainingsmenge T_{alle} erlernt,[6] abgespeichert und dann bei der Sortierung der Ergebniskandidaten zur Berechnung des Abstands zum Anfragedatensazt verwendet (Abschnitt 6.2.3).

7.1.6. Zusammenfassung

In diesem Abschnitt wurde die Ähnlichkeit von Zeichenketten auf Wahrscheinlichkeiten basierend definiert. Dabei wurde die Wahrscheinlichkeit $\text{P}(t \mid s)$, dass ein Datenbankwert t der korrekte Wert zu einer Anfragezeichenkette s ist, zurückgeführt auf die Wahrscheinlichkeit

[6] Die Trainingsmengen werden in Abschnitt 8.2 erläutert.

$\mathrm{P}(s \mid t)$, dass die Anfragezeichenkette aus dem Datenbankwert entstanden ist. Weiterhin wurde gezeigt wie diese Wahrscheinlichkeit mit den Wahrscheinlichkeiten der elementaren Editieroperationen angenähert werden kann. Es wurde außerdem erläutert, wie die Wahrscheinlichkeiten der Editieroperationen mit Hilfe einer Trainingsmenge erlernt werden können.

7.2. Lernen der Ähnlichkeit von Datensätzen

In diesem Abschnitt wird beschrieben, wie das Ähnlichkeitsmaß für Datensätze mit Hilfe einer Trainingsmenge erlernt werden kann. In Abschnitt 4.3 wurde beschrieben, wie der Abstand eines Datensatzpaares (q, r) auf die Zugehörigkeit zur Klasse der *ähnlichen* beziehungsweise *nicht ähnlichen* Datensätze zurückgeführt werden kann:

$$\delta_{\mathrm{svm}}(q, r) = f\left(v_\delta\left(q, r\right)\right) + z$$

Hier wird nun beschrieben, wie die Klassifikationsfunktion f und damit auch das Abstandsmaß erlernt werden kann. Dieses Lernverfahren wird mit Hilfe einer Support Vector Machine durchgeführt. Diese wird im Folgenden vorgestellt. Anschließend wird die Erzeugung der Trainingsdaten und die Durchführung des Lernverfahrens erläutert.

7.2.1. Support Vector Machine

Eine Support Vector Machine ist ein binärer Klassifikator von Vektoren in einem Vektorraum. Sie soll zu einem Anfragevektor entscheiden, in welcher der beiden Klassen er sich befindet [Bur98; Wik08c]. Die Entscheidungsfunktion definiert eine Hyperebene im mehrdimensionalen Raum und wird mit Hilfe einer Trainingsmenge erlernt. Die Entscheidungsfunktion wird dabei so optimiert, dass die Hyperebene die Klassen mit einem möglichst breiten Rand trennt. Dadurch soll eine gute Klassifizierung der zum Trainingszeitpunkt unbekannten Anfragevektoren erreicht werden. Um einen breiten Rand zu erhalten, wird der Abstand der Vektoren, die der Ebene am nächsten liegen, maximiert.[7] Diese Vektoren werden als *Stützvektoren* (engl.: support vectors) bezeichnet.

Mathematische Formulierung

Gegeben ist eine Trainingsmenge $T = \{(x_1, c_1), \ldots, (x_n, c_n)\}$. Dabei ist jeweils $x_i \in \mathbb{R}^m$ ein Vektor und $c \in \{-1, +1\}$ die Klasse dieses Vektors.

Gesucht ist eine Hyperebene, die die Trainingsvektoren in zwei Klassen trennt. Eine solche Ebene kann mit Hilfe eines Normalenvektors w und eines Wertes b, der den Abstand der Ebene zum Ursprung angibt, repräsentiert werden. Die Punkte, die auf dieser Ebene liegen, erfüllen dann folgende Gleichung:

$$\langle w, x\rangle - b = 0$$

Davon ausgehend wird einem zu klassifizierenden Vektor x_i eine Klasse $c(x_i)$ zugeordnet, je nachdem auf welcher Seite der Hyperebene er sich befindet:

$$c(x_i) = \mathrm{sgn}(\langle w, x\rangle + b)$$

[7] Weil nur die der Trennebene am nächsten liegenden Vektoren betrachtet werden, hängen Support Vector Machines kaum von der Anzahl der Vektoren in beiden Klassen ab.

Lineare Trennbarkeit

In diesem Abschnitt wird der Fall behandelt, dass die Trainingsdaten linear trennbar sind (siehe Abbildung 7.2 links). In diesem Fall gibt es in der Regel unendlich viele mögliche Trennebenen. Von diesen soll diejenige Hyperebene ausgewählt werden, die den größten Abstand zu beiden Klassen hat. Das ist gleichbedeutend damit, dass der kleinste Abstand der Trainingsvektoren zur Trennebene maximiert werden soll. Dies kann folgendermaßen als Optimierungsproblem formuliert werden [Bur98]:[8]

$$\text{minimiere bezüglich } w,\, b\text{: } \frac{1}{2}\,||w||^2$$
$$\text{mit der Nebenbedingung: } \forall i : y_i \cdot (\langle w, x_i \rangle + b) \geq 1$$

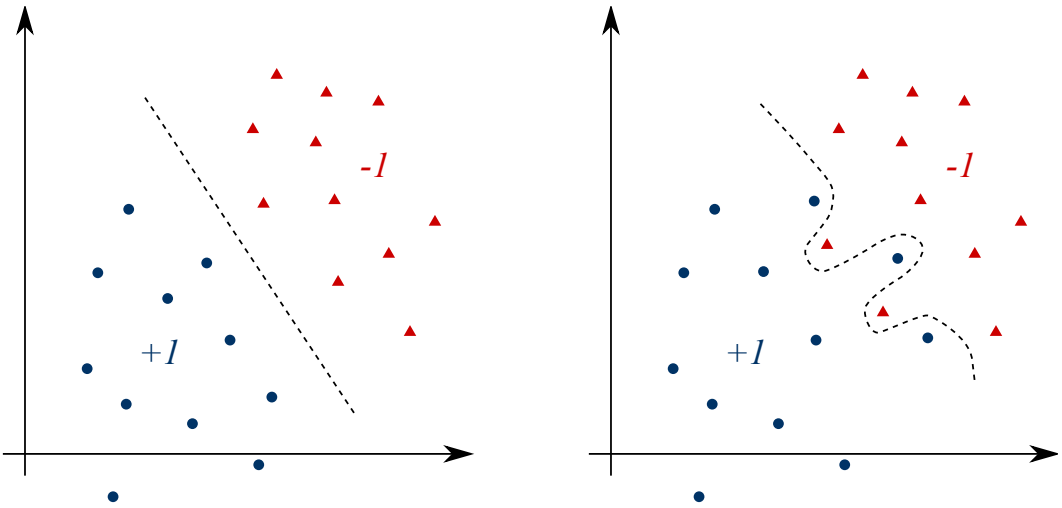

Abbildung 7.2.: Linear trennbare Daten und nicht linear trennbare Daten

Nicht-lineare Trennbarkeit

Wenn die Trainingsdaten hingegen nicht linear trennbar sind, gibt es keine solche Hyperebene (siehe Abbildung 7.2 rechts). In diesem Fall wird zugelassen, dass einige der Trainingsvektoren auf der „falschen“ Seite der Ebene liegen (auf engl. wird dies auch mit „soft margin“ bezeichnet). Solche falschen Klassifizierungen werden jedoch mit Kosten versehen, die dann ebenfalls minimiert werden sollen. Dies wird mit Hilfe von Schlupfvariablen ξ_i erreicht.

$$\text{minimiere bezüglich } w,\, b\text{: } \frac{1}{2}\,||w||^2 + C \cdot \sum_{i=1}^{m} \xi_i$$
$$\text{mit der Nebenbedingung: } \forall i : y_i \cdot (\langle w, x_i \rangle + b) \geq 1 - \xi_i$$

[8] Zur Lösung des Optimierungsproblem bei Support Vector Machines wird in der Regel das dazu duale Problem mit Quadratischer Programmierung gelöst.

Der Faktor C erlaubt hierbei eine Gewichtung zwischen einem möglichst großen Rand und möglichst wenig falschen Klassifizierungen der Trainingsmenge. Durch die Verwendung dieser Schlupfvariablen kann somit außerdem eine Überanpassung an die Trainingsdaten verhindert werden, weil Ausreißer als solche behandelt werden können.

Abbildung in höhere Dimension

Die Daten sind jedoch nicht immer gut durch eine Hyperebene trennbar. Die in diesem Absatz skizzierte Erweiterung bildet die Vektoren in eine höhere Dimension ab und führt dort die Klassifikation durch. In dieser höheren Dimension ist es dann leichter eine geeignete Trennebene zu finden.

Diese Abbildung in eine höhere Dimension findet implizit mit dem sogenannten *Kernel-Trick* statt. Die Details dieses Verfahrens sind in [Bur98] beschrieben. Es gibt verschiedene Arten von Kernels (unter anderem: polynomial, radiale Basisfunktion (RBF) und sigmoid), von denen hier die radiale Basisfunktion ausgewählt wird. Sie wird in [HCL03] empfohlen, unter anderem, weil sie mit nur einem einstellbaren Parameter γ auskommt:

$$K(x_i, x_j) = e^{-\gamma||x_i - x_j||^2} \text{ mit } \gamma > 0$$

7.2.2. Durchführung

In diesem Abschnitt wird die Durchführung des Lernverfahrens der Ähnlichkeitsfunktion für Datensätze beschrieben. Dabei wird zunächst die Erzeugung der Trainingsdaten und anschließend die Optimierung der Parameter der Support Vector Machine erläutert.

Erzeugung der Trainingsdaten

Es wird eine Menge von Trainingsdaten $T = \{(x_1, c_1), \ldots, (x_n, c_n)\}$ benötigt. Die Vektoren x_i mit Klasse $c_i = +1$ repräsentieren dabei Datensatzpaare, die als *ähnlich* angesehen werden. Vektoren x_i mit Klasse $c_i = -1$ stehen für Datensatzpaare, die als *nicht ähnlich* angesehen werden.

Die Vektoren der Klasse $+1$ wurden hier bei der Adresskorrektur erzeugt, indem einem per OCR gelesenen Datensatz der zugehörige korrekte Datensatz aus der Datenbank zugeordnet wurde. Diese Zuordnung erfolgte größtenteils manuell und wird in Abschnitt 8.2 beschrieben (Zum Lernen wird hier die Trainingsmenge T_{alle} verwendet). Ein solches Datensatzpaar ist beispielsweise (q, r_1) mit $q =$ („Bedin“, „10963“, „Mäckemstr.“) und $r =$ („Berlin“, „10963“, „Möckernstr.“). Dazu wird der entsprechende Vektor gebildet (siehe Abschnitt 4.3.2):[9]

$$v_\delta(q, r_1) = \begin{pmatrix} 2 \\ 0 \\ 2 \end{pmatrix}$$

Für Erzeugung der Vektoren der Klasse -1 sind Paare von Datensätzen gesucht, die *nicht ähnlich* sind. Es gibt viele verschiedene Möglichkeiten, solche Paare zu erzeugen. Ich habe zwei Verfahren erwogen, die in den folgenden Absätzen erläutert werden. Beim ersten Verfahren werden die per OCR gelesenen Datensätze zufällig mit den Datensätzen der Datenbank

[9] Für das Beispiel wurde der Einfachheit halber die einfache Editierdistanz als attributweise Distanz verwendet. In der Implementierung wird die gewichtete Editierdistanz mit erweiterten Operationen benutzt.

kombiniert. (Als *ähnlich* klassifizierte Datensatzpaare werden dabei ausgeschlossen.) Ein solches Datensatzpaar ist beispielsweise (q, r_2) mit $q = (\text{„Bedin“}, \text{„10963“}, \text{„Mäckemstr.“})$ und $r_2 = (\text{„Hamburg“}, \text{„22309“}, \text{„Caesar-Klein-Ring“})$, der entsprechende Vektor ist:

$$v_\delta(q, r_2) = \begin{pmatrix} 7 \\ 5 \\ 16 \end{pmatrix}$$

Beim zweiten Verfahren werden die Datensatzpaare der Klasse -1 nicht zufällig erzeugt. Stattdessen werden die per OCR gelesenen Datensätze mit einem möglichst ähnlichen Datensatz der Datenbank kombiniert, der jedoch nicht der eigentlich korrekte Datensatz ist. Die Idee dabei ist, dass das Abstandsmaß für Datensätze bei der Sortierung der Ergebniskandidaten verwendet wird und daher vor allem eine gute Abgrenzung gegenüber weiteren ähnlichen Datensätzen liefern sollte. Ein Datensatzpaar bei diesem Verfahren ist beispielsweise (q, r_3) mit $q = (\text{„Bedin“}, \text{„10963“}, \text{„Mäckemstr.“})$ und $r_3 = (\text{„Berlin“}, \text{„10965“}, \text{„Möckernstr.“})$, der entsprechende Vektor ist:

$$v_\delta(q, r_1) = \begin{pmatrix} 2 \\ 1 \\ 2 \end{pmatrix}$$

Für die Wahl, welche der beiden Möglichkeiten für die Repräsentanten der Klasse -1 zum Einsatz kommen, sind folgende Überlegungen relevant. Die ersten Methode erzeugt in jedem Fall Beispiele von Datensätzen, die nicht zusammen passen. Durch die zufällige Auswahl des jeweils zweiten Datensatzes, sind die Klassen jedoch recht weit voneinander entfernt.[10] Die Klassifikationsfunktion der Support Vector Machine soll bei der Adresskorrektur verwendet werden, um die Ergebniskandidaten zu sortieren. Die Ergebniskandidaten sind dabei in der Regel alle schon relativ ähnlich zum Anfragedatensatz. Die auf der Klassifikationsfunktion basierende Ähnlichkeitsfunktion sollte daher vor allem zwischen mehreren recht ähnlichen Datensätzen eine gute Sortierung gewährleisten. Ich habe mich daher dazu entschieden die beiden Verfahren zu kombinieren, so dass in der Trainingsmenge als Repräsentanten der Klasse -1 einerseits zufällig zusammengestellte Datensatzpaare sind; andererseits sind auch solche Datensatzpaare enthalten, bei denen der zweite Datensatz dem ersten recht ähnlich ist, er jedoch nicht die zugehörige korrekte Adresse darstellt.

Vor dem Training der Support Vector Machine sollten die Werte der einzelnen Attribute auf das Intervall $[0, 1]$ normiert werden, damit kein Attribut auf Grund eines größeren absoluten Wertes mit einer höheren Gewichtung eingeht [HCL03]. Als Komponenten der Vektoren werden hier die aus der gewichteten Editierdistanz gewonnenen Wahrscheinlichkeiten verwendet, daher sind die Werte schon auf das Intervall $[0, 1]$ skaliert.

Bestimmung der Parameter

In diesem Abschnitt wird die Bestimmung von guten Parametern für die Support Vector Machine beschrieben. Es existieren verschiedene Implementierungen von Support Vector Machines, ich verwende hier das Programm LIBSVM, das unter anderem auch eine Programmierschnittstelle für die Sprache C++ anbietet. Als Kernel der Support Vector Machine verwende ich den in [HCL03] empfohlenen Kernel *Radial Basis Function (RBF)*. Dieser nimmt implizit eine Abbildung der Daten in eine höhere Dimension vor und kann somit den Fall von nicht linear trennbaren Daten besser behandeln. Diese Support Vector Machine hat zwei Parameter, die festgelegt werden müssen:

[10] Bei der verwendeten Trainingsmenge ließen sie sich linear trennen.

$C \downarrow \gamma \rightarrow$	0,001	0,003	0,01	0,03	0,1	0,3	1,0	3,0	10,0
0,1	69,67 %	69,80 %	93,12 %	93,16 %	93,16 %	93,13 %	93,12 %	93,12 %	93,10 %
1	93,12 %	93,16 %	93,16 %	93,16 %	93,16 %	93,16 %	93,16 %	93,16 %	93,16 %
10	93,16 %	93,16 %	93,16 %	93,16 %	93,16 %	93,16 %	93,16 %	93,15 %	93,27 %
100	93,16 %	93,16 %	93,16 %	93,16 %	93,16 %	93,16 %	93,13 %	93,37 %	93,63 %
1.000	93,16 %	93,16 %	93,16 %	93,16 %	93,16 %	93,13 %	93,39 %	**93,96 %**	93,61 %
10.000	93,16 %	93,16 %	93,16 %	93,16 %	93,12 %	93,34 %	93,74 %	93,91 %	93,84 %

Tabelle 7.1.: Treffsicherheit der Support Vector Machine für verschiedene Parameterkombinationen

- C bestimmt die Gewichtung der Kosten für falsche Klassifizierungen und
- γ ist der Faktor im Exponenten des RBF-Kernels.

Die Parameter werden mit einer Gittersuche (engl.: grid search) bestimmt [HCL03]. Bei einer Gittersuche werden verschiedene Kombinationen von Parametern nacheinander untersucht und jeweils die Qualität der resultierenden Funktion untersucht. Die Qualität wird hier mit einer Kreuzvalidierung (engl.: cross validation) als Anteil der korrekt klassifizierten Vektoren (Treffsicherheit, engl.: accuracy) bestimmt. Bei der Kreuzvalidierung wird die Trainingsmenge nacheinander jeweils in zwei Teilmengen zerlegt, das Model mit Hilfe der einen Menge trainiert und die Treffsicherheit bezüglich der anderen Teilmenge bestimmt. Ziel der Kreuzvalidierung ist es, eine Überanpassung der Klassifikationsfunktion an die Trainingsmenge zu vermeiden, so dass sie auch bisher unbekannte Testmenge gut klassifizieren kann.

Der Parameter C durchläuft bei der Gittersuche Werte der Menge {0,1, 1, 10, 100, 1.000, 10.000}, für den Parameter γ wähle ich Werte der Menge {0,001, 0,003, 0,01, 0,03, 0,1, 0,3, 1,0, 3,0, 10,0}. In Tabelle 7.1 ist jeweils die durchschnittliche Treffsicherheit der Kreuzvalidierung für alle Kombinationen der beiden Parameter dargestellt.

Viele Kombinationen von Parameter haben die gleiche Treffsicherheit, was wahrscheinlich darin begründet ist, dass nur relativ wenige Trainingsdaten (ungefähr 8.000 Vektoren) verwendet wurden und die Vektoren ohnehin recht gut in zwei Klassen separierbar sind. Die Parameterkombination $C = 1.000, \gamma = 3{,}0$ erreicht mit knapp 94,0 % insgesamt die beste Treffgenauigkeit und wird daher für die Klassifikationsfunktion verwendet. Das so generierte Modell (die Klassifikationsfunktion) wird abgespeichert und dann für die Sortierung der Ergebniskandidaten verwendet.

Verwendung der Klassifikationsfunktion zur Ähnlichkeitsbestimmung

Bei der Sortierung der Datensätzen nach Ähnlichkeit (Abschnitt 6.2.3) wird die so trainierte Klassifikationsfunktion verwendet, um die Ähnlichkeit der Ergebniskandidaten zum Anfragedatensatz zu bestimmen. In Abschnitt 4.3.2 wurde bereits gezeigt, wie ausgehend vom Funktionswert der Entscheidungsfunktion f der Abstand von zwei Datensätzen bestimmt werden kann:

$$\delta_{\text{svm}}(q, r) = f\left(v_\delta\left(q, r\right)\right) + z$$

7.2.3. Zusammenfassung

In den vorangehenden Abschnitten wurde erläutert, wie die Ähnlichkeitsfunktion für Datensätze trainiert werden kann. Sie basiert auf der Klassifikationsfunktion eines binären Klassifikators, der Support Vector Machine. Die Support Vector Machine optimiert die Entscheidungsfunktion mit Hilfe von Trainingsdaten, für die die Klasse jeweils bekannt ist. Dabei versucht sie, den Abstand zwischen der trennenden Hyperebene und den Vektoren zu maximieren. Sie kann dabei auch die Fälle behandeln, in denen die Daten nicht linear trennbar sind, und verwendet dazu Schlupfvariablen und eine Abbildung der Daten in eine höhere Dimension. Eine Support Vector Machine wurde mit Trainingsdaten trainiert und die Parameter dabei mit Hilfe einer Gittersuche und anschließender Kreuzvalidierung optimiert. Die beim Training erlernte Klassifizierungsfunktion wird abgespeichert und kann bei der Adresskorrektur zur Sortierung der Ergebniskandidaten verwendet werden.

7.3. Weitere Möglichkeiten

In diesem Abschnitt werden kurz weitere Möglichkeiten beziehungsweise Modifikationen der Lernverfahren angeführt, die jedoch nicht oder noch nicht implementiert sind.

Die Wahrscheinlichkeit, dass t die korrekten Zeichenkette zu einer gelesenen Zeichenkette s ist, basiert auf der Wahrscheinlichkeit, dass die Zeichenkette s beim OCR-Vorgang als t erkannt wurde. Hier wären noch weitere Faktoren denkbar. Unter anderem könnte ein von der OCR-Software zur Verfügung gestellter Konfidenzwert ausgenutzt werden, der für die Zeichenkette (oder sogar jeden Buchstaben) angibt, mit welcher Sicherheit er korrekt gelesen wurde.

Ein weiteres Lernverfahren könnte auf Basis der gelesenen Kombinationen von Postleitzahlen und zugehörigen Orten die Liste der Synonyme für Orte erweitern. Somit wäre diese Liste nicht mehr statisch vorgegeben, sondern würde im laufenden Betrieb erweitert werden.

7.4. Zusammenfassung

In diesem Kapitel wurde untersucht, wie die Ähnlichkeitsmaße für Zeichenketten und Datensätze erlernt werden können. Abschließend können daher Antworten auf zwei weitere der in der Einleitung (Abschnitt 1.4) gestellten Fragen gegeben werden:

5. *Wie kann die Ähnlichkeitsfunktion für Zeichenketten mit Hilfe von Trainingsdaten verbessert werden?*

   Die Gewichte der Editierdistanz können mit Hilfe von Trainingsdaten erlernt werden. Das Ergebnis der Editierdistanz repräsentiert dann die Wahrscheinlichkeit, dass die eine Zeichenkette beim OCR-Prozess in die andere überführt wurde.

6. *Wie kann die Ähnlichkeitsfunktion für Datensätze mit Hilfe von Trainingsdaten verbessert werden?*

   Das Ähnlichkeitsmaß für Datensätze basiert auf der Klassifikationsfunktion einer Support Vector Machine. Diese Funktion kann mit Hilfe von Trainingsdaten so optimiert werden, dass sie eine möglichst gute Trennung der Klassen sicherstellt. Damit kann auch das Ähnlichkeitsmaß verbessert werden.

Die Auswirkungen dieser beiden Lernverfahren auf die Qualität der Ergebnisse werden in Abschnitt 8.5.1 experimentell untersucht.

8. Evaluation

Dieses Kapitel beschreibt die Ergebnisse der Evaluation der entwickelten Indexstruktur für Datensätze und die Ergebnisse des Systems zur Adresskorrektur. Dabei werden zunächst die Eckdaten der verwendeten Datenbank erläutert und die Erzeugung der Test- und Trainingsdaten beschrieben. Anschließend werden die in der Testmenge auftretenden Fehler analysiert und beispielsweise die Häufigkeiten der Editieroperationen angegeben.

Der Fri wird experimentell mit verschiedenen Datenmengen untersucht und es werden geeignete Werte für die Parameter bestimmt. Er wird mit dem einfachen Ansatz zur Indizierung von Datensätzen mit Hilfe eines Tries bezüglich der Effizienz der Anfragebeantwortung verglichen. Abschließend wird die Qualität der Ergebnisse der Adresskorrektur analysiert und der Einfluss der Heuristiken und Lernverfahren untersucht.

8.1. Datenbank

Dieser Abschnitt führt einige Eckdaten der verwendeten Datenbank auf. Die Erzeugung der Datensätze aus der Datenbasis wurde in Abschnitt 6.1 beschrieben. Insgesamt gibt es 1.210.918 Datensätze für Straßen, 8.079 für Großempfänger und 17.223 für Postfachbereiche. Dies ergibt in der Summe 1.236.220 Datensätze.

In diesem Absatz wird die Anzahl der jeweils verschiedenen Werte für ein Attribut angegeben. Es gibt 14.236 verschiedene Ortsnamen. Von insgesamt 100.000 möglichen fünfstelligen Postleitzahlen sind 27.861 vergeben; davon werden 8.234 Postleitzahlen für Straßen, 2.404 für Großempfänger und 17.223 für Postfächer verwendet. Es gibt 419.729 verschiedene Werte für das Attribut *str*, die sich aus 411.649 Straßennamen, 8.079 Namen von Großempfängern und dem Begriff „Postfach“ zusammensetzen. Diese Daten sind in Tabelle 8.1 zusammengefasst. Zusätzlich wurden insgesamt 16.667 Synonyme für Ortsnamen und 14.756 Synonyme für Straßennamen erzeugt.

Wie im letzten Absatz erwähnt, gibt es in der Datenbank insgesamt ungefähr 1,2 Mio Datensätze und zusätzlich Listen von Synonymen für die Attribute *ort* und *str*. Für die Evaluation der Effizienz der Indexstrukturen habe ich aus dieser Datenbank zusätzlich kleinere Mengen zufällig ausgewählter Adressdatensätze erstellt. Sie werden im Folgenden mit $D_{5.000}$, $D_{10.000}$, $D_{50.000}$, $D_{100.000}$ und $D_{500.000}$ bezeichnet, wobei der Index die Anzahl der enthaltenen Datensätze angibt. Die komplette Datenbank heißt damit $D_{1{,}2\,\text{Mio}}$.

ort	14.236	
plz	27.861	= 8.234 für Straßen + 2.404 für Großempfänger + 17.223 für Postfächer
str	419.729	= 411.649 Straßennamen + 8.079 Großempfänger + „Postfach“

Tabelle 8.1.: Anzahl der verschiedenen Attributwerte für die Attribute *ort*, *plz* und *str*

8.2. Trainings- und Testdaten

Für die Lernverfahren (Kapitel 7) und die Evaluation werden Trainings- beziehungsweise Testdaten benötigt. Die Testmengen werden hier mit T bezeichnet und enthalten Paare (q, r) von Datensätzen, wobei q die mittels OCR gelesene Adresse und r die dazugehörige korrekte Adresse ist. Mit diesen Daten können die Gewichte der Editierdistanz und die Ähnlichkeitsfunktion für Datensätze erlernt werden. Außerdem kann mit Hilfe solcher Daten die Qualität des vorgeschlagenen Verfahrens zur Adresskorrektur quantifiziert werden.

Zum Testen habe ich zunächst 1.000 Testbriefe mit Adressen bedruckt und die zugehörigen Adressen abgespeichert. Die Briefe habe ich dann durch die Briefsortiermaschine fahren lassen und dabei jeweils die per OCR erkannte Adresse mitprotokolliert. Die so gewonnenen Daten konnte ich während der Entwicklungsphase zum Testen verwenden.

Für die eigentliche Evaluation habe ich erneut Testmengen erzeugt, um eine größere Anzahl von Datensätzen und damit eine größere Aussagekraft der gewonnenen Ergebnisse zu erhalten. Ich habe dabei sowohl nur für diesen Zweck hergestellte Testbriefe (*Testpost*),[1] als auch echte Briefe aus dem laufenden Tagesgeschäft verwendet (*Echtpost*). Bei der Erzeugung der Testmengen bin ich folgendermaßen vorgegangen:

1. Durchfahren der Briefe durch die Sortiermaschine.
2. Export der jeweils per OCR gelesenen Zeichenkette und des Bildes des Briefs.[2]
3. Manuelle Zuordnung der korrekten Adresse zu jeder gelesenen Zeichenkette unter Zuhilfenahme des exportierten Bildes.

Zur Unterstützung der manuellen Zuordnung der jeweils korrekten Adresse zu einem Brief habe ich ein Hilfsprogramm entwickelt, in dem jeweils die erkannten Zeichenketten und exportierten Bilder nebeneinander dargestellt werden (siehe Abbildung 8.1).

Insgesamt habe ich auf diese Weise 4.298 Testbriefe und 1.181 echte Briefe eingelesen. Bei 340 Testbriefen und 72 Echtbriefen wurde bei der optischen Zeichenerkennung kein Text gelesen, beispielsweise weil Briefe verkehrt herum eingelegt wurden und sich die Adresse somit auf der von der Kamera abgewandten Seite befand. Ein mittlerweile behobener Fehler in der OCR-Software führte dazu, dass bei 132 Testbriefen und 26 echten Briefen die Zeichenketten nicht korrekt exportiert wurden. Die Datensätze dieser Briefe wurden nicht weiter verwendet. Bei der Zuordnung der korrekten Adressen für die Briefe stellte sich weiterhin heraus, dass die Adressen auf einigen der Testbriefe anonymisiert worden waren. Bei diesen Adressen wurden offensichtlich Ort und Postleitzahl zufällig kombiniert. Diese Datensätze mussten manuell identifiziert und von der weiteren Verwendung ausgeschlossen werden; es handelte sich um 1.040 Datensätze. Bei den echten Briefen waren 14 an Adressen ins Ausland adressiert und konnten bei der Adresskorrektur daher ignoriert werden. Insgesamt konnten die Adressen von 10 Testbriefen und von 11 Echtbriefen auch bei manueller Untersuchung keiner Adresse der Datenbank zugeordnet werden. Dies lag beispielsweise daran, dass eine Straße angegeben war, die nicht in der Datenbank verzeichnet ist.[3] Solche Datensätze werden hier mit *Adresse*

[1] Die Testbriefe umfassen die oben genannten 1.000 Briefe und weitere knapp 3.000 schon für andere Testzwecke von PostCon bedruckte Briefe.

[2] Das ist das von der Zeilenkamera aufgenommene und zur optischen Zeichenerkennung verwendete Foto.

[3] Beispielsweise „Straße der Jugend 42, 37412 Herzberg". Die Postleitzahl ist zwar diesem Ort zugeordnet, es gibt dort jedoch keine Straße mit diesem Namen. In einem anderen Ort namens „Herzberg" und der Postleitzahl „04916" gibt es eine solche Straße. Es ist jedoch nicht mit Sicherheit entscheidbar, wohin dieser Brief eigentlich gesendet werden sollte. Eine Auflistung dieser nicht zugeordneten Adressen findet sich im Anhang A.4

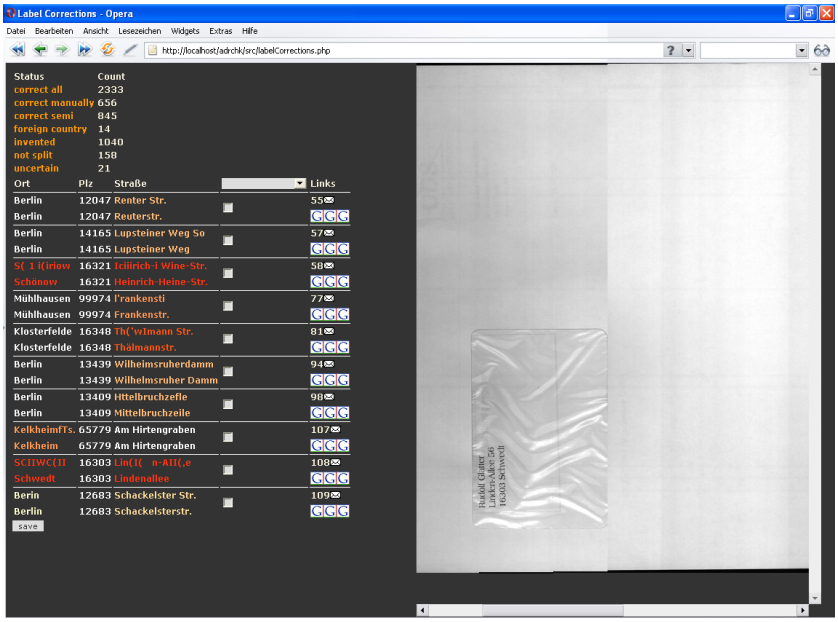

Abbildung 8.1.: Hilfsprogramm zur manuellen Zuordnung der korrekten Adressen

	Testpost	Echtpost
Insgesamt	4.298	1.181
Kein Text gelesen	- 340	- 72
Bug der OCR-Software	- 132	- 26
Anonymisiert	- 1.040	- 0
Ausland	- 0	- 14
Adresse unbekannt	- 10	- 11
Verbleibend	2.776	1.058

Tabelle 8.2.: Verbleibende Testbriefe

unbekannt bezeichnet. In Tabelle 8.2 sind zusammenfassend die von der weiteren Verwendung ausgeschlossenen Datensätze aufgeführt.

Von den insgesamt 4.298 Testbriefen beziehungsweise 1.181 echten eingelesenen Briefen bleiben somit noch 2.776 beziehungsweise 1.058 übrig. Diese Datenmengen werden im Folgenden T_{test} beziehungsweise T_{echt} genannt. Die Vereinigung $T_{\text{test}} \cup T_{\text{echt}}$ wird mit T_{alle} bezeichnet und umfasst 3.834 Datensatzpaare.

Tabelle 8.3 stellt dar, wie viele der Datensätze jeweils auf normale Straßen, Großempfänger und Postfächer entfallen.

In Tabelle 8.4 sind weitere Merkmale der Testdatenmengen angegeben. Beispielsweise wurde bei insgesamt 13 Briefen die Adresse komplett in Versalien (Großbuchstaben) geschrieben. Bei 439 Briefen wurde ein Synonym anstelle des eigentlichen Ortsnamens verwendet. Weil der eigentlich korrekte Ortsname auch den Ortszusatz (z. B. „bei Berlin“) enthält, sind darunter auch viele Synonyme der Art „Schönefeld“ → „Schönefeld bei Berlin“. Bei 940 Straßennamen wurde ein Synonym verwendet. Dies ist beinhaltet jedoch hauptsächlich

	T_{test}	T_{echt}	T_{alle}
Straßen	2.753	967	3.720
Postfächer	20	57	77
Großempfänger	3	34	37
Summe	2.776	1.058	3.834

Tabelle 8.3.: Straßen, Postfächer und Großempfänger

	T_{test}	T_{echt}	T_{alle}
Großschreibungen	3	10	13
Synonyme für Ortsnamen	178	261	439
Synonyme für Straßennamen	784	156	940

Tabelle 8.4.: Besonderheiten in den Testdaten

alternative Schreibweisen für das Wort „Straße“ (beispielsweise „Berliner Str.“ → „Berliner Straße“). Diese besonderen Merkmale sind offensichtlich unterschiedlich auf die Testbriefe und die echten Briefen verteilt. Bei der Evaluation werden diese beiden Mengen daher auch jeweils separat aufgeführt.

Um die FRI-Indexstruktur mit dem einfachen Ansatz zur Indizierung von Datensätzen vergleichen zu können, wird eine Testmenge benötigt, in der beispielsweise keine Großempfänger enthalten, aber auch keine Synonyme verwendet werden. Diese Menge ist eine Teilmenge von T_{alle}, wird mit T_{einfach} bezeichnet und umfasst 3.512 Datensätze.

8.3. Fehler

In diesem Abschnitt werden Ergebnisse der Untersuchung der Testmengen auf Rechtschreib-, Tipp- und OCR-Fehler dargestellt. Dabei werden vor allem die Auswirkungen der Fehler auf Zeichenkettenebene und nicht ihre Ursachen betrachtet. Unter anderem wird die Verteilung der Fehler auf die drei Attribute *ort*, *plz* und *str* angegeben und die Häufigkeiten der verschiedenen Editieroperationen analysiert.

In Tabelle 8.5 ist für die drei Testmengen T_{test}, T_{echt} und T_{alle} aufgeschlüsselt, wie häufig die Adresse komplett richtig war. Dabei bedeutet *komplett richtig*, dass alle drei Attribute exakt mit den eigentlich manuell ermittelten, korrekten Werten übereinstimmten. Insbesondere sind also keine OCR-Fehler, Rechtschreib- und Tippfehler aufgetreten. Insgesamt sind also gut 60 % der Datensätze komplett richtig und müssen nicht korrigiert werden. Die Verteilung war bei den Testbriefen und bei der Echtpost ungefähr gleich.

	T_{test}	2.776	T_{echt}	1.058	T_{alle}	3.834
Komplett richtig	*61,7 %*	1.713	*60,7 %*	642	*61,4 %*	2.355
Mindestens eine Abweichung	*38,3 %*	1.063	*39,3 %*	416	*38,6 %*	1.479

Tabelle 8.5.: Fehler aufgeschlüsselt nach Testmenge

In Tabelle 8.6 sind diese Werte für die einzelnen Attribute getrennt dargestellt. Dabei bedeutet *komplett richtig*, dass der Wert dieses Attributes mit dem eigentlich korrekten Wert exakt übereinstimmte. Dabei ist zu sehen, dass die Postleitzahl nur sehr selten falsch

erkannt beziehungsweise falsch auf den Brief geschrieben wurde (nur in 2,5 % der Fälle). Dies liegt unter anderem daran, dass die optische Zeichenerkennung der Postleitzahl auf Ziffern beschränkt wird. Die Straße hingegen stimmte relativ häufig nicht mit dem eigentlich korrekten Wert überein (bei 29,7 % der Fälle). Dies liegt daran, dass bei der Straße auch Rechtschreibfehler auftreten können. Außerdem sind die Zeichenketten deutlich länger als die Postleitzahl und daher mit einer höheren Wahrscheinlichkeit von Fehlern betroffen.

	ort		*plz*		*str*	
Komplett richtig	*84,8 %*	3.241	*97,4 %*	3.738	*70,3 %*	2.694
Mindestens eine Abweichung	*15,2 %*	593	*2,5 %*	96	*29,7 %*	1.140
Anzahl Editieroperationen		1.322		135		3.533

Tabelle 8.6.: Fehler in T_{alle} aufgeschlüsselt nach Attributen

In der letzten Zeile der Tabelle 8.6 ist die Anzahl der beobachteten Editieroperationen dargestellt. Die Zahl in der zweiten Spalte bedeutet beispielsweise, dass beim Attribut *ort* bei allen Testdaten zusammen 1.322 Editieroperationen nötig waren, um den erkannten Ortsnamen in den eigentlich korrekten Ortsnamen zu überführen.[4] Jede Abweichung beim Attribut *ort* umfasste also durchschnittlich $\frac{1.322}{593} \approx 2{,}2$ Editieroperationen. Eine Ursache dafür sind Fehler der optischen Zeichenerkennung, die sich auf mehrere benachbarte Buchstaben beziehen (engl.: burst errors). Dieses Phänomen konnte ich bei der manuellen Korrektur der Datensätze häufig beobachten.

In Abbildung 8.2 ist dargestellt, wie viele Datensätze es jeweils zu einer bestimmten Distanz gab. Dabei ist die Distanz von Datensätzen hier einfach als Summe der attributweisen einfachen Editierdistanz gewählt. Es ist zu sehen, dass nur ein kleiner Teil der Datensätze eine große Distanz hatte. Ein Beispiel für eine besonders stark abweichende Adresse ist „*15831 (li*tlins(lort* / F)orfstral,~(, 2 1 C*“, wobei die zugehörige korrekte Adresse folgende ist: „*15831 Jühnsdorf / Dorfstr.*“.

In Tabelle 8.7 sind für die verschiedenen Typen von Editieroperationen einerseits die Anzahl der Editieroperationen dieses Typs insgesamt und andererseits die Anzahl der verschiedenen Editieroperationen aufgeschlüsselt. Die Anzahl der verschiedenen Operationen war dabei für jeden Typ deutlich geringer; einige der Operationen traten also besonders häufig auf. Zufälligerweise gab es gleich viele Verschmelzungen wie Aufspaltungen von Buchstaben.

	Anzahl	Verschiedene
Substitution	349	116
Löschung	406	30
Einfügung	380	66
Verschmelzung	243	28
Aufspaltung	243	50

Tabelle 8.7.: Editieroperationen aufgeschlüsselt nach Typ

In Tabelle 8.8 sind beispielhaft jeweils die zehn am häufigsten beobachteten Verschmelzungen und Aufspaltungen aufgelistet. Dabei ist deutlich ein Zusammenhang zwischen der optischen Ähnlichkeit der Buchstaben und der beobachteten Häufigkeit zu erkennen. Die Verschmelzungen waren in den meisten Fällen entstanden, weil die Buchstaben dicht beieinander gedruckt waren, so dass die optische Zeichenerkennung sie als *einen* Buchstabe

[4] Die im Folgenden angegebenen Werte zur Editierdistanz beziehen sich auf die einfache Editierdistanz.

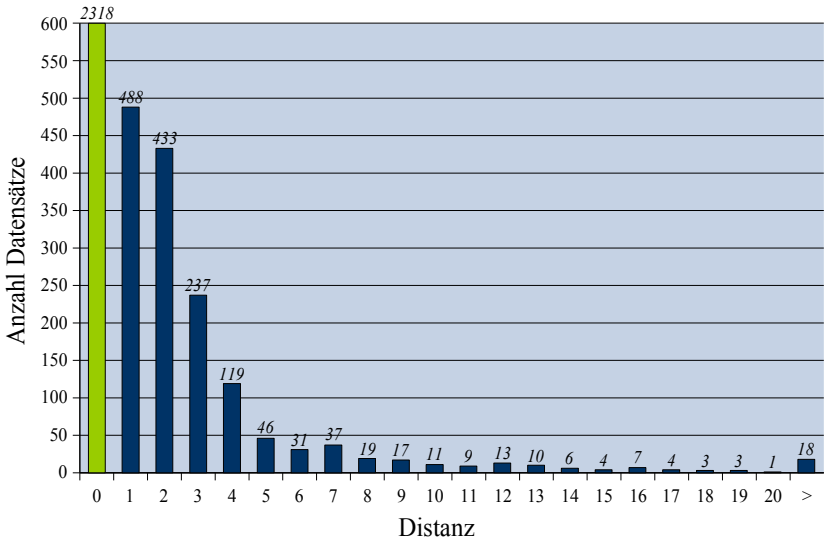

Abbildung 8.2.: Fehler in T_{alle}, aufgeschlüsselt nach Distanz

betrachtet hat. Die Aufspaltungen waren vor allem bei Briefen mit einem Sichtfenster und einer speziellen Schriftart mit relativ dünnen Linien aufgetreten. Es ist zu erkennen, dass bei diesen zehn häufigsten Editieroperationen kaum Großbuchstaben betroffen waren, weil diese einerseits nur selten von Tipp- und Rechtschreibfehlern betroffen sind [Kuk92] und andererseits bei der OCR in den meisten Fällen korrekt identifiziert wurden. Die einzigen hier auftretenden häufigen Operationen, die Großbuchstaben enthalten, waren „ S“ → „s“, „-S“ → „s“ und „s“ → „ S“. Diese stammen offensichtlich von alternativen Schreibweisen für das Wort „Straße“ (zusammen oder auseinander geschrieben oder mit Bindestrich). Die vollständige Tabelle der Anzahl der beobachteten erweiterten Editieroperationen inklusive der daraus berechneten Wahrscheinlichkeit für das Auftreten der Operation findet sich im Anhang A.5.

In Tabelle 8.9 sind die erlernten Wahrscheinlichkeiten der Ersetzungsoperationen aufgeführt. Die Tabelle ist auf das Attribut *plz* und auf die Ziffern von 0 bis 9 beschränkt, damit die Darstellung auf einer Seite möglich ist.[5] Es ist zu sehen, dass die Ziffern der Postleitzahl nur in sehr wenigen Fällen verändert wurden. Dies liegt unter anderem daran, dass die Testmenge nicht besonders groß war, aber auch daran, dass die Postleitzahl in den meisten Fällen von der OCR korrekt erkannt wurde. Bei der Postleitzahl konnten außerdem auch nur sehr wenige Verschmelzungen und Aufspaltungen beobachtet werden.

8.4. Indexstruktur für Datensätze

In diesem Abschnitt Ergebnisse der experimentellen Untersuchung der Indexstruktur für Datensätze, dem FRI, präsentiert. Zunächst wurde der Einfluss des Parameters *minSizeTrie*

[5] Insgesamt hat die Tabelle k Spalten und k Zeilen mit $k = 2 \cdot 26$ Buchstaben + 10 Ziffern + Umlaute + Sonderzeichen.

[6] Wie in Abschnitt 7.1 erwähnt, wird auch jeder nicht in den Trainingsdaten beobachteten Editieroperation ein Wert von $\alpha = 0{,}00001$. In der Darstellung wird hier stattdessen 0,0001 verwendet.

Operation	Anzahl
„rl“ → „d“	70
„rn“ → „m“	37
„li“ → „h“	35
„ S“ → „s“	25
„li“ → „n“	7
„li“ → „ä“	7
„rf“ → „d“	7
„li“ → „ü“	5
„ri“ → „d“	4
„-S“ → „s“	4
„u“ → „ti“	20
„n“ → „ii“	19
„n“ → „ri“	17
„s“ → „ S“	15
„h“ → „li“	14
„r“ → „i-“	9
„u“ → „ii“	9
„ß“ → „ss“	8
„d“ → „(l“	8
„d“ → „(I“	7

Tabelle 8.8.: Erweiterte Editieroperationen

	→ „0“	→ „1“	→ „2“	→ „3“	→ „4“	→ „5“	→ „6“	→ „7“	→ „8“	→ „9“
„0“ →	0.9936	0.0028	0.0005	0.0001	0.0001	0.0001	0.0001	0.0001	0.0001	0.0011
„1“ →	0.0001	0.9911	0.0010	0.0003	0.0006	0.0001	0.0001	0.0040	0.0001	0.0001
„2“ →	0.0001	0.0001	0.9980	0.0001	0.0001	0.0001	0.0001	0.0001	0.0001	0.0001
„3“ →	0.0001	0.0001	0.0009	0.9953	0.0001	0.0001	0.0001	0.0001	0.0001	0.0001
„4“ →	0.0001	0.0011	0.0001	0.0001	0.9965	0.0001	0.0001	0.0001	0.0001	0.0001
„5“ →	0.0001	0.0007	0.0001	0.0015	0.0001	0.9924	0.0001	0.0007	0.0015	0.0007
„6“ →	0.0001	0.0001	0.0001	0.0001	0.0001	0.0001	0.9961	0.0001	0.0012	0.0001
„7“ →	0.0001	0.0008	0.0001	0.0001	0.0001	0.0001	0.0008	0.9948	0.0001	0.0001
„8“ →	0.0001	0.0019	0.0001	0.0076	0.0019	0.0001	0.0001	0.0001	0.9866	0.0019
„9“ →	0.0001	0.0023	0.0001	0.0001	0.0001	0.0011	0.0011	0.0001	0.0001	0.9941

Tabelle 8.9.: Einfache Editieroperationen beim Attribut plz[6]

und der Berechnung der Priorität der Knoten analysiert. Anschließend wurde der FRI mit dem einfachen Trie-Index zur Suche von ähnlichen Datensätzen verglichen. Außerdem wurde die Effizienz vom FRI beim Einsatz für die Adresskorrektur untersucht. Im folgenden Abschnitt wird jedoch zunächst kurz die Durchführung der Experimente beschrieben.

8.4.1. Durchführung

Dieser Absatz enthält die technischen Rahmenbedingungen der experimentellen Untersuchung. Sowohl die FRI-Datenstruktur als auch das Adresskorrektursystem wurden objektorientiert in der Programmiersprache C++ implementiert. Bei der Implementierung kam die Software-Bibliothek BOOST [Boo08] zum Einsatz. Als Trie wurde die Implementierung eines Ternary Search Tries verwendet [Ekm07], als Support Vector Machine kam die LIBSVM [CL01] zum Einsatz. Kompiliert wurde das Programm mit dem GNU C COMPILER (GCC).

Das Modul für die Adresskorrektur wird als dynamische Bibliothek (engl.: Dynamic Link Library, DLL) in die OCR-Software eingebunden. Alle hier angegebenen Tests fanden jedoch nicht auf dem Steuerrechner der Briefsortiermaschine, sondern lokal auf einem Laptop statt. Dieser Laptop war mit einem INTEL CORE DUO Prozessor (Taktfrequenz 1,83 GHz) sowie einem Arbeitsspeicher von 2 Gigabyte ausgerüstet.

Die Laufzeit wurde jeweils mit Hilfe einer Zeitnehmer-Klasse (engl.: timer class) gemessen [Wil05]. Der Speicherbedarf der Datenstrukturen wurde jeweils mit Hilfe von WINDOWS MANAGEMENT INSTRUMENTATION (WMIC) [Mic08] ermittelt.

Für die folgenden experimentellen Untersuchungen kamen jeweils entweder der einfache Index für Datensätze[7] oder der von mir vorgeschlagene FRI-Index zum Einsatz. Der einfache Index wird mit I_{Trie} bezeichnet, der FRI mit I_{FRI}.

Als Datenbanken wurden die in Abschnitt 8.1 genannten Mengen $D_{5.000}$, $D_{10.000}$, $D_{50.000}$, $D_{100.000}$, $D_{500.000}$ und $D_{1,2\,\text{Mio}}$ verwendet. Die Trainingsmengen waren die in Abschnitt 8.2 erwähnten Mengen T_{test}, T_{echt}, T_{alle} und T_{einfach}. Bei jedem der folgenden Abschnitte werden die verwendete Indexstruktur, Datenbank und Testmenge einleitend in einer Tabelle angegeben.

Der FRI wurde dabei zunächst ohne Synonyme und ohne die für die Adresskorrektur vorgenommenen Anpassungen verwendet, um ihn mit der einfachen Trie-Indexstruktur vergleichen zu können (diese unterstützt beispielsweise keine Synonyme). Lediglich die unterschiedliche Gewichtung der Attribute ($g_{ort} = 1$, $g_{plz} = 2$, $g_{str} = 1$) wurde beibehalten. Um diese Gewichtung auch beim einfachen Index zu realisieren, könnte die Postleitzahl beispielsweise zweimal hintereinander geschrieben werden, so dass eine Abweichung der Postleitzahl gleichfalls mit Kosten von 2 verbunden wäre. Dies wurde hier jedoch nicht gemacht, um den einfachen Index gegenüber dem FRI nicht durch den zusätzlichen Speicheraufwand zu benachteiligen. Dadurch liefern beide Indexstrukturen jedoch nicht in allen Fällen die gleichen Ergebnisse.

8.4.2. Einfluss der Parameter

Der FRI hat zwei Parameter, die den Speicherbedarf und die Laufzeit der Ähnlichkeitssuche beeinflussen. Dies ist zum einen der Parameter *minSizeTrie*, der vor allem eine Reduzierung des Speicherplatzes erreichen soll. Zum anderen ist dies der Parameter *prio*, der bestimmt, in welcher Reihenfolge die Knoten im Algorithmus ausgeführt werden, wenn sie den gleichen erwarteten Abstand haben. Die experimentellen Untersuchungen fanden auf den Datenbanken $D_{5.000}$, …, $D_{1,2\,\text{Mio}}$ mit der Testmenge T_{einfach} statt.

[7] Bei diesem wurden die Attributwerte konkateniert und in einem Trie indiziert; siehe Abschnitt 5.3.1.

Index	Datenbank	Testmenge
I_{FRI}	$D_{5.000}, \ldots, D_{1{,}2\,\text{Mio}}$	T_{einfach}

Einfluss des Parameters *minSizeTrie*

In diesem Abschnitt wird untersucht, wie der Speicherplatz und die Anfragezeit beim FRI vom Parameter *minSizeTrie* abhängt. Dieser Parameter gibt an, bis zu welcher Menge von Datensätzen die Werte in einem Index als Liste und nicht als Trie verwaltet werden (siehe Abschnitt 5.3.7). Es soll hier also ein Kompromiss gefunden werden zwischen der einfacheren Speicherung als Liste, die mit einer jeweils linearen Suchzeit einhergeht, und einer Speicherung als Trie, bei dem die Suchzeit sublinear in der Anzahl der enthaltenen Werte ist. Für *minSizeTrie* wurden die Werte 0 (die Indizes sind ausnahmslos Tries), 10, 100 und 200 getestet.

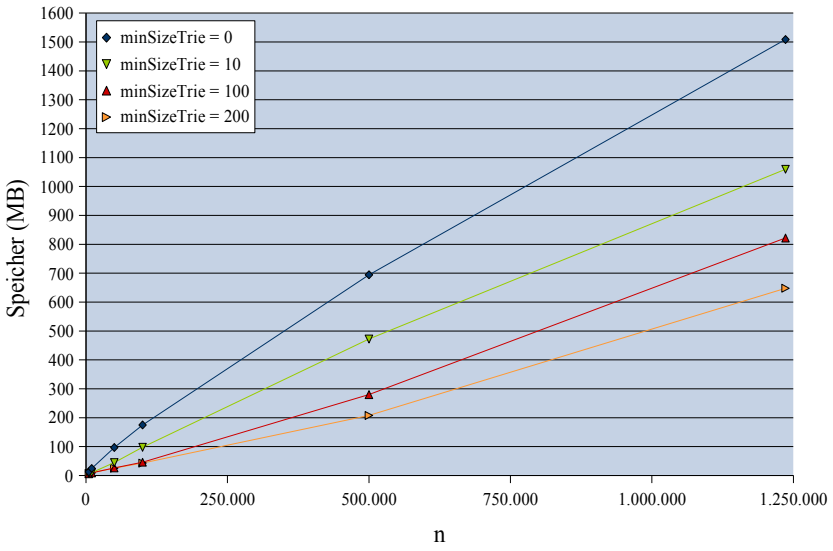

Abbildung 8.3.: Speicherbedarf in Abhängigkeit des Parameters *minSizeTrie*

In Abbildung 8.3 ist dargestellt, wie der Speicherplatz der FRI-Datenstruktur von Anzahl der Datensätze n und dem Paremeter *minSizeTrie* abhing. Dabei ist sehr deutlich eine lineare Abhängigkeit des Speicherbedarfs von der Anzahl der Datensätze zu erkennen. Dies deckt sich mit der theoretischen Analyse in Abschnitt 5.3.6. Dort wurde (allerdings für den ungünstigsten Fall) ein asymptotischer Speicherbedarf von $O(n \cdot m^2 \cdot (1 + S))$ ermittelt.[8]

In Abbildung 8.3 ist außerdem zu erkennen, dass der Speicherbedarf geringer war, je höher der Wert *minSizeTrie* gewählt wurde. Für $minSizeTrie = 200$ wurde weniger als die Hälfte des Speicherplatzes gegenüber dem Fall $minSizeTrie = 0$ benötigt (648 MB gegenüber 1.508 MB). Dies liegt daran, dass bei relativ wenigen Elementen die einfache Speicherung in einer Liste weniger Platz benötigt, als in einem Trie, bei dem die Verwaltung der Knoten relativ speicherintensiv ist. (Ab einer gewisser Anzahl an Werten kann durch die

[8] m ist die Anzahl der Attribute, S die maximale Anzahl von Synonymen für einen Attributwert. Hier gibt es $m = 3$ Attribute und $S = 0$ Synonyme.

gemeinsame Speicherung der Präfixe in einem Trie jedoch sogar eine Kompression erreicht werden. [SM96]).

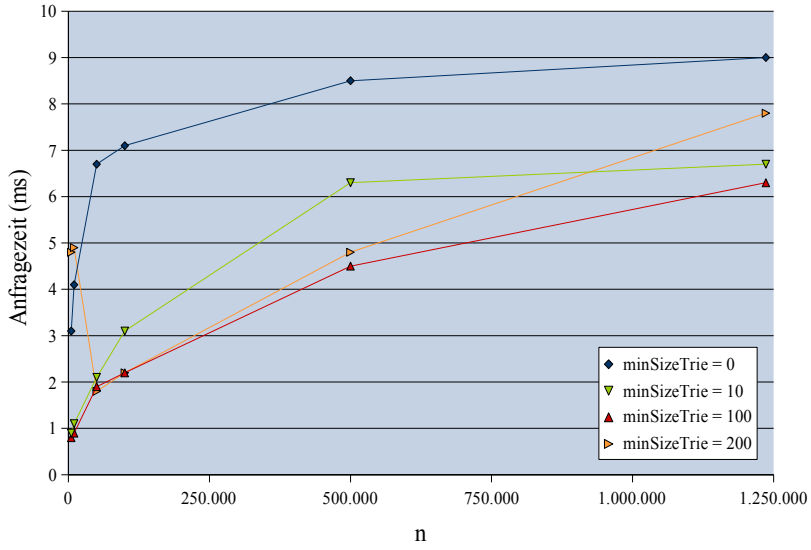

Abbildung 8.4.: Anfragezeit in Abhängigkeit des Parameters $minSizeTrie$

Für jeden der Datensätze in T_{einfach} wurde der ähnlichste Datensatz gesucht („Top-1-Anfrage“). Die durchschnittliche Anfragezeit der Datensätze ist in Abbildung 8.4 in Abhängigkeit der Datenbankgröße und des Parameters $minSizeTrie$ dargestellt.[9] Für $minSizeTrie = 0$ (die oberste Kurve) ist deutlich zu erkennen, dass die Suchlaufzeit sublinear mit der Anzahl der Datensätze steigt. Beispielsweise war bei $n = 500.000$ die durchschnittliche Anfragezeit 8,5 s, während sie bei $n = 1.236.220$ mit 9,0s nur unwesentlich höher war. Dieses sublineare Wachstum liegt unter anderem daran, dass die zu Grunde liegenden Tries ebenfalls ein sublineares Wachstum der Anfragezeit in Abhängigkeit der Anzahl der gespeicherten Zeichenketten haben.

Wenn für den Parameter $minSizeTrie$ Werte größer als 0 gewählt werden, sind einige der Indizes beim Fri eine Liste und kein Trie. Die Suchzeit bei diesen Indizes ist damit linear abhängig von der Anzahl der enthaltenen Werte. Mit höheren Werten von $minSizeTrie$ tendiert dadurch auch die Anfragezeit vom gesamten Fri mehr zu einem linearen Wachstum. Dies ist beispielsweise beim Vergleich der Kurven für $minSizeTrie = 0$ beziehungsweise $minSizeTrie = 100$ zu erkennen. Dieses Verfahren lohnt sich daher vor allem bei eher kleinen Datenbanken. Jedoch ergab sich auch bei einer Datenbankgröße von $n = 1.236.220$ noch eine merkliche Verringerung der durchschnittlichen Suchlaufzeit bei Wahl von $minSizeTrie = 100$ (6,3 ms) gegenüber $minSizeTrie = 0$ (9,0 ms).

Eine Anomalie war bei der Wahl von $minSizeTrie = 200$ zu beobachten: Die durchschnittliche Suchlaufzeit war für die kleinen Datenbanken ($n = 5.000$ beziehungsweise 10.000) höher als für größere Datenbanken. Dies liegt wahrscheinlich daran, dass in diesem Fall auch

[9] Die Berechnung der Priorität der Knoten erfolgt hier mit dem Verfahren, das sich im folgenden Abschnitt als bestes herausstellen wird.

die Indizes der ersten Kategorie als Liste verwaltet werden und somit bei jeder Suchanfrage komplett durchlaufen werden müssen. Dies führt zu einer deutlichen Erhöhung der Anfragezeit.

Einfluss der Berechnung der Knotenpriorität

In diesem Abschnitt wird der Einfluss der Berechnung der Knotenpriorität auf die Laufzeit der Ähnlichkeitssuche untersucht (siehe Abschnitt 5.3.5). Beim Suchalgorithmus werden die Knoten in einer Prioritätswarteschlange verwaltet und es wird immer der Knoten als nächstes ausgeführt, bei dem der geringste Abstand zum Anfragedatensatz erwartet wird. Wenn mehrere Knoten den gleichen mindestens erwarteten Abstand haben, kann prinzipiell ein beliebiger Knoten ausgewählt werden. Es bietet sich jedoch an, diese Wahl nicht dem Zufall zu überlassen, sondern gezielt einen möglichst „guten“ Knoten auszuwählen. Ich habe mehrere Strategien implementiert, die im Folgenden kurz vorgestellt werden.

1. Die Knoten werden willkürlich sortiert.
2. Die Knoten werden aufsteigend nach der Anzahl der Geschwister sortiert. Damit soll verhindert werden, dass ein Knoten ausgeführt wird, wenn er viele Geschwister hat (die dann in den meisten Fällen direkt danach ausgeführt werden würden). Die Suche soll stattdessen möglichst an einer anderen Stelle im Baum fortgesetzt werden.
3. Die Knoten werden absteigend nach der Ebene im Baum sortiert. Dadurch sollen Knoten bevorzugt werden, bei denen schon relativ viele Attribute definiert sind und somit nur noch wenig Attribute mit Werten belegt werden müssen.
4. Die Knoten werden erst absteigend nach Ebene sortiert. Die *Suchknoten* werden anschließend aufsteigend nach der Größe des zu durchsuchenden Indexes sortiert. Dadurch soll eine Suche in großen Indizes vermieden werden.
5. Die Knoten werden erst absteigend nach Ebene sortiert. Die *Suchknoten* werden anschließend aufsteigend nach der Suchtoleranz sortiert. Dadurch soll eine Suche mit einer hohen Suchtoleranz vermieden werden.
6. Die Knoten werden erst absteigend nach Ebene sortiert. Die *Suchknoten* werden anschließend aufsteigend nach dem Produkt von Indexgröße und Suchtoleranz sortiert. Dadurch soll eine Suche mit einer hohen Suchtoleranz in großen Indizes vermieden werden.

In Abbildung 8.5 sind die durchschnittlichen Anfragezeiten dieser sechs Strategien dargestellt. Variante 6 (Sortierung nach Ebene absteigend und anschließend nach Indexgröße · Suchtoleranz aufsteigend) erwies sich hier knapp als die beste Strategie. Sie wurde daher für die weiteren experimentellen Untersuchungen verwendet.

8.4.3. Einfacher Index

In diesem Abschnitt werden kurz die Ergebnisse der Untersuchung des einfachen Indexes zur Indizierung von Datensätzen angegeben. Bei diesem einfachen Index werden die Datensätze als Zeichenketten kodiert und in einem Trie indiziert (siehe Abschnitt 5.3.1). Ziel war hier das Finden einer möglichst guten Konfiguration, um diese anschließend mit dem Fri vergleichen zu können. Der einzige Freiheitsgrad, der bei diesem Index gegeben ist, ist die Reihenfolge

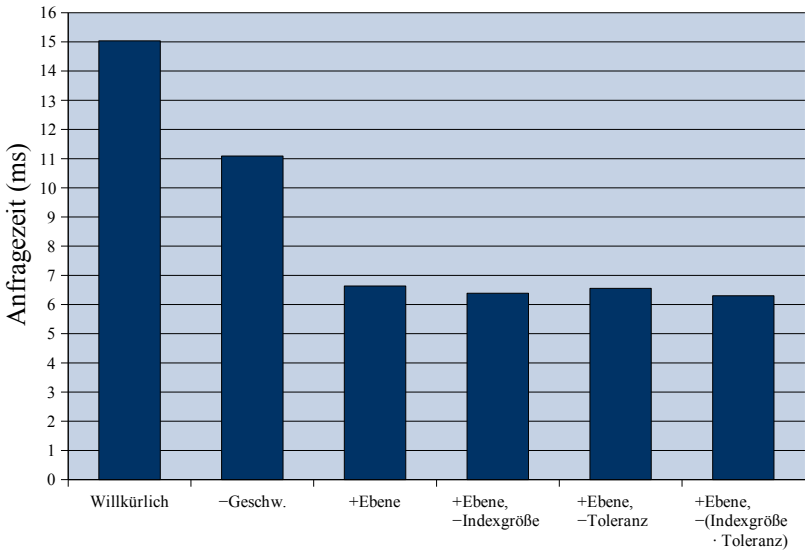

Abbildung 8.5.: Anfragezeiten für die verschiedenen Prioritätsberechnungen

der Attribute bei der Kodierung. Es wurden daher alle Permutationen der drei Attribute *ort*, *plz* und *str* betrachtet. Die Untersuchung erfolgte mit der kompletten Datenbank $D_{1,2\,\text{Mio}}$ und der Testmenge T_{einfach}.

Index	Datenbank	Testmenge
I_{Trie}	$D_{1,2\,\text{Mio}}$	T_{einfach}

In Abbildung 8.6 ist der Speicherbedarf für jede der sechs Permutationen dargestellt. Der Speicherplatz war jeweils unterschiedlich, weil die Einsparung durch die gemeinsame Speicherung der Präfixe von der Reihenfolge der Attribute abhängt. Wenn beispielsweise als erstes Attribut *ort* gewählt wird, muss jeder Ortsname nur einmal abgespeichert werden, unabhängig davon, wie viele zugehörige Werte es für die Attribute *plz* und *str* gibt. Wenn *ort* jedoch als drittes Attribut der Kodierung gewählt wird, müssen der Werte des Attributs *ort* für jeden Datensatz gespeichert werden. Als besonders platzsparend erwies es sich hier, die Attribute *ort* und *plz* beziehungsweise *plz* und *ort* als erste Attribute zu verwenden, weil diese eng korreliert sind und es somit bei einer Konkatenation viele gemeinsame Präfixe gibt.

Jedoch hängt nicht nur der Speicherbedarf, sondern auch die durchschnittliche Anfragezeit von der Reihenfolge der Attribute ab. Man könnte vermuten, dass die Anfragezeit geringer ist, wenn ein relativ selten verändertes Attribut wie die Postleitzahl das erste Attribut der Kodierung ist. Dies war jedoch nicht der Fall, wie in Abbildung 8.7 zu sehen ist. Die durchschnittliche Suchlaufzeit verhielt sich für die verschiedenen Varianten ähnlich wie der Speicherbedarf, jedoch waren die Unterschiede hier noch stärker ausgeprägt.

Als schnellste Variante stellte sich hier die Kodierung der Attribute in der Reihenfolge *ort*, *plz* und *str* heraus. Diese Variante wurde daher beim folgenden Vergleich mit der Fri-Datenstruktur verwendet.

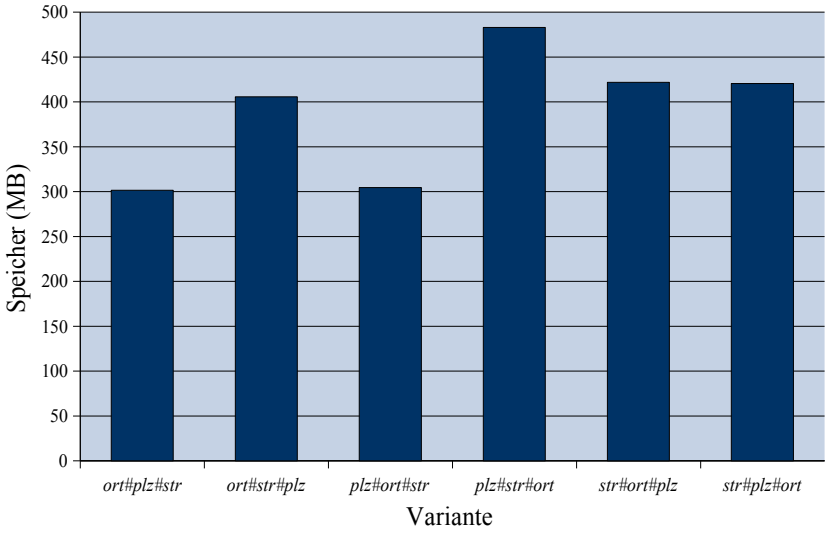

Abbildung 8.6.: Speicherplatz der einfachen Indexstrukturen

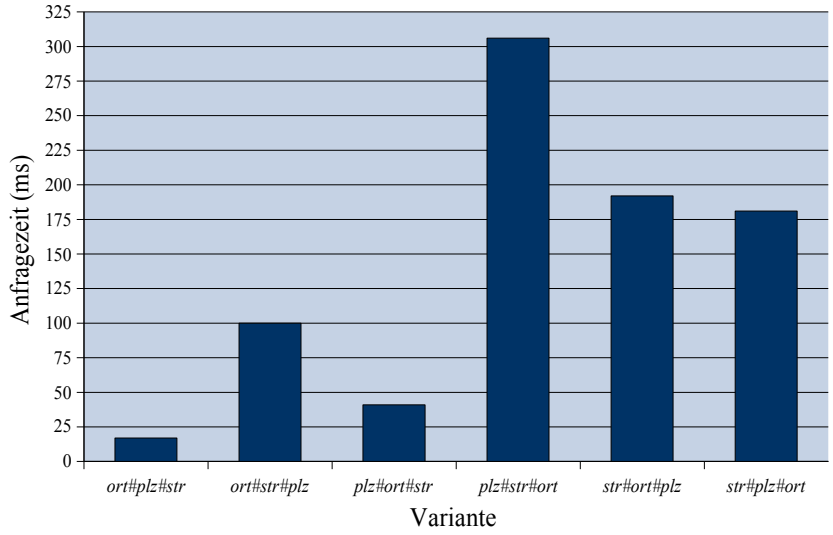

Abbildung 8.7.: Anfragezeiten der einfachen Indexstrukturen

8.4.4. Vergleich

In einem weiteren Experiment wurde ein Vergleich der Indexstrukturen I_{Trie} und I_{FRI} vorgenommen. Dabei kam jeweils die effizienteste Variante zum Einsatz (beim FRI ist der Parameter $minSizeTrie = 100$, die Reihenfolge der Attribute beim einfachen Index ist *ort#plz#str*). Unter anderem wurde der Speicherplatz und die durchschnittliche Anfragezeit für die verschiedenen Datenmengen $D_{5.000}, \ldots, D_{1{,}2\,\text{Mio}}$ untersucht. Als Testmenge kam weiterhin T_{einfach} zum Einsatz, weil die einfache Indexstruktur nicht die Möglichkeit bietet beispielsweise Synonyme zu erfassen.

Index	Datenbank	Testmenge
I_{Trie}, I_{FRI}	$D_{5.000}, \ldots, D_{1{,}2\,\text{Mio}}$	T_{einfach}

In Abbildung 8.8 ist der Speicherbedarf der beiden Indexstrukturen im Vergleich dargestellt. Dabei ist zu sehen, dass der FRI bei der größten Datenbank zwei bis drei mal so viel Platz (821,7 MB gegenüber 301,6 MB) benötigte wie der einfache Trie-Index. Dies liegt daran, dass die Datenstruktur beim FRI deutlich umfangreicher ist und beispielsweise für jeden Wert eines Attributs einen eigenen kleinen Index verwaltet. Das erwartete sublineare Wachstum der Tries in Abhängigkeit von der Datenbankgröße zeigt sich hier jedoch nicht. Offensichtlich hat die gemeinsame Speicherung von Präfixen bei der verwendeten Datenbank keinen großen Einfluss.

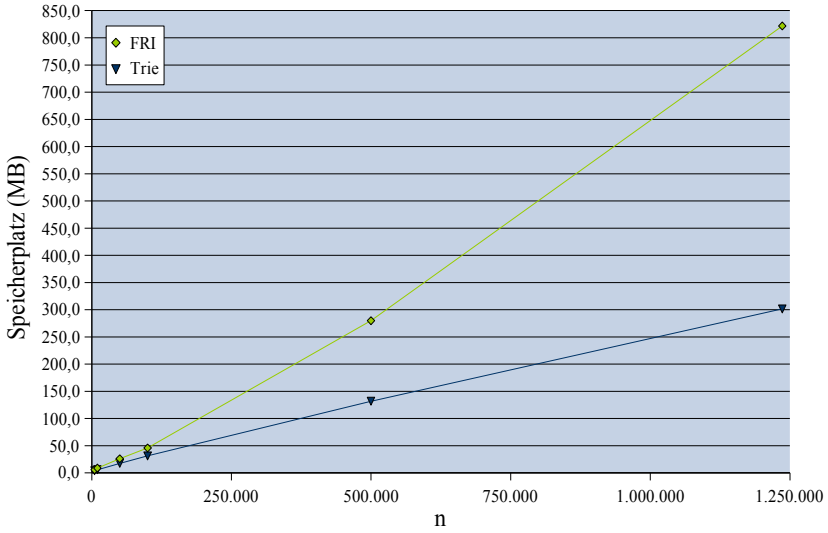

Abbildung 8.8.: Vergleich des Speicherbedarfs von I_{Trie} und I_{FRI}

In Abbildung 8.9 ist die durchschnittliche Anfragezeit der Datensätze in der Menge T_{einfach} für beide Indizes in Abhängigkeit der Datenbankgröße abgetragen. Dabei zeigt sich für beide Indexstrukturen deutlich die sublineare Abhängigkeit der Anfragezeit von der Datenbankgröße. Der Verlauf ist sehr ähnlich, was die asymptotische Analyse unterstützt, bei der die Laufzeit beim FRI in Abhängigkeit der Laufzeit von Tries angegeben wurde (allerdings für den ungünstigsten Fall und nicht wie hier bei den experimentellen Untersuchungen für einen

durchschnittlichen Fall). Es ist außerdem zu sehen, dass die durchschnittliche Anfragezeit beim FRI deutlich unter der des einfachen Indexes lag. Bei der kompletten Datenbank $D_{1,2\,\text{Mio}}$ war der FRI zwei bis drei mal so schnell.

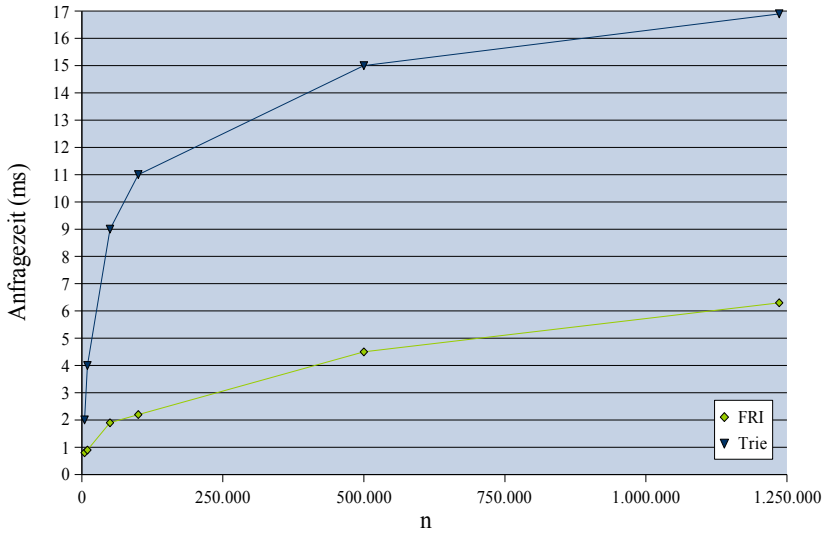

Abbildung 8.9.: Vergleich der Anfragezeit von I_{Trie} und I_{FRI}

In Tabelle 8.10 ist aufgelistet, wie viele Suchen mit welcher Suchtoleranz einerseits beim einfachen Index und andererseits beim FRI durchgeführt wurden. Bei I_{Trie} sind dies die Suchen im gesamten Trie. Beispielsweise wurde für jeden der 3.512 Datensätze eine Suche mit Toleranz 0 durchgeführt und für 1.319 musste eine Suche mit Toleranz 1 durchgeführt werden.[10] Beim FRI gibt die Zahl jeweils die Anzahl aller Suchen in sowohl den Indizes der ersten als auch denen der zweiten Kategorie an.

Es ist abzulesen, dass beim FRI deutlich weniger Suchen mit einer hohen Suchtoleranz ausgeführt wurden als beim einfachen Index. Dafür wurden mehr Suchen mit einer geringeren Toleranz durchgeführt. Beim einfachen Index wird bei jeder Suche die gesamte Datenbank durchsucht, während sich die Suchen im FRI auf kleinere Indizes beschränken (nur jeweils die verschiedenen Werte eines Attributs werden durchsucht). Die Suchen mit einer hohen Suchtoleranz werden beim FRI außerdem vor allem in den kleineren Indizes der zweiten Kategorie durchgeführt, weil auf der obersten Ebene nur selten mit einer hohen Toleranz gesucht wird.

Die schnellere Anfragebeantwortung beim FRI kann also unter anderem damit erklärt werden, dass nur wenig Suchen mit einer hohen Toleranz in den Indizes durchgeführt werden. Die hohe Anzahl der Suchen mit einer geringen Toleranz fällt dabei kaum ins Gewicht, weil beispielsweise eine Suche mit Toleranz 0 einem einfachen Nachschlagen im Trie entspricht; die Laufzeit liegt dabei im Mikrosekunden-Bereich.

In Abbildung 8.10 ist die durchschnittliche Anfragezeit der beiden Indexstrukturen nach Distanz des ähnlichsten Datensatzes aufgeschlüsselt. Dabei ist deutlich das exponentielle

[10] Bei einer Suche des ähnlichsten Werte im Trie wird die Suchtoleranz beginnend mit 0 schrittweise um 1 erhöht bis ein Ergebnis gefunden wurde (siehe Abschnitt 5.2.2).

Suchtoleranz	I_{FRI}	I_{Trie}
0	14.862	3.512
1	5.814	1.319
2	2.662	872
3	1.854	465
4	490	245
5	267	135
6	86	95
7	35	76
8	22	50
9	8	35
10	4	23
11	2	18
12	0	14
13	0	5
14	0	3

Tabelle 8.10.: Anzahl der Suchen aufgeschlüsselt nach Suchtoleranz

Wachstum bezüglich der nötigen Suchtoleranz beim Trie und auch beim FRI zu erkennen. Die Laufzeit beim FRI schwankt dabei jedoch und ist, wie schon bei der theoretischen Analyse (Abschnitt 5.3.6) gesehen, stark abhängig von der Verteilung der Werte in der Datenbank.

Abschließend kann festgehalten werden, dass die einfache Indexstruktur I_{Trie} für die Adresskorrektur zu langsam wäre, weil sie in vielen Fällen (insgesamt 101 bei T_{einfach}) länger als 100 ms für die Beantwortung einer Ähnlichkeitsanfrage benötigte. Bei der von mir vorgeschlagenen Indexstruktur I_{FRI} war dies nur ein Drittel so häufig (bei 34 Datensätzen) der Fall. Außerdem bietet der FRI die Möglichkeit der Verwaltung von Synonymen, was beim einfachen Trie nicht ohne Weiteres möglich ist.

8.4.5. Weitere Ergebnisse

Dieser Abschnitt präsentiert drei weitere Ergebnisse der experimentellen Untersuchungen. Die vorgenommenen Optimierungen zur Berechnung der Editierdistanz werden betrachtet, die im Suchalgorithmus aufgewandte Zeit analysiert und die Laufzeit bei einer Suche nach mehr als nur dem ähnlichsten Datensatz betrachtet.

Optimierung der Berechnung der Editierdistanz

Die Laufzeiten des Suchalgorithmus' wurden gemessen einerseits ohne und andererseits mit den in Abschnitt 5.3.7 erwähnten Optimierungen zur Berechnung der Editierdistanz (Abschätzung mit Länge und Hamming-Distanz sowie Ukkonen-Cutoff). Dabei ergab sich keinen messbarer Geschwindigkeitsvorteil. Sie kommen jedoch ohnehin nur bei der Suche in den Indizes zum Tragen, die weniger als $minSizeTrie$ Elemente haben. Nur bei diesen Listen wird die Editierdistanz einzeln für die Elemente berechnet; bei einer Suche in einem Trie sind die Optimierungen nicht anwendbar.

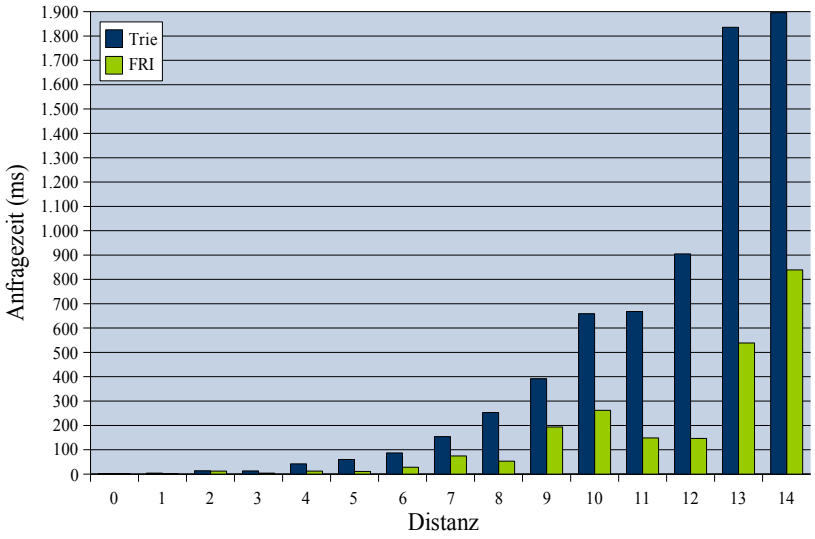

Abbildung 8.10.: Vergleich der Anfragezeit von I_{Trie} und I_{FRI} in Abhängigkeit der Distanz

Aufgewandte Zeit im Suchalgorithmus

Die Zeit beim Suchalgorithmus wird einerseits zur Suche in den Indexstrukturen, andererseits aber auch mit der Verwaltung verwendet, also beispielsweise zur Berechnung der mindestens erwarteten Distanz. Bei einer genaueren experimentellen Untersuchung hat sich herausgestellt, dass ein großer Teil der Zeit (durchschnittlich mehr als 50 %) bei einer Suchanfrage nur für die Berechnung der mindestens zu erwartenden Distanz für die Knoten in der Prioritätswarteschlange verwendet wurde. Dies wäre also sicherlich ein guter Ansatzpunkt, um Optimierungen der Implementierung vorzunehmen.

Suchen nach mehr als einem Datensatz

Wenn bei der Suche im FRI zu einem Anfragedatensatz nicht nur der eine ähnlichste, sondern mehrere Ergebnisse gefunden werden sollen, erhöht sich die Suchzeit. Es sind beispielsweise der ähnlichste Datensatz und außerdem alle solchen Datensätze gesucht, bei denen der Abstand der um höchstens 5 größer ist. Bei dieser Suchanfrage, der Datenbank $D_{1,2\,\text{Mio}}$ und der Testmenge T_{einfach} stieg die durchschnittliche Anfragezeit von 6,3 ms (Suche des ähnlichsten Datensatzes) auf 57,6 ms. Dies ist ein recht starker Anstieg und daher sollten solche Suchanfragen mit einer hohen Suchtoleranz beispielsweise bei der Adresskorrektur vermieden werden.

8.4.6. FRI bei der Adresskorrektur

In diesem Abschnitt werden Untersuchungsergebnisse bezüglich der Effizienz vom FRI bei der Verwendung für die Adresskorrektur dargestellt. Die Datenbank unterscheidet sich von den bisherigen Betrachtungen darin, dass hier zusätzlich auch die Synonyme für die Attribute

ort und *str* eingesetzt wurden. Außerdem wurde die separate Behandlung für Großempfänger (siehe Abschnitt 6.2.2) sowie die Vorverarbeitung der Adresskorrektur (beispielsweise Umwandeln von Großbuchstaben, siehe Abschnitt 6.2.1) aktiviert. Die Experimente konnten daher auch mit der vollständigen Testmenge T_{alle} durchgeführt werden.

Index	Datenbank	Testmenge
I_{FRI}	$D_{1,2\,\text{Mio}}$ + Synonyme	T_{alle}

Weil in dieser Datenbank $D_{1,2\,\text{Mio}}$ hier zusätzlich auch Synonyme enthalten waren, wurde wiederum ein geeigneter Werte für den Parameter $minTrieSize$ bestimmt. Dieser unterschied sich von dem in Abschnitt 8.4.2 ermittelten Wert. In Experimenten mit mehreren Werten stellte sich heraus, dass die durchschnittliche Anfragezeit bei $minTrieSize = 20$ in diesem Fall am geringsten ist.

Der oben bestimmte Berechnungsstrategie für die Knotenpriorität ist hier ebenfalls geeignet.

Die Indexstruktur soll für die Adresskorrektur verwendet werden, bei der nur 100 ms zur Verfügung stehen. Als Abbruchkriterium wurde hier daher ein Zeitlimit von 100 ms verwendet.

Der Speicherplatz der FRI-Datenstruktur für die Datenbank $D_{1,2\,\text{Mio}}$ mit Synonymen betrug 1.255 MB. Der Aufbau des Indexes benötigte 180 s, also drei Minuten. Die durchschnittliche Anfragezeit betrug hier 3,6 s.[11]

Für den überwiegenden Teil der Adressen (3.769 von 3.834, also 98,3 %) konnte innerhalb der zur Verfügung stehenden Zeit eine ähnliche Adresse in der Datenbank gefunden werden. Bei einigen Adressen (65 von 3.834, also 1,7 %) konnte in dieser Zeit keine ähnliche Adresse gefunden werden. Dies hatte verschiedene Gründe, die in Abschnitt 8.5 erläutert werden (beispielsweise wurde die Adresse nicht richtig in die Bestandteile zerlegt). In Abbildung 8.11 ist die Anzahl der Adressen nach der benötigten Zeit zur Suche der ähnlichsten Adresse aufgeschlüsselt (gerundet auf volle 20 ms).

Es ist zu vermuten, dass auf dem bei der Briefsortierung eingesetzten Rechner die Anzahl der korrigierten Adressen etwas höher ist, weil dieser mit einer deutlich besseren Hardware ausgestattet ist als der hier verwendete Laptop. Dieser Rechner besitzt vier Prozessorkerne (INTEL CORE 2 QUAD), muss dafür jedoch die Berechnung auch für mehrere Adressen gleichzeitig durchführen, weil sich jeweils mehrere Briefe auf der Strecke zwischen Kamera und Weiche befinden (siehe Abschnitt 1.1).

8.5. Adresskorrektur

Dieser Abschnitt präsentiert die Untersuchungergebnisse der Qualität der Adresskorrektur. Die Ergebnisse werden dabei für die Testmenge T_{alle} angegeben, die sowohl die Testbriefe als auch die echten Briefe umfasst.

Index	Datenbank	Testmenge
I_{FRI}	$D_{1,2\,\text{Mio}}$ + Synonyme	T_{test}, T_{echt}, T_{alle}

Dafür wird jedoch zunächst erläutert, wie die Qualität eines Verfahrens zur Adresskorrektur gemessen werden kann. Ziel bei der Adresskorrektur ist es, zu möglichst vielen per OCR gelesenen Adressen die eigentlich korrekte Adresse in der Datenbank zu bestimmen. Die Qualität eines Verfahrens zur Adresskorrektur kann also als Anteil der richtig korrigierten

[11] Dies ist deutlich weniger als die in den vorangegangenen Abschnitten ermittelten 6,3 ms und liegt vor allen daran, dass hier die Suche spätestens nach 100 ms abgebrochen wurde.

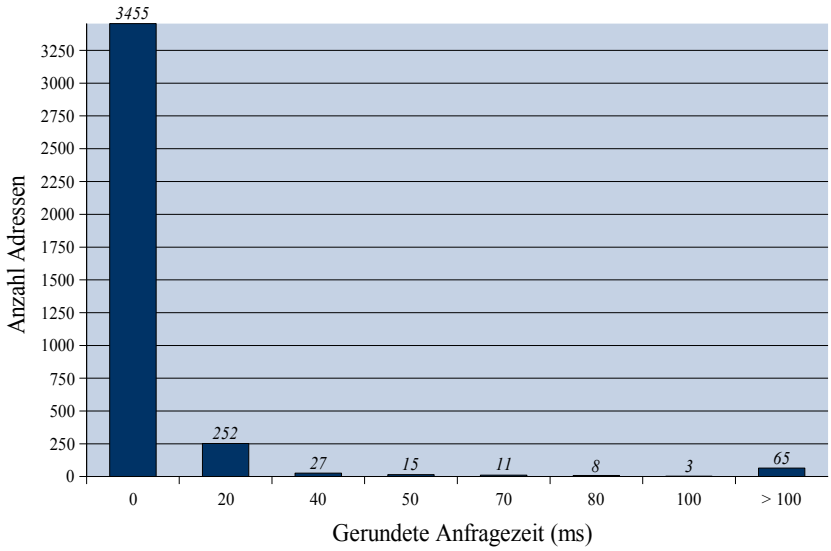

Abbildung 8.11.: Anzahl der Adressen aufgeschlüsselt nach gerundeter Anfragezeit

Adressen in einer Testmenge gemessen werden. In den von mir generierten Testmengen ist zu einer gelesenen Adresse jeweils auch die eigentlich korrekte Adresse vermerkt. Deshalb konnten die im folgenden Abschnitt erläuterten Verfahren jeweils auf eine Testmenge angewandt werden, und anschließend der Anteil der richtig bestimmten Adressen ermittelt werden.

8.5.1. Lernverfahren

Der Fri ist in der Lage, zu einer Anfrageadresse eine ähnliche Adresse in der Datenbank zu finden; dies geschieht bezüglich der einfachen Editierdistanz. Die so gefundene Adresse ist jedoch nicht notwendigerweise die korrekte Adresse. Bei dem vorgeschlagenen Verfahren zur Adresskorrektur werden daher im Fri mehrere Ergebniskandidaten gesucht, die dann anschließend bezüglich einer genaueren Ähnlichkeitsfunktion sortiert werden. In diesem Abschnitt wird untersucht, welchen Einfluss diese anschließende Sortierung auf die Qualität der Ergebnisse hat.

Um mehrere Ergebnisse bei der Suche im Fri zu erhalten, wurde daher nach dem ersten Fund weiter gesucht, bis die Suchtoleranz den Abstand des ersten Fundes um einen gewissen Schwellwert überstieg. Ich habe als Schwellwert hier den Wert 2 verwendet. Damit wurden zu einem Anfragedatensatz bis zu 99 Ergebniskandidaten erzeugt.[12] Insgesamt habe ich drei Verfahren zur Sortierung der Ergebniskandidaten experimentell untersucht:

Einfache Editierdistanz: Die Ergebniskandidaten werden mit Hilfe der einfachen Editierdistanzen sortiert. Dabei werden wieder die Gewichte $g_{ort} = 1$, $g_{plz} = 2$, $g_{str} = 1$ verwendet. Es ist der Datensatz r gesucht, bei dem folgender Ausdruck minimal ist:

$$g_{ort} \cdot \delta_{\text{edit}}(q.ort, r.ort) + g_{plz} \cdot \delta_{\text{edit}}(q.plz, r.plz) + g_{str} \cdot \delta_{\text{edit}}(q.str, r.str)$$

[12] Die Zahl 99 kommt hier zufällig zu Stande und ist nicht vorgegeben.

Sortierverfahren	Richtig		Falsch	
Einfache Editierdistanz	3.733	*99,04 %*	36	*0,96 %*
Gewichtete Editierdistanz	3.740	*99,23 %*	29	*0,77 %*
Gewichtete Editierdistanz + SVM	3.739	*99,20 %*	30	*0,80 %*

Tabelle 8.11.: Vergleich der Sortierverfahren

(Der FRI muss in diesem Fall nur einen Ergebniskandidaten liefern, dieser erfüllt genau diese Bedingung.)

Gewichtete Editierdistanz: Die Ergebniskandidaten werden mit Hilfe der gewichteten Editierdistanzen sortiert. Es ist der Datensatz r gesucht, bei dem folgender Ausdruck minimal ist:

$$\delta_{\text{edit}}(q.ort, r.ort) + \delta_{\text{edit}}(q.plz, r.plz) + \delta_{\text{edit}}(q.str, r.str)$$

Die Gewichtung der einzelnen Attribute ist hier schon implizit durch die Gewichte der Editieroperationen gegeben. Die Editieroperationen beim Attribut plz haben beispielsweise im Allgemeinen höhere Kosten, weil dieses Attribut auch in den Trainingsdaten seltener verändert auftauchte.

Gewichtete Editierdistanz und Support Vector Machine: Die Ergebniskandidaten werden mit Hilfe Klassifikationsfunktion der Support Vector Machine sortiert. Es ist der Datensatz r gesucht, bei dem folgender Ausdruck minimal ist:

$$\delta_{\text{svm}}(q, r)$$

Die Funktion δ_{svm} berechnet dabei intern die gewichtete Editierdistanz für jedes Attribut (siehe Abschnitt 7.2).

Die Qualität dieser drei Sortierverfahren wurde mit Hilfe der Testmenge T_{alle} evaluiert. Hier erfolgte also das Testen mit der gleichen Datenmenge wie schon das Training. Wegen der relativ kleinen Trainingsmenge habe ich mich für dieses Vorgehen entschieden, damit ein sichtbarer Effekt zu erkennen ist.[13] Im Anschluss an diese Arbeit sollte noch die externe Validität, also die Übertragbarkeit der erlernten Funktionen auf zum Trainingszeitpunkt unbekannte Daten überprüft werden.

Von den insgesamt 3.834 Datensätzen in T_{alle} wurden die 65 Adressen ausgelassen, für die innerhalb der zur Verfügung stehenden Zeit keine Ergebniskandidaten gefunden wurden. Für die verbleibenden 3.769 Datensätze wurde jeweils mit Hilfe vom FRI eine Menge von Ergebniskandidaten generiert und diese anschließend mit einem der drei Verfahren sortiert.[14]

Die Anzahl der von der Testmenge T_{alle} richtig zugeordneten Adressen ist in Tabelle 8.11 im Vergleich angegeben. Dabei zeigten sich nur recht geringe Unterschiede zwischen den drei Verfahren. Am besten schnitt die gewichteten Editierdistanz ab, beinahe gleichauf mit der gewichteten Editierdistanz bei anschließender Verwendung der Support Vector Machine (dabei wurden bis auf wenige Ausnahmen die Datensätze auch gleich zugeordnet).

Bei der gewichteten Editierdistanz wurde in 99,23 % der Fälle (alle bis auf 29 Adressen) die richtige Adresse bestimmt. Das ist eine Verbesserung gegenüber der einfachen Editierdistanz

[13] Das Training der Support Vector Machine fand jedoch mit einer Kreuzvalidierung statt, siehe Abschnitt 7.2.

[14] Die Sortierung benötigt kaum messbare Zeit, so dass die jeweils zur Verfügung stehenden 100 ms dadurch nicht überschritten werden.

von 8 zusätzlich korrekt bestimmten Adressen. Die Ursache dafür ist, dass die gewichtete Editierdistanz die spezifischen Fehler des OCR-Systems besser abbilden kann.

Die Support Vector Machine bot keinen Vorteil gegenüber der gewichteten Editierdistanz. Dies liegt unter anderem daran, dass auch die gewichteten Editierdistanz ein gutes Maß für die Ähnlichkeit von zwei Adressen ist. Bei der absteigenden Sortierung nach attributweise summierter, gewichteter Editierdistanz werden die Ergebniskandidaten implizit aufsteigend nach dem Produkt der attributweisen Wahrscheinlichkeiten sortiert (siehe Abschnitt 7.1.4). Dabei wird also die Adresse bestimmt, die folgenden Ausdruck maximiert:

$$\mathrm{P}(r.ort \mid q.ort) \cdot \mathrm{P}(r.plz \mid q.plz) \cdot \mathrm{P}(r.str \mid q.str)$$

Dies entspräche der Wahrscheinlichkeit $\mathrm{P}(r \mid q)$, wenn man Unabhängigkeit der Fehler in den einzelnen Attributen voraussetzen würde. Diese Unabhängigkeit ist hier zwar beispielsweise auf Grund von großflächigen OCR-Fehlern (engl.: burst errors) nicht gegeben, dennoch ist dieses Maß offensichtlich eine sinnvolle Methode um die Ergebniskandidaten zu sortieren. Dies erklärt das gute Abschneiden der Sortierung mit Hilfe der gewichteten Editierdistanz.

Die Unterschiede zwischen den Verfahren waren insgesamt jedoch nur recht gering. Dies liegt daran, dass schon einfachen Editierdistanz mit den Gewichten $g_{ort} = 1, g_{plz} = 2, g_{str} = 1$ relativ viele Adressen korrigieren konnte. Deshalb konnten die feineren Sortiermethoden nur noch einen kleinen Vorteil erreichen. Die verbleibenden 29 Adressen wiesen in der Regel Fehler auf, die kaum von der Sortierung der Ergebniskandidaten abhängen. Beispielsweise wurden die Adressbestandteile falsch extrahiert oder die Adresse ist bei der optischen Zeichenerkennung stark entstellt worden. Eine Liste dieser falsch bestimmten Adressen findet sich in Anhang A.4.

Bemerkung Hier wurde die Support Vector Machine nur für die Sortierung der Ergebniskandidaten verwendet. Sie könnte auch dazu verwendet werden, einen Ergebnisdatensatz r zu verwerfen, falls das Tupel (q, r) von der Klassifikationsfunktion der SVM als „nicht ähnlich“ klassifiziert wird. Dieses Vorgehen wurde ebenfalls getestet. In den Fällen, in denen die ähnlichste Adresse jedoch einen großen Abstand zur Anfrageadresse hat, wird sie verworfen, selbst wenn sie die eigentlich korrekte Adresse ist. Die SVM wird daher hier nicht für die Klassifikation verwendet.

8.5.2. Ergebnis

In diesem Abschnitt werden die Ergebnisse der Evaluation des Adresskorrektur-Systems präsentiert. Dabei wird die komplette Testmenge T_{alle} betrachtet, auch inklusive der Datensätze, zu denen im Fri keine ähnliche Adresse gefunden wurde. Im Folgenden wird unter anderem beispielsweise analysiert, wie viele Postleitzahlen vom Adresskorrektursystem korrekt bestimmt werden konnten. Die Ergebnisse der optischen Zeichenerkennung (also vor der Adresskorrektur) und die Ergebnissen der Adresskorrektur werden dabei jeweils gegenübergestellt.

Die wichtigsten Ergebnisse sind in Tabelle 8.12 aufgeführt.[15] Die erste Spalte bezieht sich auf die Ergebnisse der optischen Zeichenerkennung und die zweite bezieht sich auf die Ergebnisse der Adresskorrektur. (Die dritte Spalte wird weiter unten noch erläutert.) Die ersten drei Zeilen führen die Ergebnisse bezüglich der gesamten Adresse auf. Die Zahl 2.355 bedeutet beispielsweise, dass 2.355 der per OCR gelesenen Adressen genau so auch in der Datenbank enthalten waren (siehe dazu auch Abschnitt 8.3). Die Zahl 29 in der zweiten

[15] Diese Tabelle ist in Anhang A.6 für die Testmengen T_{test} und T_{echt} aufgeschlüsselt dargestellt.

	OCR		Adresskorrektur		Adr. + Verw.[17]	
Adresse richtig	2.355	*61,4 %*	3.740	*97,5 %*	3.740	*97,5 %*
Adresse falsch	1.479	*38,6 %*	29	*0,8 %*	94	*2,5 %*
Verworfen	—		65	*1,7 %*	—	
PLZ richtig	3.738	*97,5 %*	3.763	*98,1 %*	3.820	*99,6 %*
PLZ falsch	96	*2,5 %*	6	*0,2 %*	14	*0,4 %*
Verworfen	—		65	*1,7 %*	—	
LR richtig	3.790	*98,9 %*	3.768	*98,3 %*	3.827	*99,8 %*
LR falsch	44	*1,1 %*	1	*0,0 %*	7	*0,2 %*
Verworfen	—		65	*1,7 %*	—	

Tabelle 8.12.: Ergebnis der Adresskorrektur

Spalte bedeutet dementsprechend, dass bei Adresskorrektur 29 Adressen falsch zugeordnet wurden (bei insgesamt 3.834 Adressen in T_{alle} entspricht das 0,8 %).

Die folgenden drei Zeilen beziehen sich auf die Ergebnisse, wenn man nur das Attribut *plz* betrachtet. Wie schon in Abschnitt 8.3 erwähnt, war bei 96 der Testdatensätze die Postleitzahl falsch. Die letzten drei Zeilen stellen die Statistiken bezüglich der Leitregion (LR) der Postleitzahl dar.[16] Dies ist vor allem für die Briefsortierung relevant, weil die Briefe je nach Leitregion in ein anderes Fach einsortiert werden (siehe Abschnitt 1.1).

Die Zeile *Verworfen* enthält die Adressen, zu denen in der vorgegebenen Zeit kein passender Datensatz in der Datenbank gefunden wurde.

Es war auf allen drei Ebenen (gesamte Adresse, Postleitzahl, Leitregion) eine deutliche Verbesserung der Qualität der Ergebnisse der Adresskorrektur gegenüber den nicht korrigierten OCR-Ergebnissen zu beobachten. Falsch zugeordnete Adressen gab es beispielsweise nur in 0,8 % der Fälle.

Wenn man nur die falsch erkannten Adressen betrachtet und die von vornherein richtig gelesenen Adressen außen vor lässt, ergibt sich auf der Ebene der Adressen eine Korrekturrate von $\frac{3.740-2.355}{1.479} = 93{,}6\,\%$.

Noch nicht betrachtet wurden jedoch bisher die nicht korrigierten, also verworfenen Adressen. Diese Briefe können beispielsweise aussortiert werden, um dann in einem zweiten Versuch möglicherweise besser eingelesen und richtig sortiert zu werden. Eine weitere Möglichkeit wäre es, bei diesen Adressen die von der OCR erkannte Postleitzahl für die Sortierung zu verwenden. Dies ist in vielen Fällen richtig, weil die Postleitzahl bekanntermaßen ohnehin sehr häufig korrekt ist. Die Ergebnisse, wenn bei den nicht korrigierten Adressen einfach die gelesene Adresse verwendet wird, sind in der dritten Spalte der Tabelle 8.12 dargestellt. In dieser Spalte gibt es also keine verworfenen Datensätze mehr, sondern diese wurden jeweils in die Kategorien *PLZ richtig*/*PLZ falsch* beziehungsweise *LR richtig*/*LR falsch* einsortiert. (Unter den Verworfenen sind einige Adressen mit von vornherein richtiger und einige mit falscher Postleitzahl.) Bei diesem Verfahren steigt also die Anzahl der richtig erkannten Postleitzahlen, aber auch die Anzahl der falsch erkannten. Ob dieses Verfahren eingesetzt wird, muss daher für den praktischen Einsatz noch geklärt werden.

Die Postleitzahl war auch vor der Korrektur schon in nur sehr wenig Fällen (2,5 %) falsch, jedoch konnte auch hier eine Verringerung auf 0,4 % erreicht werden. Die Leitregion ist nach der Korrektur bei den Testdaten sogar nur ein einziges Mal falsch bestimmt. In Bild 8.12 ist die Verbesserung der Qualität mit Hilfe der Adresskorrektur graphisch dargestellt.

[16] Die Leitregion einer Postleitzahl sind die ersten beiden Ziffern.
[17] Adresskorrektur zuzüglich verworfene Adressen

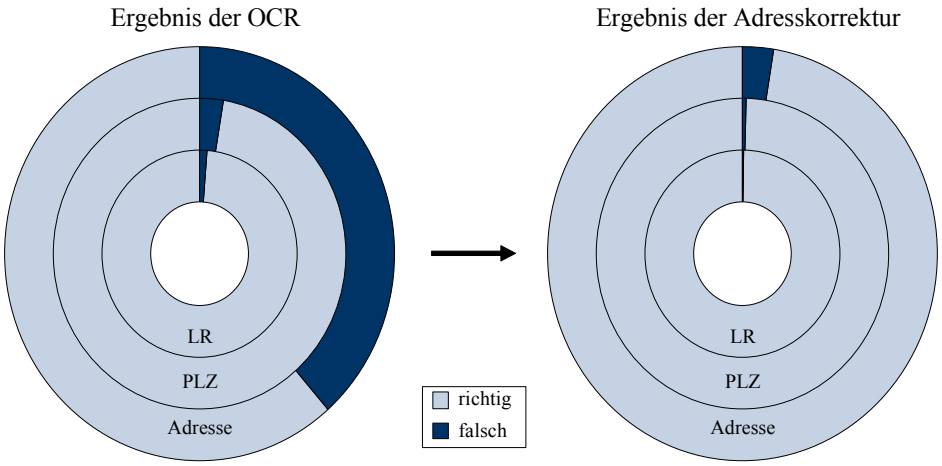

Abbildung 8.12.: Ergebnis der Adresskorrektur

In Tabelle 8.13 sind die 7 Adressen aufgeführt, deren Leitregion nach der Adresskorrektur falsch war, wenn man die nicht korrigierten Postleitzahlen ebenfalls übernimmt (diese entsprechen der Zelle ganz rechts unten in Tabelle 8.12). Dazu steht jeweils in der rechten Spalte die manuell bestimmte korrekte Adresse. Man kann an diesen Beispielen einige Schwierigkeiten erkennen, die bei der automatischen Adresskorrektur zu bewältigen sind. Teilweise wurden die Zeichen nur schlecht gelesen („F:9rHVVF::rF“ statt „Eschwege“) oder es standen weitere Vermerke im Adress-Textfeld („bearbeiter:...“).

Insgesamt funktionieren die von mir entwickelten Verfahren und Heuristiken sehr zufriedenstellend. Die verwendeten Daten (inklusive der generierten Synonyme, siehe Abschnitt 6.1), die Schritte der Vorverarbeitung (Extraktion der Adressbestandteile, Aufteilen von Orts- und Straßennamen an Trennzeichen, Kleinschreibung von Großbuchstaben, siehe Abschnitt 6.2.1), die Behandlung von Postfächern und Großempfängern (siehe Abschnitt Abschnitt 6.2.2), die FRI-Datenstruktur und auch die anschließende Sortierung leisten jeweils einen Beitrag daran, dass insgesamt eine sehr hohe Korrekturquote erreicht wird.

8.6. Zusammenfassung

In diesem Kapitel wurden zunächst die Eckdaten der verwendeten Datenbank angegeben, und die Erzeugung der Trainings- und Testdaten beschrieben. Es wurde untersucht, wie oft die verschiedenen Attribute von Fehlern betroffen sind und dabei stellte sich heraus, dass die Postleitzahl in vielen Fällen schon korrekt erkannt wird. Außerdem wurden die aufgetretenen Editieroperationen angegeben, die häufig auftraten und denen damit eine hohe Wahrscheinlichkeit (beziehungsweise geringe Kosten bei der Editierdistanz) zugeordnet werden.

In mehreren Experimenten wurden geeignete Werte für den Parameter $minSizeTrie$ und die Berechnung der Priorität der Knoten im Suchalgorithmus bestimmt. Für die minimale Größe eines Tries anstelle einer Liste hat sich $minSizeTrie = 100$ als geeignet herausgestellt,

[18] Diese Adresse wurde fälschlicherweise als „Jägerstr., 12526 Berlin“ korrigiert.

OCR-Adresse	Korrekte Adresse
Flurstraße ? 70749 Tiefenbronn	Flurstr. ? 75233 Tiefenbronn
Schlossplatz ? 17913 F:9rHVVF::rF	Schloßplatz ? 37269 Eschwege
Tschaikowskistraße ?? 04105 Leipzig 43070 Bearbeiter: jahn Tel,	Tschaikowskistr. ?? 04105 Leipzig
Kingelhöferstraße ? 12785 Bern	Klingelhöferstr. ? 10785 Berlin
VaÂ<Mtz12 1178Berin 1681 Mte*btanum	Monbijouplatz ?? 10178 Berlin
?????? ???????? ‚jäger- str.?? 15566 Schbi iciche/- Berlin	Jägerstr. ?? 15566 Schöneiche[18]
Schnellerstr. ??? All40 10249 Berlin	Schnellerstr. ??? 12439 Berlin

Tabelle 8.13.: Adressen mit falscher Leitregion

um sowohl einen geringeren Speicherbedarf, als auch eine schnellere Anfragebeantwortung zu erreichen. Die Knotenpriorität wird auf Basis der Ebene des Knotens im Entscheidungsbaum und den lokalen Berechnungskosten der Suchknoten bestimmt.

Für die einfache Indexstruktur wurde die beste Reihenfolge der Attribute bestimmt, so dass sowohl der Speicherbedarf möglichst klein, als auch die durchschnittliche für die Ähnlichkeitssuche aufgewandte Zeit möglichst gering ist. Diese Variante wurde anschließend mit dem Fri hinsichtlich Speicherbedarf und Anfragezeit bei einer Ähnlichkeitssuche verglichen. Der Speicherbedarf vom Fri stellt sich dabei als knapp zwei bis drei mal so groß heraus, jedoch kann die durchschnittliche Anfragezeit auf gut ein Drittel reduziert werden. Dies wird unter anderem damit erklärt, dass beim Fri deutlich weniger Suchen in den Indizes mit einer hohen Suchtoleranz durchgeführt werden. Es stellt sich weiterhin heraus, dass die Optimierungen der Berechnung der Editierdistanz kaum einen Einfluss auf die Anfragezeit haben, aber insgesamt ungefähr die Hälfte der Zeit für der Berechnung der mindestens erwarteten Distanz verwendet wird.

Abschließend wurde die Qualität der Ergebnisse der Adresskorrektur betrachtet. Dabei stellte sich heraus, dass die Verwendung der gewichteten Editierdistanz zur Sortierung der Ergebniskandidaten eine Verbesserung gegenüber der einfachen Editierdistanz zur Folge hat. Die zusätzliche Verwendung der Support Vector Machine wirkte sich jedoch nicht positiv aus. Insgesamt konnten mit dem vorgeschlagenen System zur Adresskorrektur 97,5 % der Adressen der Testmenge korrekt bestimmt werden. Wenn nur die Postleitzahl betrachtet wird, waren sogar 99,6 % korrekt.

9. Schluss

9.1. Zusammenfassung

Ziel dieser Diplomarbeit war die Entwicklung eines Systems zur Adresskorrektur. In der Einleitung wurde zunächst das Umfeld und die technischen Anforderungen an ein solches System beschrieben. Außerdem wurden sechs über das konkrete Problem der Adresskorrektur hinausgehende Fragen aufgeworfen, die nach geeigneten Ähnlichkeitsmaßen, Indexstrukturen und Lernverfahren jeweils für Zeichenketten und für Datensätze fragen. Diese wurden in den folgenden Kapiteln beantwortet.

Zunächst wurden die theoretischen Grundlagen erläutert und die bei der Adresskorrektur zu erwartenden Fehler untersucht, mit einem Schwerpunkt auf den OCR-Fehlern. Verwandte Arbeiten aus verschiedenen Anwendungsbereichen (Datensatzverknüpfung, Datenbanken mit Unsicherheit, Zeichenkettensuche, Rechtschreibkorrektur, OCR-Nachkorrektur, Adresskorrektur) wurden vorgestellt. Dabei wurden insbesondere auch die Ansätze dieser Arbeiten zur Definition der Ähnlichkeit von Zeichenketten und Datensätzen sowie Indexstrukturen (metrische Bäume, Tries, k-Gramme) angegeben.

Eine Reihe unterschiedlicher Ähnlichkeitsmaße für Zeichenketten wurden detailliert vorgestellt, analysiert und auf ihre Eignung für die Adresskorrektur hin untersucht. Dabei stellte sich heraus, dass insbesondere die Editierdistanz (Levenshtein-Distanz) und noch besser die gewichtete Editierdistanz geeignet ist, um die Fehler der optischen Zeichenerkennung zu modellieren. Dies bestätigte sich auch in den experimentellen Untersuchungen. Es wurde außerdem erläutert, wie die Ähnlichkeit von Datensätzen bestimmt werden kann. Die Datensätze werden auf einen Vektor abgebildet, dessen Komponenten die attributweisen Abstände der beiden Datensätze sind. Basierend auf diesem Vektor und einer Klassifikationsfunktion wird anschließend die Ähnlichkeit der Datensätze berechnet.

Als Indexstruktur für eine Ähnlichkeitssuche in Zeichenketten hat sich der Trie als gut geeignet herausgestellt. In einem Trie können effizient exakte Suchen und auch Ähnlichkeitssuchen bezüglich der Editierdistanz durchgeführt werden, wenn die Suchtoleranz nicht zu hoch ist. Die zugehörigen Datenstrukturen und Algorithmen wurden erläutert. Als Indexstruktur für Datensätze haben sich Verfahren wie metrische Bäume, k-d-Bäume und R-Bäume als nicht effizient herausgestellt, so dass eine neue Datenstruktur zur Indizierung von Datensätzen namens Fri (Fast Record Index) vorgeschlagen wurde. Diese basiert auf mehreren Tries als Indizes für die Attributwerte und erlaubt auch die Integration von Synonymen. Es wurde ein Algorithmus angegeben, der eine Ähnlichkeitssuche im Fri ermöglicht. Die im Algorithmus zu treffenden Entscheidungen werden dabei mit einem Entscheidungsbaum modelliert, wobei die Auswertung immer an der jeweils günstigsten Stelle fortgesetzt wird. Die Laufzeit wurde unter zwei vereinfachenden Annahmen abgeschätzt. Außerdem wurde eine Erweiterung zur Einsparung von Speicherplatz und zur Beschleunigung der Suche vorgestellt. Diese hat sich in der Evaluation als effektiv und effizient herausgestellt.

Beim Verfahren zur Adresskorrektur wurde zunächst erläutert, wie die Datensätze und Synonyme mit Hilfe der gegebenen Datenbank erzeugt werden. Das System zur Adresskorrektur wurde in den Phasen zur Vorverarbeitung, Kandidatengenerierung mit Hilfe der

FRI-Datenstruktur und anschließender Sortierung beschrieben.

Es wurden Lernverfahren für die Ähnlichkeitsfunktion von Zeichenketten und von Datensätzen vorgestellt, die jeweils mit Hilfe einer Trainingsmenge erlernt werden. Die Gewichte der Editierdistanz von Zeichenketten werden so bestimmt, dass sie die Wahrscheinlichkeit der jeweiligen Editieroperation repräsentieren. Die Editierdistanz von zwei Zeichenketten repräsentiert eine Annäherung an die Übergangswahrscheinlichkeit von der einen Zeichenkette zur anderen. In den experimentellen Untersuchungen konnte dieses erlernte Ähnlichkeitsmaß die Ähnlichkeit von Zeichenketten besser bestimmen, als die einfache, ungewichtete Editierdistanz. Die Ähnlichkeitsfunktion für Datensätze basiert auf einer Klassifikationsfunktion für Vektoren. Diese wird mit Hilfe einer Support Vector Machine und Trainingsdaten erlernt. Bei der Evaluation zeigt sich jedoch kein Vorteil dieses Verfahrens gegenüber der Verwendung der gewichteten Editierdistanz.

In der abschließenden Evaluation wurde unter anderem die verwendete Datenbank und die Erzeugung der Testmengen beschrieben. Die Fehler wurden auf ihre Häufigkeiten untersucht und die erlernten Wahrscheinlichkeiten der Editieroperationen angegeben. Außerdem wurde für die vorgeschlagene Indexstruktur der Einfluss der Parameter analysiert und sie anschließend hinsichtlich von Speicherbedarf und Anfragezeit mit einem einfachen Trie als Index für Datensätze verglichen. Sie zeigte dabei eine deutlich geringere Laufzeit bei der Ähnlichkeitssuche. Abschließend wurde die Qualität des Systems zur Adresskorrektur evaluiert. Dabei hat sich gezeigt, dass die Adressen der Testmengen mit dem vorgeschlagenen Verfahren in 97,5 % der Fälle korrekt erkannt wurden. Die Postleitzahl ist bei 99,6 % der getesteten Adressen korrekt bestimmt worden.

9.2. Fazit und Ausblick

Hauptbeiträge dieser Arbeit sind die FRI-Datenstruktur zur Indizierung von Datensätzen und das Verfahren zur Adresskorrektur. Im Folgenden werden beide Teile rückblickend betrachtet und bewertet, und es wird ein Ausblick auf mögliche zukünftige Arbeiten in diesen Bereichen gegeben.

Indexstruktur für Datensätze

Das Hauptentwicklungsziel beim FRI war das Erreichen einer geringen Laufzeit der Ähnlichkeitssuche. Diese sollte möglichst wenig von der Anzahl der gespeicherten Datensätze abhängen und auch Datensätze mit einem hohen Abstand schnell finden können. Dies sollte dadurch erreicht werden, dass nicht *eine* Suchanfrage mit einer hohen Toleranz in *einem* Trie durchgeführt wird, sondern *mehrere* Suchanfragen in *mehreren* Tries mit einer geringeren Suchtoleranz. Es hat sich herausgestellt, dass diese Idee funktioniert und die Suchzeit verringern kann. Das Prinzip ist dabei recht ähnlich zum Prinzip des Rückwärtswörterbuchs (siehe Abschnitt 3.2.3): Es wird mehr Speicherplatz verwendet und das Suchobjekt bei der Ähnlichkeitssuche in Teile aufgespalten. Diese können dann jeweils effizienter gefunden werden, als wenn nach dem Anfrageobjekt im Ganzen gesucht werden würde.

Der erhöhte Speicherbedarf entsteht beim FRI durch die teilweise redundant gespeicherten Daten in den Indizes. Der Speicherbedarf wächst quadratisch mit der Anzahl der Attribute (er wächst asymptotisch mit $O(n \cdot m^2)$). Der FRI ist daher weniger geeignet, wenn die Datensätze viele Attribute haben.

Der Speicherbedarf war hier jedoch auch kein primäres Entwurfskriterium. Es sollte vor allem die Suchlaufzeit der Ähnlichkeitssuche im Vergleich zu dem einfachen Trie-Index für

Datensätze reduziert werden. Dies ist gelungen, auch wenn die Reduzierung mit einem Faktor von knapp 3 (bezüglich der durchschnittlichen Anfragezeit bei der Testmenge) nicht so deutlich ist wie erhofft. Die Anfragezeit vom Fri verhält sich dabei für die verschiedenen Datenbankgrößen ähnlich wie die beim einfachen Trie-Index (siehe asymptotische Analyse in Abschnitt 5.3.6 und experimentelle Untersuchung in Abschnitt 8.4.4) und ist damit auch bei größeren Datenbeständen effizient einsetzbar.

Die Berechnung der mindestens erwarteten Distanz nimmt einen großen Teil der Laufzeit einer Suchanfrage ein und bietet daher noch Raum für weitere Optimierungen. Bei der experimentellen Untersuchung habe ich festgestellt, dass die minestens erwartete Distanz in einigen Fällen nicht korrekt berechnet wird. Dieser Fehler sollte zunächst behoben werden. Außerdem werden noch nicht alle eigentlich verfügbaren Informationen zur Berechnung der mindestens erwarteten Distanz vollständig ausgenutzt. Der Fri ist vor allem darauf ausgelegt den ähnlichsten Datensatz schnell zu finden, jedoch sind auch Top-k-Anfragen und Bereichsanfragen möglich. Die Suchlaufzeit steigt bei großem k beziehungsweise großer Suchtoleranz jedoch relativ stark an (siehe Abschnitt 8.4.5). Ein weiterer Nachteil ist die schwer Vorhersagbare Anfragezeit beim Fri. Sie wurde asymptotisch bislang nur unter zwei vereinfachenden Annahmen bestimmt und auch in der Praxis schwankt sie in Abhängigkeit der Anzahl der gefundenen Werte bei den Indexsuchen (siehe Abschnitt 8.4.4). In zukünftigen Arbeiten ist eine Analyse der Laufzeit der Ähnlichkeitssuche für den ungünstigsten Fall (engl.: worst case) ohne die vereinfachenden Annahmen sowie eine Analyse des durchschnittlichen Falls bezüglich einer Wahrscheinlichkeitsverteilung der Wert (engl.: average case) sinnvoll.

Der Fri ist vor allem für Attributwerte entwickelt worden, die Zeichenketten sind. Er funktioniert jedoch prinzipiell auch für andere Typen von Attributen (Zahlen, komplexe Objekte), solange es eine Indexstruktur gibt, die in den Attributwerten eine Ähnlichkeitssuche durchführen können. Wenn sich die Datensätze jedoch so auf Vektoren abbilden lassen, dass der Abstand der Datensätze durch den Abstand der zugehörigen Vektoren bestimmt werden kann, eignen sich andere Indexstrukturen möglicherweise besser. Wenn der entstehende Vektorraum nur eine geringe Dimensionalität hat, gibt es eine Reihe von Indexstrukturen, um diesen Vektorraum zu indizieren (k-d-Bäume, R-Bäume, ...).

Im Folgenden wird noch die Idee einer Modifikation der Datenstruktur beim Fri vorgeschlagen, bei der nicht mehr alle Indizes verwendet werden, sondern einige Indizes weggelassen werden. Dies könnte am Beispiel der Adresskorrektur so aussehen, dass auf der ersten Ebene nur die Postleitzahl indiziert, in der zweiten Ebene zu jeder Postleitzahl die zugehörigen Orte und zu jedem Ort nur die zugehörigen Straßen. Diese Struktur würde nur ungefähr ein Drittel des Speicherplatzes benötigen und mit kleineren Anpassungen am Suchalgorithmus könnte dennoch eine Ähnlichkeitssuche durchgeführt werden. (Anschließend muss jedoch noch sichergestellt werden, dass auch alle Attribute zusammen in einem Datensatz vorkommen. Sonst könnte es beispielsweise sein, dass die Straße zwar in dem Ort, jedoch nicht in dem Postleitzahlgebiet vorhanden ist.) Bei dieser Vorgehensweise würde die Reihenfolge, in der die Attribute mit Werten belegt werden, also nicht mehr vom Algorithmus, sondern vom Benutzer der Indexstruktur festgelegt.

Außerdem könnte der Suchalgorithmus zur Suche des ähnlichsten Datensatzes dahingehend modifiziert werden, dass die Sortierung der Knoten nach der mindestens zu erwartenden Distanz etwas abgewandelt wird. Es könnten beispielsweise Knoten, bei denen schon viele Attribute mit einem Wert belegt sind, etwas bevorzugt werden. Diese würden dann ausgeführt werden, auch wenn sie *nicht* die insgesamt mindestens zu erwartende Distanz haben; sie führen jedoch möglicherweise deutlich schneller zu einem Ergebnis. Durch diese Modifikation würde die Garantie, dass der ähnlichste Datensatz gefunden wird, nicht mehr gelten, weil auch Knoten ausgeführt werden, die nicht die insgesamt mindestens erwartete Distanz haben.

Dafür wäre die Anfragezeit bei der Ähnlichkeitssuche wahrscheinlich deutlich geringer.[1] Es ist noch zu überprüfen, ob dies beispielsweise bei der Adresskorrektur eine geeignete Möglichkeit ist, um die Suchzeit weiter zu verringern.

Auch bei der momentanen Implementierung sollte noch (beispielsweise durch einen formalen Beweis) sichergestellt werden, dass die Formeln zur Berechnung der mindestens erwarteten Distanz korrekt sind, d. h. dass insbesondere nie eine zu hohe mindestens erwartete Distanz für einen Knoten berechnet wird.

Sinnvoll wäre außerdem die Integration weiterer Ähnlichkeitsmaße in den Fri, beziehungsweise die zu Grunde liegenden Trie-Datenstrukturen. Es wäre insbesondere wünschenswert schon im Fri mit der gewichteten Editierdistanz suchen zu können. Bei der Adresskorrektur könnte dann der Schritt der Sortierung der Ergebniskandidaten entfallen.

Die Fri-Datenstruktur ist prinzipiell auch in anderen Anwendungsbereichen einsetzbar und nicht auf die Adresskorrektur beschränkt. Er kann verwendet werden, wenn nach ähnlichen Datensätzen gesucht, also beispielsweise auch in Datenbanksystemen zur Suche von ähnlichen Tupeln. Er ist vor allem dann effizient, wenn die Datensätze nur wenige Attribute haben. Er eignet sich in erster Linie bei „Top-1-Anfragen“, also Ähnlichkeitssuchen, bei denen nur der ähnlichste Datensatz gefunden werden soll.

Es ist noch zu evaluieren, wie sich die Effizienz vom Fri bei Datensätzen verhält, die keine Adressen sind. Er eignet sich vermutlich vor allem dann, wenn es mindestens ein selten verändertes Attribut (wie die Postleitzahl bei der Adresskorrektur) gibt, weil die Suche dann in den meisten Fällen auf einen kleinen Ausschnitt der Datenbank begrenzt bleibt.

System zur Adresskorrektur

Neben dem Fri wurde in dieser Arbeit auch erfolgreich ein System zur Adresskorrektur entwickelt. Dieses erreicht die korrekte Zuordnung von gut 97 % der getesteten Adressen. Dies liegt unter anderem daran, dass geeignete Vorverarbeitungsschritte implementiert wurden, der Fri in den meisten Fällen passende Ergebniskandidaten liefert und diese mit Hilfe der gewichteten Editierdistanz so sortiert werden, dass die eigentlich korrekte Adresse an erster Stelle steht. Ein weiterer Grund für die hohe Zahl an Korrekturen ist jedoch auch, dass von vornherein viele der Adressen (oder zumindest Postleitzahlen) schon korrekt sind. Außerdem kann die Redundanz, die durch die drei Attribute Ort, Postleitzahl und Straße gegeben ist, ausgenutzt werden.

Es konnten jedoch insgesamt 2,5 % der Adressen nicht korrigiert werden. In Zukunft ist noch genauer zu evaluieren, warum für einige der Testadressen keine Ergebniskandidaten gefunden wurden. Es ist zu prüfen, ob dies beispielsweise durch die falsche Extraktion der Adressbestandteile oder eine zu hohe Suchlaufzeit im Fri begründet ist. Das Adresskorrektursystem sollte außerdem mit neuen, zum Trainingszeitpunkt nicht bekannten Adressen getestet werden. Weiterhin sollte noch untersucht werden, wie häufig die eigentlich korrekte Adresse nicht zugeordnet wurde, obwohl sie in der Menge der Ergebniskandidaten enthalten ist. Damit kann eine genauere Einschätzung der Qualität der Sortierverfahren erfolgen.

Beim Lernverfahren für Zeichenketten wurde ein relativ einfaches Verfahren gewählt, dass auf einigen vereinfachenden Annahmen beruht (Abschnitt 7.1.2). Die so trainierte gewichtete Editierdistanz lieferte im praktischen Einsatz brauchbare Ergebnisse. Es könnte jedoch noch untersucht werden, ob die Ergebnisse verbessert werden, wenn beispielsweise der Viterbi-Algorithmus zum Lernen der Wahrscheinlichkeiten der Editieroperationen verwendet wird.

[1] Bei der Entwicklung des Suchalgorithmus zeigte sich dieses Verhalten zufällig bei einer fehlerhaften Implementierung.

Wahrscheinlich ist es auch sinnvoll, das Lernverfahren nicht nur einmal initial durchzuführen, sondern auch im laufenden Betrieb (oder beispielsweise nachts). Dadurch könnte eine größere Datenbasis erhoben werden und die Gewichte der Editierdistanz würden auch an Veränderungen der OCR-Software angepasst werden.

Die Berechnung der Wahrscheinlichkeit der eigentlich korrekten Zeichenkette zu einer gegebenen Anfragezeichenkette geschieht hier mit der Methode von Bayes (Abschnitt 7.1.1). Dabei wäre es möglicherweise sinnvoll, die Berechnung nicht nur basieren auf der gelesenen OCR-Zeichenkette, sondern auch basierend auf weiteren Werten durchzuführen. Wenn die OCR-Software beispielsweise einen Konfidenzwert zur Verfügung stellt, mit der eine Zeichenkette (oder auch ein einzelner Buchstabe) korrekt erkannt wurde, könnte dieser Wert hier einfließen. Dadurch könnten Abweichungen bei Zeichenketten beziehungsweise Buchstaben geringer gewichtet werden, wenn diese nur mit einer niedrigen Konfidenz erkannt wurden.

Die A-Priori-Wahrscheinlichkeiten für Zeichenketten werden im Adresskorrektursystem bislang nicht verwendet, weil diesbezüglich erst sehr wenig Daten zur Verfügung stehen. Im täglichen Betrieb könnten jedoch für Orte und Straßen solche Auftretenswahrscheinlichkeiten bestimmt werden. Häufig vorkommende Orte würde dann beispielsweise eine höhere Wahrscheinlichkeit zugeordnet werden, die dann wiederum bei der Sortierung der Ergebniskandidaten verwendet werden könnte.

Die verwendeten Synonyme stammen bisher aus der gegebenen Datenbasis. Möglicherweise ist es sinnvoll auch Synonyme für Ortsnamen im laufenden Betrieb zu erlernen. Dies kann durch Ausnutzung der redundanten Information von Postleitzahl und Ort geschehen. Somit wäre es möglich, auch momentan nicht bekannte Synonymen für Ortsnamen mit der Zeit zu erlernen, falls sie in einer hinreichenden Anzahl von Adressen verwendet wurden.

Es wäre weiterhin interessant zu wissen, wie viele Adressen ohne die Zeitbegrenzung von 100 ms hätten bestimmt werden können.

Das Adresskorrektursystem wurde schon testweise auf der Briefsortiermaschine eingesetzt, muss jedoch noch im praktischen Betrieb erprobt werden. Dies ist für die nahe Zukunft geplant.

Das Verfahren ist jedoch nicht nur bei der Briefsortierung einsetzbar sondern auch in anderen Kontexten, beispielsweise zur Korrektur der eingegebenen Adresse bei einem Stadtplan im Internet. Mit kleineren Modifikationen wäre das Verfahren zur Adresskorrektur auch auf nicht-deutsche Adressen anwendbar.

A. Anhang

A.1. Notationen

In dieser Arbeit werden die folgenden Notationen und Bezeichnungen verwendet (siehe auch Abschnitt 2.1).

Mathematik

- $\mathbb{N}$ bezeichnet die natürlichen Zahlen.
- $\mathbb{Z}$ bezeichnet die ganzen Zahlen.
- $\mathbb{R}$ bezeichnet die reellen Zahlen.
- $i, j \in \mathbb{N}$ bezeichnen Laufvariablen.
- $\mathrm{P}(X)$ bezeichnet die Wahrscheinlichkeit für das Eintreten des Ereignisses X.
- $\mathrm{P}(X \mid Y)$ bezeichnet die bedingte Wahrscheinlichkeit für das Eintreten des Ereignisses X gegeben Y.
- $\langle v_1, v_2 \rangle$ bezeichnet das Skalarprodukt von v_1 und v_2.
- $\lceil x \rceil$ bezeichnet $\min \{y \in \mathbb{N} \mid y \geq x\}$.
- $O(\,\cdot\,)$ bezeichnet die Laufzeit im ungünstigsten Fall, wenn nicht anders angegeben (sogenannte Landau-Symbole).

Zeichenketten

- Σ bezeichnet ein endliches Alphabet.
- $\Sigma_\epsilon = \Sigma \cup \{\epsilon\}$ bezeichnet das Alphabet Σ zuzüglich der leeren Zeichenkette ϵ.
- Σ^* bezeichnet die Menge aller endlichen Wörter über dem Alphabet Σ.
- $\epsilon =$ „“ bezeichnet die leere Zeichenkette.
- $u, w \in \Sigma$ bezeichnen Buchstaben.
- $s, t \in \Sigma^*$ bezeichnen Zeichenketten.
- $q \in \Sigma^*$ bezeichnen eine Anfragezeichenkette.
- $s[i]$ bezeichnet den i-ten Buchstaben der Zeichenkette s.
- $|s|$ bezeichnet die Länge der Zeichenkette s.
- s^r bezeichnet die Zeichenkette s in umgekehrter Reihenfolge.
- $s \circ t$ bezeichnet die Konkatenation der Zeichenketten s und t.

Datenbank

- U bezeichnet die Menge aller möglichen Datensätze.
- $D \subset U$ bezeichnet eine Datenbank.
- $n = |D|$ bezeichnet die Anzahl der Datensätze in der Datenbank.
- A_i bezeichnet die Grundmenge des i-ten Attributs.
- m bezeichnet die Anzahl der Attribute eines Datensatzes.
- $q, r \in U$ bezeichnen Datensätze.
  (Im Kontext der metrischen Räume werden sie auch Objekte und Elemente genannt.)
- $r.a$ bezeichnet das Attribut a des Datensatzes r.
- $\delta : U \times U \to \mathbb{R}_0^+$ bezeichnet ein Abstandsmaß.
- $\sigma : U \times U \to \mathbb{R}_0^+$ bezeichnet ein Ähnlichkeitsmaß.
- $e \in \mathbb{R}_0^+$ bezeichnet eine Suchtoleranz.

Indexstrukturen

- I bezeichnet einen Index beim FRI.
- v bezeichnet einen Knoten im Entscheidungsbaum beim FRI.
- T bezeichnet eine Trainings- und Testmenge.

A.2. Abkürzungen

In dieser Arbeit wurden folgende Abkürzungen verwendet:

- CPU: Central Processing Unit
- DEA: Deterministischer Endlicher Automat
- DFA: Deterministic Finite Automaton
- DLL: Dynamic Link Library
- FRI: Fast Record Index
- OCR: Optical Character Recognition
- PLZ: Postleitzahl
- RAM: Random Access Memory
- RBF: Radial Basis Function
- SVM: Support Vector Machine
- TST: Ternary Search Trie

A.3. Datenbasis

In Abschnitt 6.1 wurde beschrieben, wie die Synonyme für Orts- und Straßennamen erzeugt werden. Dabei kommen folgende Listen mit Abkürzungen zum Einsatz.

Abkürzungen bei Ortsnamen

Abkürzung	Ausgeschrieben
a d	an dem, an den, auf der
v d	vor dem, vor der
a	am, an, auf
Altm	Altmark
Ammerl	Ammerland
Anh	Anhalt
b	bei
Bay	Bayern
Bergstr	Bergstraße
Bez	Bezirk
Bz	Bezirk
d	dem, der
Dillkr	Dillkreis
Dithm	Dithmarschen
Emsl	Emsland
Ennepe-Ruhrkr	Ennepe-Ruhrkreis
Enzkr	Enzkreis
Enzkr	Enz-Kreis
Erzgeb	Erzgebirge
Fränk	Fränkische
Friesl	Friesland
geb	gebirge
Gem	Gemeinde
Gm	Gemeinde
Grafsch	Grafschaft
Han	Hannover
Harlingerl	Harlinger Land
Hess	Hessen
Hl Blut	Heiligen Blut
Hochschwarz	Hochschwarzwald
Hochschwarzw	Hochschwarzwald
Hohenz	Hohenzollern
Hohenz	Hohenzollern
Holst	Holstein
Hotzenw	Hotzenwald
Hzgt	Herzogtum
i	im, in
id	in dem, in den, in der
Jeverl	Jeverland
Kr	Kreis
Kyffh	Kyffhäuser
l	land
Lauenb	Lauenburg
Markgräflerl	Markgräflerland
Meckl	Mecklenburg
Mittelfr	Mittelfranken
Mittelholst	Mittelholstein
NB	Niederbayern
Niederbay	Niederbayern
Niedersachs	Niedersachsen
NL	Niederlausitz
Nordb	Nordbahn
Nordfriesl	Nordfriesland
NWM	Nordwestmecklenburg
OB	Oberbayern
Oberbay	Oberbayern
Oberfr	Oberfanken
Oberpf	Oberpfalz
Oberwesterw	Oberwesterwald
Odenw	Odenwald
Odw	Odenwald

OFr.	Oberfranken
OL	Oberlausitz
Oldb	Oldenburg
OPf.	Oberpfalz
Osterzgeb	Osterzgebirge
Ostholst	Ostholstein
PLZ	Postleitzahl
Prign	Prignitz
Reinhardsw	Reinhardswald
Rheinhess	Rheinhessen
Rheinl	Rheinland
Sa.	Sachsen
Sachs	Sachsen
Sächs Saale	sächsiche Saale
Sächs Schweiz	Sächsische Schweiz
Sächs. Schw.	Sächsische Schweiz
Sachs-Anh	Sachsen-Anhalt
Sauerl	Sauerland
Schaumb	Schaumburg
Schaumb-Lippe	Schaumburg-Lippe
Schurw	Schurwald
Schwab	Schwaben
Schwäb	Schwäbisch
Schwarzw	Schwarzwald
Siegerl	Siegerland
Spreew	Spreewald
St	Sankt, Sanct
str	straße
Thür	Thüringen
Thüringer W	Thüringer Wald
Thüringerw	Thüringerwald
Ü	über
u	und
u.	und
Uckerm	Uckermark
UFr.	unterfranken
Unterfr	Unterfranken
v	vor, vorm
Vogtl	Vogtland
Vw.	Vorwerk
w	wald
Weserbergl	Weserbergland
Westerw	Westerwald
Westf	Westfalen
Württ	Württemberg

Abkürzungen bei Straßennamen

Abkürzung	Ausgeschrieben
1.	erste
2.	zweite
3.	dritte
4.	vierte
Aussiedl.	Aussiedlung
Bgm.	Bürgermeister
Dr.	Doktor
Ev.	Evangelisch
Fr.	Friedrich
Hl.	Heiliges
Joh.	Johann
Kol.	Kolonie
Nr.	Nummer
Pfr.	Pfarrer
Prof.	Professor
Siedl.	Siedlung
St.	Sankt, Sanct
Wilh.	Wilhelm
a. d.	an den
a.	am
b.	bei

d.	der
i.	im
u.	und
v.	von
str.	straße
Str.	Straße

A.4. Testdaten

Dieser Abschnitt enthält die Listen nicht Adressen, die auch manuell nicht korrigiert werden konnten oder nicht richtig in die Bestandteile zerlegt wurden. Um das Postgeheimnis zu wahren sind die Buchstaben der Namen der Empfänger sowie die Hausnummer jeweils durch „?“ ersetzt.

Bei der Zuordnung der korrekten Adressen zu den mittels OCR eingelesenen Adressen konnten die folgenden 21 Adressen auch manuell nicht korrigiert werden, beispielsweise weil die Adresse unbekannt ist (siehe Abschnitt 8.2).

- Herm ?????? ???? / Hoffmannstraße ?? / 53113 Bonn
- ???????? ????? / ???????? / Schatthauser Str. ? / 21629 Horneburg
- ???????????? Zossen / Hauptallee ???/? / 15838 Wünsdorf OT Waldstadt
- ???????? ?????????????? / ombH / Immsolltr. ???-?? / 10553 Berlin
- ???-???????? ???????? / ???? ????. ??. ????-????? ??????? / Schloß Reichartshausen / 65375 Oestrich-Winkel
- ???????? ??????? ?? / ????? ????? / Kölner Weg ? – ?? / 50321 Brühl
- ??????????? / Berlin-Schöneberg / Grundbuch3mt / 10820 Berlin
- .????????? ?????? / Liel)kriuchtstr. ?? / 16548 Glicnicke
- An die / ?????????? ???????????? AG / Direktion / Kredit-Betrieb / Herrn ????????? ????ö??? / 65793 Wiesbaden
- ????? ???? / ?????????? ????????? ????????? / ?reudentaler Strasse ? / 78465 Konstanz
- ??????? ??????? / Dorfstr. ? / 03222 Groß Beuchow
- ???????? ??????? ?? / ???????? ??? ???????????? / Herrn ??????? ??????? / Kölner Weg ? – ?? / 50321 Brühl
- ????????????????? ???????? / Frau ???? / Postfach / 68149 Mannheim
- ???? ????????????????? ?????? ??? / c/o ???? ?? ?? ?? / z.H. Frau ???????? / Rewestr. ? / 01683 Ketzerbachtal
- Herrn / ?????? ?????????? / Straße der Jugend ?? / 37412 Herzberg
- ????????????????? ?????? / Abteilung 2 / Geschäftsstelle 79 / ???? ????????? o.V.i.A. / 10548 Berlin -
- Frau ???????????? / ????? ?????? / ????????? ????? / ????????-??????? / Gebäude Vill/5 / 86836 Lagerlechfeld
- Herrn ?????????????? / ?????? ??????? / 4./???????????????? 21 / Am Alsberg ?? / 56477 Rennerod
- Herrn / ?????? ??????? / ???? / Lyoner Str. ?? / 60528 Berlin
- Herrn Professor / Dr. ???? ??????? / ???????? / Institut für ????????? ????????? / Lutherstr. ? / 17487 Greifswald
- Frau / ???????????????? ??? / ??? ??????-???? / ?-??? ????????? / Hauptstr. ?? / 56814 Ernst

Die folgenden 14 Adressen wurden vom Algorithmus zur Extraktion der Adressbestandteile nicht korrekt in die Bestandteile zerlegt (siehe Abschnitt 6.2.1 und Abschnitt 8.5).

- ???? ???????????-?? ??? / ??? GmbH / Tschaikowskistraße ?? / 04105 Leipzig / 43070 Bearbeiter: jahn Tel,
- ??? GmbH & Co. KG / Am Mühlgraben ? / Großkoschen / 01968 Senftenberg
- Frau / ???? ?????? / Schulstr. ?? / OT Mattendorf / 03149 Wiesengrund
- Herrn / ???????? ?????? / Mühlenweg ? / OT Viez / 19230 Hagenow
- Flerm / ??????? ???-???????? / Helene-Mayer-Ring ?? / App, 568 / 80809 München
- Herm / ??????? ???? / Nr. ?? / 01825 Bömersdorf-Breitenau
- ?????? ?????? / Drakestr. ?? / 12205 BerlinerAnwaitsvereins
- Der Justizbote / GmbH / Frau ???????? / ?? / 10553 gerl(n
- ??????? ??????? / Manfred-von-Richthofen- / Str. ? / 12101 Berlin-Tempelhof
- Herm ???????? ???????? / Hallerstraße ?-? / 4. Obergeschoss / 10587 Berlin
- Amtsgericht Strausberg / Grundbuchamt / Klosterstr. ?? / ???, / 15344 Strausberg L, -:z:==-
- Amtsgericht Charlottenburg / Handelsregister / Amtsgerichtsplatz ? sarnd / !)ehörde / 14057 Berlin
- AtehrINlkmmfen / VaÂ<Mtz12 / 1178Berin / 1681 / Mte*btanum
- ..???????und????????????? / Je.neimer Str. ?? / 14197 BerlinerAnwaltsvereins

Die folgenden 29 Adressen wurden bei der Adresskorrektur falsch korrigiert. In der ersten Zeile ist jeweils die OCR-Adresse, in der zweiten die vom Algorithmus bestimmte und in der dritten die eigentlich korrekte Adresse vermerkt.

ort	*plz*	*str*
Senftenberg	01968	Großkoschen
Senftenberg	01968	Grünstr.
Großkoschen	01968	Am Mühlgraben
Altenkrchen	57610	Auf'm Eichhahn
Altenkirchen	57610	Auf dem Eichelchen
Almersbach	57610	Auf'm Eichhahn
Köhlen	27624	Dorfstr.- Ost Nr.
Köhlen	27624	Feldstr.
Köhlen	27624	Dorfstraße-Ost
Idar-Oberstein	55743	Klefernweg
Idar-Oberstein	55743	Fliederweg
Idar-Oberstein	55743	Kiefernstr.
Hagenow	19230	Ot Viez
Hagenow	19230	Irisweg
Hagenow	19230	Lilienweg
Lohmar	53797	Postfach
Lohmar	53790	Postfach
Lohmar	53787	Postfach
Stuttgart	70549	Postfach
Stuttgart	70509	Postfach
Stuttgart	70049	Postfach
Gr-immen	18507	R.Schumannstr.
Grimmen	18507	Schulstr.
Grimmen	18507	Robert-Schumann-Str.
Wiesengrund	03149	OT Mattendorf
Wiesengrund	03149	Mattendorfer Str.
Wiesengrund	03149	Schulstr.
Netistrelitz	17235	,X.-Frie(Irich-Stl'.
Neustrelitz	17235	Kranichstr.
Neustrelitz	17235	Adolf-Friedrich-Str.
Berlin	10119	Alte Schönhauser Allee
Berlin	10119	Schönhauser Allee
Berlin	10119	Alte Schönhauser Str.
KilIcildort	88630	13iichstr.
Pfullendorf , Baden	88630	Ulrichstr.
Pfullendorf	88630	Bachstr.

Beesko„-Kohls(lorf	15848	Dorlslr.
Bornow	15848	Dorfstr.
Beeskow	15848	Dorfstr.
Fredersdorf	15370	Fckistraße
Fredersdorf	15370	Fließstr.
Fredersdorf	15370	Feldstr.
Strausbug	15344	Gtistav-Ktirtze-I>i-omciia(le
Strausberg	15344	Kirschallee
Strausberg	15344	Gustav-Kurtze-Promenade
Stratist)erg	15344	(j∼istav-Kiirtze-iromenade
Strausberg	15344	Fließstr.
Strausberg	15344	Gustav-Kurtze-Promenade
Berlin	12159	fir4ndierystraße
Berlin	12159	Kundrystr.
Berlin	12159	Handjerystr.
S(1 i(iriow	16321	Iciiirich-i Wine-Straße
Bernau	16321	Heinestr.
Schönow	16321	Heinrich-Heine-Str.
Schbi iciche/Berlin	15566	‚jägerstr.
Berlin	12526	Jägerstr.
Schöneiche	15566	Jägerstr.
Strausberg L, -:z:==-	15344	317,
Strausberg	15344	Markt
Strausberg	15344	Klosterstr.
Fetii)zich	70736	1,in(Icril)(’ililweg
Fellbach	70736	Geibelweg
Fellbach	70736	Lindenbühlweg
Berfin	14197	Lu(i,wig-Baniav-Platz’-)
Berlin	14197	Lilienweg
Berlin	14197	Ludwig-Barnay-Platz
Berlin-Tempelhof	12101	Str.
Berlin	12105	Str. 7
Berlin	12101	Manfred-von-Richthofen-Str.
Zeiitlien	15738	N(’iriü)(‚rger Str.
Zeuthen	15738	Bremer Str.
Zeuthen	15738	Nürnberger Str.
Springe	31832	Postfach
Springe	31821	Postfach
Springe	31813	Postfach
Ostfil(IL,rii	73760	1∼icarda-1 liicii-Str
Ostfildern	73760	Ulrichstr.
Ostfildern	73760	Ricarda-Huch-Str.
Oranienbun;	16515	liiing(-str. IOC
Oranienburg	16515	Ringstr.
Oranienburg	16515	Rungestr.
Gicss v	35394	Tiirinei iweg
Gießen	35394	Fliednerweg
Gießen	35394	Tannenweg

A.5. Editieroperationen

In Abschnitt 7.1 wurde beschrieben, wie die Wahrscheinlichkeiten der Editieroperationen erlernt werden können. Die Ergebnisse sind in folgender Tabelle dargestellt. Die Auflistung umfasst jedoch nur alle erweiterten Editieroperationen (Aufspaltungen und Verschmelzungen), die in den Trainingsdaten mindestens zwei mal beobachtet wurden. Dazu ist jeweils die Anzahl und die davon ausgehend berechnete Wahrscheinlichkeit der Editieroperation angegeben.

Operation	Wahrscheinlichkeit	Anzahl
„ “ → „St“	0.002404	2
„ D“ → „d“	0.068966	2
„ S“ → „s“	0.063291	25
„-S“ → „s“	0.022222	4
„.“ → „aB“	0.001816	3
„.“ → „al“	0.001816	3
„.“ → „aß“	0.002421	4
„B“ → „li“	0.000948	2

„F“ → „l’“	0.013423	2
„H“ → „fl“	0.009804	3
„S“ → „r.“	0.002312	2
„W“ → „Ho“	0.008850	2
„W“ → „VV“	0.013274	3
„a“ → „ ,i“	0.001528	3
„a“ → „-A“	0.001019	2
„a“ → „ci“	0.001019	2
„a“ → „e “	0.001019	2
„a“ → „ii“	0.003057	6
„b“ → „)(“	0.003578	2
„b“ → „l)“	0.003578	2
„ch“ → „z“	0.003623	2
„d“ → „(I“	0.007667	7
„d“ → „(]“	0.002191	2
„d“ → „(l“	0.008762	8
„d“ → „cl“	0.002191	2
„e“ → „(,“	0.000827	5
„e“ → „ ,J“	0.000331	2
„e“ → „ci“	0.000827	5
„e“ → „cr“	0.000331	2
„e“ → „ii“	0.000331	2
„eg“ → „f“	0.010929	2
„en“ → „m“	0.002445	2
„er“ → „.“	0.000650	2
„er“ → „l“	0.000650	2
„er“ → „u“	0.000650	2
„h“ → „ii“	0.004831	5
„h“ → „li“	0.013527	14
„ig“ → „∼“	0.032258	2
„in“ → „-“	0.000878	2
„le“ → „i“	0.005510	2
„li“ → „b“	0.001066	2
„li“ → „h“	0.018657	35
„li“ → „n“	0.003731	7
„li“ → „ä“	0.003731	7
„li“ → „ü“	0.002665	5
„ll“ → „H“	0.009709	3
„ll“ → „e“	0.012945	4
„m“ → „ M“	0.002890	2
„m“ → „ii“	0.005780	4
„m“ → „in“	0.005780	4
„m“ → „ni“	0.004335	3
„m“ → „ri“	0.002890	2
„m“ → „rn“	0.007225	5
„n“ → „ii“	0.004584	19
„n“ → „il“	0.001206	5
„n“ → „ri“	0.004101	17
„n“ → „rl“	0.001206	5
„n“ → „ti“	0.000724	3
„on“ → „.“	0.029412	2
„r“ → „aß“	0.000330	2
„r“ → „i-“	0.001484	9
„r“ → „ni“	0.000659	4
„r.“ → „n“	0.001212	2
„rf“ → „d“	0.051852	7
„ri“ → „d“	0.018519	4
„ri“ → „h“	0.013889	3
„rl“ → „d“	0.037940	70
„rl“ → „h“	0.001626	3
„rl“ → „t“	0.001626	3
„rn“ → „m“	0.359223	37
„s“ → „ S“	0.006115	15
„t“ → „l(“	0.000605	2
„u“ → „ii“	0.009534	9
„u“ → „ti“	0.021186	20
„w“ → „\,“	0.009217	2
„w“ → „\v“	0.009217	2
„w“ → „vv“	0.009217	2
„ß“ → „ss“	0.170213	8

A.6. Ergebnis der Adresskorrektur

In Abschnitt 8.5 wurden die Statistiken der Ergebnisse der Adresskorrektur für die Testmenge T_{alle} angegeben. Im Folgenden werden diese Daten für jeweils für die Testpost T_{test} und die Echtpost T_{echt} aufgeschlüsselt.

Ergebnis der Adresskorrektur bezüglich der Testpost:

	OCR		Adresskorrektur		Adr. + Verw.	
Adresse richtig	1.713	*61,7 %*	2.701	*97,3 %*	2.701	*97,3 %*
Adresse falsch	1.063	*38,3 %*	20	*0,7 %*	75	*2,7 %*
Verworfen	—		55	*2,0 %*	—	
PLZ richtig	2.713	*97,7 %*	2.717	*97,9 %*	2.767	*99,7 %*
PLZ falsch	63	*2,3 %*	4	*0,1 %*	9	*0,3 %*
Verworfen	—		55	*2,0 %*	—	
LR richtig	2.739	*98,7 %*	2.720	*98,0 %*	2.772	*99,9 %*
LR falsch	37	*1,3 %*	1	*0,0 %*	4	*0,1 %*
Verworfen	—		55	*2,0 %*	—	

Ergebnis der Adresskorrektur bezüglich der Echtpost:

	OCR		Adresskorrektur		Adr. + Verw.	
Adresse richtig	642	*60,7 %*	1.039	*98,2 %*	1.039	*98,2 %*
Adresse falsch	416	*39,3 %*	9	*0,9 %*	19	*1,8 %*
Verworfen	—		10	*0,9 %*	—	
PLZ richtig	1.025	*96,9 %*	1.046	*98,9 %*	1.053	*99,5 %*
PLZ falsch	33	*3,1 %*	2	*0,2 %*	5	*0,5 %*
Verworfen	—		10	*0,9 %*	—	
LR richtig	1.051	*99,3 %*	1.048	*99,1 %*	1.055	*99,7 %*
LR falsch	7	*0,7 %*	0	*0,0 %*	3	*0,3 %*
Verworfen	—		10	*0,9 %*	—	

Trivia

Die höchste in Deutschland vergebene Postleitzahl ist „99998“, es gibt einen Ort namens „Heide 1 und 2“ sowie die Straßennamen „Üülle Browäi“ und „Venedig“.

Literaturverzeichnis

[Aoe89] Aoe, J.I.: An efficient digital search algorithm by using a double-array structure. In: *IEEE Transactions on Software Engineering* 15 (1989), Nr. 9, S. 1066–1077. – http://portal.acm.org/citation.cfm?id=75622

[AS07] Askitis, N. ; Sinha, R.: HAT-trie: a cache-conscious trie-based data structure for strings. In: *Proceedings of the 30th Australasian conference on Computer science* 62 (2007), S. 97–105. – http://portal.acm.org/citation.cfm?id=1273749.1273761

[BCP02] Bartolini, I. ; Ciaccia, P. ; Patella, M.: String Matching with Metric Trees Using an Approximate Distance. In: *Proceedings of the 9th International Symposium on String Processing and Information Retrieval* (2002), S. 271–283. – http://www-db.deis.unibo.it/research/papers/SPIRE02.pdf

[BHS07] Bocek, T. ; Hunt, E. ; Stiller, B.: Fast Similarity Search in Large Dictionaries / Department of Informatics, University of Zurich. 2007 (ifi-2007.02). – Forschungsbericht. – http://fastss.csg.uzh.ch/ifi-2007.02.pdf

[BK73] Burkhard, W.A. ; Keller, R.M.: Some approaches to best-match file searching. In: *Communications of the ACM* 16 (1973), Nr. 4, S. 230–236. – http://portal.acm.org/citation.cfm?id=362025

[BM72] Bayer, R. ; McCreight, EM: Organization and maintenance of large ordered indexes. In: *Acta Informatica* 1 (1972), Nr. 3, S. 173–189. – http://portal.acm.org/citation.cfm?id=944331.944347

[BM77] Boyer, R.S. ; Moore, J.S.: A fast string searching algorithm. In: *Communications of the ACM* 20 (1977), Nr. 10, S. 762–772

[BM00] Brill, E. ; Moore, R.C.: An improved error model for noisy channel spelling correction. In: *Proceedings of the 38th Annual Meeting on Association for Computational Linguistics* (2000), S. 286–293. – http://www.aclweb.org/anthology/P00-1037

[BM02] Bilenko, M. ; Mooney, R.J.: Learning to combine trained distance metrics for duplicate detection in databases. In: *Submitted to CIKM-2002* (2002). – http://www.cs.utexas.edu/~ml/papers/marlin-tr-02.pdf

[BM03] Bilenko, M. ; Mooney, R.J.: Adaptive duplicate detection using learnable string similarity measures. In: *Proceedings of the 9th ACM SIGKDD international conference on Knowledge discovery and data mining* (2003), S. 39–48. – http://research.microsoft.com/~mbilenko/papers/03-marlin-kdd.pdf

[BM05] Bruce, B.F. ; Macia, C.: *Method and system for name and address validation and correction*. 2005 http://www.freepatentsonline.com/20050137991.html

[BO97] Bozkaya, T. ; Ozsoyoglu, M.: Distance-based indexing for high-dimensional metric spaces. In: *Proceedings of the 1997 ACM SIGMOD international conference on Management of data* (1997), S. 357–368. – http://portal.acm.org/citation.cfm?id=253345

[Boo08] Boost: *C++ Libraries*. 2008. – http://www.boost.org/

[BQ06] Baker, C.A. ; Quine, D.B.: *Method for enhancing mail piece processing system*. 2006 http://www.freepatentsonline.com/7258277.html

[Bra05] Branting, L.K.: Name Matching in Law Enforcement and Counter-Terrorism. In: *ICAIL 2005 Workshop on Data Mining, Information Extraction, and Evidentiary Reasoning for Law Enforcement and Counter-Terrorism* (2005). – http://www.karlbranting.net/papers/icail2005.pdf

[Bri59] Briandais, R. de l.: File searching using variable length keys. In: *Proceedings of the Western Joint Computer Conference* 15 (1959), S. 285–298

[BS97] Bentley, J.L. ; Sedgewick, B.: Fast algorithms for sorting and searching strings. In: *Proceedings of the 8th annual ACM-SIAM symposium on Discrete algorithms* (1997), S. 360–369. – http://portal.acm.org/citation.cfm?id=314161.314321

[BS98] Bentley, J. ; Sedgewick, B.: *Ternary Search Trees*. 1998. – http://www.ddj.com/windows/184410528

[Bud91] Budzinsky, C.D.: Automated Spelling Correction. In: *Statistics Canada* (1991)

[Bur98] Burges, C.J.C.: A Tutorial on Support Vector Machines for Pattern Recognition. In: *Data Mining and Knowledge Discovery* 2 (1998), Nr. 2, S. 121–167. – http://research.microsoft.com/~cburges/papers/SVMTutorial.pdf

[BYCMW94] Baeza-Yates, R. ; Cunto, W. ; Manber, U. ; Wu, S.: Proximity matching using fixed-queries trees. In: *Proceedings of the 5th Annual Symposium on Combinatorial Pattern Matching* 807 (1994), S. 198–212. – ftp://sunsite.dcc.uchile.cl/pub/users/rbaeza/papers/fqtrees.ps.gz

[BYG92] Baeza-Yates, R. ; Gonnet, G.H.: A new approach to text searching. In: *Communications of the ACM* 35 (1992), Nr. 10, S. 74–82

[BYN97] Baeza-Yates, R. ; Navarro, G.: A practical index for text retrieval allowing errors. In: *Proceedings of the 23rd Latin American Conference on Informatics (CLEI'97), Valparaiso, Chile* (1997). – http://www.dcc.uchile.cl/~gnavarro/ps/clei97.ps.gz

[BYN98] Baeza-Yates, R. ; Navarro, G.: Fast approximate string matching in a dictionary. In: *Proceedings of the South American Symposium on String Processing and Information Retrieval* (1998), S. 14–22. – http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=712978

[CB04] Cucerzan, S. ; Brill, E.: Spelling correction as an iterative process that exploits the collective knowledge of web users. In: *Proceedings of EMNLP* 4 (2004), S. 293–300. – http://acl.ldc.upenn.edu/acl2004/emnlp/pdf/Cucerzan.pdf

[CCGK07] Chaudhuri, S. ; Chen, B.C. ; Ganti, V. ; Kaushik, R.: Example-driven Design of Efficient Record Matching Queries. In: *VLDB '07: Proceedings of the 33rd international conference on Very large data bases*, VLDB Endowment, 2007. – ISBN 978–1–59593–649–3, S. 327–338. – http://research.microsoft.com/users/skaushi/vldb07-eg.pdf

[Cha06] Chapman, Sam: *String Similarity Metrics for Information Integration*. 2006. – http://www.dcs.shef.ac.uk/~sam/stringmetrics.html

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

[Cle01] Clement, J.: Dynamical Sources in Information Theory: A General Analysis of Trie Structures. In: *Algorithmica* 29 (2001), Nr. 1, S. 307–369. – http://algo.inria.fr/flajolet/Publications/RR3645.ps

[CN02] Chávez, E. ; Navarro, G.: A metric index for approximate string matching. In: *Proceedings of the 5th Latin American Symposium on Theoretical Informatics* 2286 (2002), S. 181–195. – http://garota.fismat.umich.mx/~elchavez/publica/metind.ps.gz

[Coh06] Cohen, W.W.: *SecondString - a Java toolkit for developing and evaluating approximate string comparision operators*. 2006. – http://secondstring.sourceforge.net/

[CRF03] Cohen, W.W. ; Ravikumar, P. ; Fienberg, S.E.: A comparison of string distance metrics for name-matching tasks. In: *Proceedings of the IJCAI-2003 Workshop on Information Integration on the Web* (2003). – http://www.isi.edu/info-agents/workshops/ijcai03/papers/Cohen-p.pdf

[CRPZ97] Ciaccia, Paolo ; Rabitti, Fausto ; Patella, Marco ; Zezula, Pavel: M-tree: An efficient access method for similarity search in metric spaces. In: *Proceedings of the 23rd International Conference on Very Large Data Bases* (1997), S. 426–435. – http://www-db.deis.unibo.it/research/papers/VLDB97.pdf

[CRZP02] Ciaccia, Paolo ; Rabitti, Fausto ; Zezula, Pavel ; Patella, Marco: *The M-Tree project*. 2002. – http://www-db.deis.unibo.it/Mtree/

[Dam64] Damerau, F.J.: A technique for computer detection and correction of spelling errors. In: *Communications of the ACM* 7 (1964), Nr. 3, S. 171–176. – http://portal.acm.org/citation.cfm?id=363994

[Deu08] Deutsche Post Direkt GmbH: *Datafactory Streetcode*. 2008. – http://www.deutschepost.de/dpag?xmlFile=link1015590_3877

[DM81] Dunlavey, M.R. ; Miller, L.A.: Technical corrections: On spelling correction and beyond. In: *Communications of the ACM* 24 (1981), Nr. 9, S. 608–609. – http://portal.acm.org/citation.cfm?id=383429

[DS07] Dalvi, N. ; Suciu, D.: Efficient query evaluation on probabilistic databases. In: *The VLDB Journal The International Journal on Very Large Data Bases* 16 (2007), Nr. 4, S. 523–544. – http://www.vldb.org/conf/2004/RS22P1.PDF

[Dun46] Dunn, H.L.: Record Linkage. In: *American Journal of Public Health and the Nations Health* 36 (1946), Nr. 12, S. 1412

[Ekm07] Ekman, Rasmus: *Structured Associative Containers*. 2007. – http://abc.se/~re/code/tst/tst_docs/

[FG99] Ferragina, P. ; Grossi, R.: The string B-tree: a new data structure for string search in external memory and its applications. In: *Journal of the ACM (JACM)* 46 (1999), Nr. 2, S. 236–280. – http://portal.acm.org/citation.cfm?id=301973

[Fre60] Fredkin, E.: Trie memory. In: *Communications of the ACM* 3 (1960), Nr. 9, S. 490–499

[FS69] Fellegi, I.P. ; Sunter, A.B.: A Theory for Record Linkage. In: *Journal of the American Statistical Association* 64 (1969), Nr. 328, S. 1183–1210. – http://www.jstor.org/view/01621459/di985900/98p0492d/0

[GIJ+01] Gravano, L. ; Ipeirotis, P.G. ; Jagadish, H.V. ; Koudas, N. ; Muthukrishnan, S. ; Pietarinen, L. ; Srivastava, D.: Using q-grams in a DBMS for Approximate String Processing. In: *IEEE Data Engineering Bulletin* 24 (2001), Nr. 4, S. 28–34. – http://www.cs.columbia.edu/~pirot/publications/deb-dec2001.pdf

[HA03] Hodge, V.J. ; Austin, J.: A Comparison of Standard Spell Checking Algorithms and a Novel Binary Neural Approach. In: *IEEE Transactions on knowledge and data engineering* 15 (2003), Nr. 5, S. 1073. – http://eprints.whiterose.ac.uk/689/1/hodgevj2.pdf

[Ham50] Hamming, R.W.: Error detecting and error correcting codes. In: *Bell Syst. Tech. J* 29 (1950), Nr. 2, S. 147–160. – http://engelschall.com/~sb/hamming/

[HCL03] Hsu, C.W. ; Chang, C.C. ; Lin, C.J.: A practical guide to support vector classification. In: *National Taiwan University, Tech. Rep., July* (2003). – http://www.csie.ntu.edu.tw/~cjlin/papers/guide/guide.pdf

[HHR97] Hemaspaandra, E. ; Hemaspaandra, L.A. ; Rothe, J.: Raising NP lower bounds to parallel NP lower bounds. In: *ACM SIGACT News* 28 (1997), Nr. 2, S. 2–13. – http://portal.acm.org/citation.cfm?id=261344

[Hop05] Hoppe, Manuel: *OpenGeoDB – freie Geokoordinaten-Datenbank*. 2005. – http://opengeodb.sourceforge.net/

[HS03] Hjaltason, G.R. ; Samet, H.: Index-driven similarity search in metric spaces. In: *ACM Transactions on Database Systems (TODS)* 28 (2003), Nr. 4, S. 517–580. – http://portal.acm.org/citation.cfm?id=958948

[Hyv04] Hyvönen, J.: *Scalability Matters - Multiple Column Fuzzy Matching with Sub-linear Scalability*. 2004 http://www.syslore.com/download.php?file=Scalability_Matters___mCorrection_technology_White_Paper.pdf

[Hyv05] Hyvönen, J.: *Searching for Symbol String*. 13. Juli 2005 http://www.freepatentsonline.com/EP1552429.html

[HZW02] Heinz, S. ; Zobel, J. ; Williams, H.E.: Burst tries: a fast, efficient data structure for string keys. In: *ACM Transactions on Information Systems (TOIS)* 20 (2002), Nr. 2, S. 192–223. – http://goanna.cs.rmit.edu.au/~jz/fulltext/acmtois02.pdf

[Jar89] Jaro, M.A.: Advances in Record-Linkage Methodology as Applied to Matching the 1985 Census of Tampa, Florida. In: *Journal of the American Statistical Association* 84 (1989), Nr. 406. – http://www.fcsm.gov/working-papers/mjaro.pdf

[Jar95] Jaro, M.A.: Probabilistic linkage of large public health data files. In: *Statistics in medicine* 14 (1995), Nr. 5-7, S. 491–498

[JHZ02] Jin, Rong ; Haupmann, Alex G. ; Zhai, Cheng X.: A content-based probabilistic correction model for OCR document retrieval. In: *The 25th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval Workshop Program (SIGIR 2002), Tampere, Finland, August* (2002), S. 11–15. –
http://citeseer.ist.psu.edu/jin02contentbased.html

[KCG90] Kernighan, M.D. ; Church, K.W. ; Gale, W.A.: A Spelling Correction Program Based on a Noisy Channel Model. In: *Proceedings of COLING-90* (1990), S. 205–2. –
http://acl.ldc.upenn.edu/C/C90/C90-2036.pdf

[KM83] Kalantari, I. ; McDonald, G.: Data structure and an algorithm for the nearest point problem. In: *IEEE Transactions on Software Engineering* 9 (1983), Nr. 5, S. 631–634. –
http://portal.acm.org/citation.cfm?id=1313699

[KMJP77] Knuth, D.E. ; Morris Jr, J.H. ; Pratt, V.R.: Fast Pattern Matching in Strings. In: *SIAM Journal on Computing* 6 (1977), S. 323

[Knu73] Knuth, D.E.: The art of computer programming. Vol. 3: Sorting and searching. In: *Reading* (1973)

[KR87] Karp, R.M. ; Rabin, M.O.: Efficient randomized pattern-matching algorithms. In: *IBM Journal of Research and Development* 31 (1987), Nr. 2, S. 249–260

[KR02] Kolak, O. ; Resnik, P.: OCR error correction using a noisy channel model. In: *Proceedings of the second international conference on Human Language Technology Research* (2002), S. 257–262. –
http://www.cs.umd.edu/users/okan/publications/hlt02.pdf

[Kuk92] Kukich, K.: Technique for automatically correcting words in text. In: *ACM Computing Surveys (CSUR)* 24 (1992), Nr. 4, S. 377–439. –
http://portal.acm.org/citation.cfm?id=146380&dl=GUIDE,

[KWLL05] Kim, M.S. ; Whang, K.Y. ; Lee, J.G. ; Lee, M.J.: n-gram/2L: a space and time efficient two-level n-gram inverted index structure. In: *Proceedings of the 31st international conference on Very large data bases* (2005), S. 325–336. –
http://www.vldb2005.org/program/paper/wed/p325-kim.pdf

[Lev66] Levenshtein, V.I.: Binary Codes Capable of Correcting Deletions, Insertions and Reversals. In: *Soviet Physics Doklady* 10 (1966), S. 707. –
http://adsabs.harvard.edu/abs/1966SPhD...10..707L

[LH03] Lasko, T.A. ; Hauser, S.E.: Approximate string matching algorithms for limited-vocabulary OCR output correction. In: *Proceedings of SPIE* 4307 (2003), S. 232. –
http://wwwetb.nlm.nih.gov/lhc/docs/published/2001/pub2001012.pdf

[LMRS07] Litwin, W. ; Mokadem, R. ; Rigaux, P. ; Schwarz, T.: Fast nGram-Based String Search Over Data Encoded Using Algebraic Signatures. In: *VLDB*, 2007, S. 207–218. –
http://www.lamsade.dauphine.fr/FILES/publi708.pdf

[LW75] Lowrance, Roy ; Wagner, R.A.: An Extension of the String-to-String Correction Problem. In: *Journal of the Association for Computing Machinery* 22 (1975), Nr. 2, S. 177–183. –
http://dalcin.org/referencias/pdf/697.pdf

[LWY07] Li, Chen ; Wang, Bin ; Yang, Xiaochun: VGRAM - Improving Performance of Approximate Queries on String Collections Using Variable-Length Grams. In: *VLDB*, 2007, S. 303–314. –
http://www.vldb.org/conf/2007/papers/research/p303-li.pdf

[Man07] Manning, C.: *Introduction to information retrieval.* 2007

[MC75] Morris, R. ; Cherry, L.L.: Computer Detection of Typographical Errors. In: *IEEE Transactions on Professional Communication* 18 (1975), März, Nr. 1, S. 54–64

[McI82] McIlroy, M.: Development of a Spelling List. In: *IEEE Transactions on Communications* 30 (1982), Nr. 1, S. 91–99

[MF82] Mor, M. ; Fraenkel, AS: A hash code method for detecting and correcting spelling errors. In: *Communications of the ACM* 25 (1982), Nr. 12, S. 935–938. –
http://portal.acm.org/citation.cfm?id=358728.358752

[Mic08] Microsoft Corporation: *Windows Management Instrumentation.* 2008. –
http://msdn.microsoft.com/en-us/library/aa394531(VS.85).aspx

[Mor68] Morrison, Donald R.: PATRICIA - Practical Algorithm To Retrieve Information Coded in Alphanumeric. In: *Journal of the ACM* 15 (1968), Nr. 4, S. 514–534. – ISSN 0004–5411. –
http://portal.acm.org/citation.cfm?id=321481

[MS04] Mihov, S. ; Schulz, K.U.: Fast approximate search in large dictionaries. In: *Computational Linguistics* 30 (2004), Nr. 4, S. 451–477. –
http://www.lml.bas.bg/~stoyan/J04-4003.pdf

[MSP05] Mois, P. ; Sepulveda, M. ; Proschle, H.: Street Address Correction Based on Spelling Techniques. In: *Proceedings of the 22nd British National Conference on Databases, Bncod 22, Sunderland, UK, July 5-7, 2005* (2005), S. 166–172. –
http://books.google.com/books?id=hOkjzcilSlkC&pg=PA166

[MWK+06] Mierswa, I. ; Wurst, M. ; Klinkenberg, R. ; Scholz, M. ; Euler, T.: YALE: rapid prototyping for complex data mining tasks. In: *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining* (2006), S. 935–940

[Nav01] Navarro, G.: A guided tour to approximate string matching. In: *ACM Computing Surveys (CSUR)* 33 (2001), Nr. 1, S. 31–88. –
http://www.dcc.uchile.cl/~gnavarro/ps/acmcs01.1.ps.gz

[NBY00] Navarro, G. ; Baeza-Yates, R.: A hybrid indexing method for approximate string matching. In: *Journal of Discrete Algorithms* 1 (2000), Nr. 1, S. 205–239. –
http://www.dartmouth.edu/~cbbc/courses/bio270/PDFs-03W/Doug-Navarro.pdf

[NBYST01] Navarro, G. ; Baeza-Yates, R.A. ; Sutinen, E. ; Tarhio, J.: Indexing Methods for Approximate String Matching. In: *IEEE Data Engineering Bulletin* 24 (2001), Nr. 4, S. 19–27. –
http://www.dcc.uchile.cl/~gnavarro/ps/deb01.ps.gz

[NKAJ59] Newcombe, H.B. ; Kennedy, J.M. ; Axford, S.J. ; James, A.P.: Automatic Linkage of Vital Records. In: *Science* 130 (1959), Nr. 3381, S. 954–959

[PCAAL00] Perez-Cortes, J.C. ; Amengual, J.C. ; Arlandis, J. ; Llobet, R.: Stochastic Error-Correcting Parsing for OCR Post-Processing. In: *Proceedings of the International Conference on Pattern Recognition ICPR '00*. Washington, DC, USA : IEEE Computer Society, 2000, S. 4405

[Phi00] Philips, L.: The double metaphone search algorithm. In: *C/C++ Users Journal* 18 (2000), Nr. 6, S. 38–43. –
http://www.ddj.com/cpp/184401251

[Pos08] PostCon Deutschland GmbH: *Consolidation Services*. 2008. –
http://www.postcon.de/

[PS06] Pestov, V. ; Stojmirovic, A.: Indexing Schemes for Similarity Search: an Illustrated Paradigm. In: *Fundamenta Informaticae* 70 (2006), Nr. 4, S. 367–385. –
http://www.searchlores.org/library/indexingschemes.pdf

[PW97] Porter, E.H. ; Winkler, W.E.: *Approximate String Comparison and Its Effects on an Advanced Record Linkage System*. Bureau of the Census, 1997. – 190–199 S. –
http://www.fcsm.gov/working-papers/porter-winkler.pdf

[Rec05] Reckhemke, Andre: *The construction of application-specific and index supported string similarity predicates*, Poznan University of Technology, Diplomarbeit, 2005. –
http://wwwiti.cs.uni-magdeburg.de/iti_db/publikationen/diplomarbeiten/mt_reckhemke_05.pdf

[Rin03] Ringlstetter, C.: *OCR-Korrektur und Bestimmung von Levenshtein-Gewichten*, LMU, University of Munich, Diplomarbeit, 2003. –
http://www.cis.uni-muenchen.de/~heller/SuchMasch/apcadg/literatur/data/ringlstetter03.pdf

[RJH91] Rosenbaum, Walter S. ; J. Hilliard, John: *System and method for deferred processing of OCR scanned mail*. 9. Juli 1991
http://www.freepatentsonline.com/5031223.html

[RO05] Russo, L.M.S. ; Oliveira, A.L.: Faster Generation of Super Condensed Neighbourhoods Using Finite Automata. In: *String Processing and Information Retrieval: 12th International Conference, SPIRE 2005, Buenos Aires, Argentina, November 2-4, 2005: Proceedings* (2005). –
http://www.inesc-id.pt/ficheiros/publicacoes/2958.pdf

[RS06] Reckhemke, A. ; Schallehn, E.: Combining Index Structures for application-specific String Similarity Predicates. In: Brass, Stefan (Hrsg.) ; Hinneburg, Alexander (Hrsg.): *Grundlagen von Datenbanken*, Institute of Computer Science, Martin-Luther-University, 2006, S. 125–129. –
http://dbs.informatik.uni-halle.de/GvD2006/gvd06_reck_schallehn.pdf

[Rus18] Russel, Robert: *Soundex*. 1918
http://patft.uspto.gov/netacgi/nph-Parser?patentnumber=1261167

[RY98] Ristad, Eric S. ; Yianilos, Peter N.: Learning string-edit distance. In: *IEEE Transactions on Pattern Recognition and Machine Intelligence* 20 (1998), Nr. 5, S. 522–532. –
http://www.qou.edu/homePage/arabic/researchProgram/distanceLearning/learningString.pdf

[Sch03] Schulz, K.U.: *Nachkorrektur von Ergebnissen einer optischen Charaktererkennung*. 2003
http://www.cis.uni-muenchen.de/~uli/kurse/ws0708/hist_ocr/material/schulz_ocr.pdf

[Sch06] Schulz, Jochen: *Unscharfe Suche in großen Adressbeständen*, FH Nordakademie Elmshorn, Diplomarbeit, 2006. –
http://wasteland.homelinux.net/~jrschulz/papers/2006-diplom/dipl.pdf

[SM96] Shang, H. ; Merrettal, TH: Tries for approximate string matching. In: *IEEE Transactions on Knowledge and Data Engineering* 8 (1996), Nr. 4, S. 540–547. –
http://doi.ieeecomputersociety.org/10.1109/69.536247

[Str04] Strohmaier, C.M.: *Methoden der lexikalischen Nachkorrektur OCR-erfasster Dokumente*, Ludwig-Maximilians-Universität München, Diss., 2004. –
http://edoc.ub.uni-muenchen.de/3674/1/Strohmaier_Christian.pdf

[Taf70] Taft, R.L.: *Name Search Techniques*. Bureau of Systems Development, New York State Identification and Intelligence System, 1970

[TE96] Tong, X. ; Evans, D.A.: A statistical approach to automatic OCR error correction in context. In: *Proceedings of the Fourth Workshop on Very Large Corpora* (1996), S. 88–100. –
http://acl.ldc.upenn.edu/W/W96/W96-0108.pdf

[TJTSF00] Traina Jr, C. ; Traina, A.J.M. ; Seeger, B. ; Faloutsos, C.: Slim-Trees: High Performance Metric Trees Minimizing Overlap Between Nodes. In: *Proceedings of the 7th International Conference on Extending Database Technology: Advances in Database Technology* (2000), S. 51–65. –
http://www.informedia.cs.cmu.edu/documents/EDBT_SlimTree.pdf

[TM01] Toutanova, K. ; Moore, R.C.: Pronunciation modeling for improved spelling correction. In: *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics* (2001), S. 144–151. –
http://www.stanford.edu/~krist/papers/P02-1019.pdf

[Uhl91] Uhlmann, J.K.: Satisfying general proximity/similarity queries with metric trees. In: *Information processing letters* 40 (1991), Nr. 4, S. 175–179

[Ukk85] Ukkonen, E.: Finding approximate patterns in strings. In: *Journal of Algorithms* 6 (1985), Nr. 1, S. 132–137

[Ukk92] Ukkonen, E.: Approximate string-matching with q-grams and maximal matches. In: *Theoretical Computer Science* 92 (1992), Nr. 1, S. 191–211. –
http://www.cs.helsinki.fi/u/ukkonen/TCS92.pdf

[Wik08a] Wikipedia: *Levenshtein-Distanz*. 11. Mai 2008. –
http://de.wikipedia.org/w/index.php?title=Levenshtein-Distanz&oldid=44430254

[Wik08b] Wikipedia: *Schreibmaschinendistanz*. 11. Mai 2008. –
http://de.wikipedia.org/w/index.php?title=Schreibmaschinendistanz&oldid=45909601

[Wik08c] Wikipedia: *Support Vector Machine*. 11. Mai 2008. –
http://de.wikipedia.org/w/index.php?title=Support_Vector_Machine&oldid=45434719

[Wil05] Wilder, Kenneth: *C++ Timer Class*. 2005. –
http://oldmill.uchicago.edu/~wilder/Code/timer/

[Win99] Winkler, W.E.: The state of record linkage and current research problems. In: *RR99/03, US Bureau of the Census* (1999). –
http://www.census.gov/srd/papers/pdf/rr99-04.pdf

[WM91] Wu, S. ; Manber, U.: Fast text searching with errors / University of Arizona, Department of Computer Science. 1991 (TR-91-11). – Forschungsbericht. –
http://citeseer.ist.psu.edu/wu91fast.html

[Yan04] Yancey, W.E.: *An Adaptive String Comparator for Record Linkage*. 2004. –
http://www.census.gov/srd/papers/pdf/rrs2004-02.pdf

[Yia93] Yianilos, P.N.: Data structures and algorithms for nearest neighbor search in general metric spaces. In: *Proceedings of the 4th annual ACM-SIAM Symposium on Discrete algorithms* (1993), S. 311–321. –
http://portal.acm.org/citation.cfm?id=313789

[ZD95] Zobel, J. ; Dart, P.W.: Finding Approximate Matches in Large Lexicons. In: *Software - Practice and Experience* 25 (1995), Nr. 3, S. 331–345. – http://www.cs.ubc.ca/local/reading/proceedings/spe91-95/spe/vol25/issue3/spe948jz.pdf

[ZWYY03] Zhou, X. ; Wang, G. ; Yu, J.X. ; Yu, G.: M+-tree – a new dynamical multidimensional index for metric spaces. In: *Proceedings of the 14th Australasian database conference on Database technologies 2003-Volume 17* (2003), S. 161–168. – http://portal.acm.org/citation.cfm?id=820085.820118