Selbstkonfigurierendes Entity Resolution Framework für Event Stream Processing

---

Kevin Sapper

21.04.2017

Version: 1.0

Hochschule RheinMain

Fachbereich Design Informatik Medien
Studiengang Master Informatik

Masterarbeit

Selbstkonfigurierendes Entity Resolution Framework für Event Stream Processing

Kevin Sapper

Referent Prof. Dr. Adrian Ulges
Hochschule RheinMain
Fachbereich Design Informatik Medien

Koreferent Prof. Dr. Reinhold Kröger
Hochschule RheinMain
Fachbereich Design Informatik Medien

Betreuer Thomas Strauß
Universum Group

21.04.2017

Kevin Sapper

Selbstkonfigurierendes Entity Resolution Framework für Event Stream Processing

Masterarbeit, 21.04.2017

Referenten: Prof. Dr. Adrian Ulges und Prof. Dr. Reinhold Kröger

Betreuer: Thomas Strauß

Hochschule RheinMain

Studiengang Master Informatik

Fachbereich Design Informatik Medien

Unter den Eichen 5

65195 Wiesbaden

Zusammenfassung

Entity Resolution ist der Prozess, in einer oder mehreren Datenquellen Gruppen von Datensätzen zu identifizieren, die derselben realen Entität entsprechen. Dabei gibt es kein einzigartiges Attribut, welches zur Zuordnung genutzt werden kann. Die Unterschiede in den Datensätzen entstehen, beispielsweise durch Rechtschreibfehler oder fehlende und vertauschte Attribute, welche eine Vieldeutigkeit erzeugen, die bei manueller Betrachtung durch einen Menschen meist nur mit viel Zeitaufwand aufzulösen sind. Damit ein Entity Resolution Workflow diese Vieldeutigkeiten auslösen kann, muss dieser abhängig von der Domäne der Daten konfiguriert werden. Diese Konfiguration besteht aus einer Vielzahl von Parametern, die auch von einem Domänexperten nur aufwändig zu bestimmen sind. Erster Beitrag dieser Arbeits ist deshalb die Analyse und Entwicklung von Verfahren, die eine Selbstkonfiguration der Parameter in Abhängigkeit der Datenquelle ermöglichen. Dabei liegt der Fokus dieser Arbeit auf Entity Resolution Verfahren für Event Stream Processing Systeme. Hierbei ist neben der Qualität der Ergebnisse auch die Antwortzeit von Bedeutung, welche oft im Subsekundenbereich liegen muss. Die Suche nach Duplikaten ist jedoch mit enormen Kosten verbunden, die beim vollständigen Durchsuchen aller Bestandsdatensätze in einer quadratischen Komplexität resultiert. Der Zweite Beitrag dieser Arbeit ist daher ein sog. Blocking-Verfahren zur Reduzierung der Komplexität, welches für Event Stream Processing tauglich ist. Für die Selbstkonfiguration bedeutet dies, dass neben der Qualität auch die Effizienz berücksichtigen muss. Die analysierten und entwickelten Verfahren wurden in einem prototypischen System implementiert, dass sich unüberwacht (ohne Eingreifen des Benutzers) vor der Laufzeit selbst konfiguriert und anschließend Anfragen aus einem Ereignisstrom beantwortet. Die Auswertung dieses Systems zeigt, dass die Selbstkonfiguration auf einem Datensatz mit 4 Mio Einträgen ein F-Measure von bis zu 70 % erreicht und bei 1.3 Mio Anfragen im Durchschnitt über 500 pro Sekunde beantwortet.

1 Einleitung

Heutzutage werden von vielen Unternehmen riesige Mengen von Daten gesammelt. Diese Daten sind meist Abbildungen von Entitäten der realen Welt, wie z.B. von Personen, Produkten oder Veröffentlichungen. Bei den meisten Abbildungen ist es nicht möglich einzigartige Attribute abzuleiten, anhand derer zwei oder mehr Abbildungen derselben Entitäten zugeordnet werden können. Zudem sind diese oft fehlerhaft und unvollständig, beispielsweise durch Rechtschreibfehler, fehlende oder vertauschte Attribute, wodurch Mehrdeutigkeiten entstehen. Diese Mehrdeutigkeiten aufzulösen, ist auch bei manueller Betrachtung durch einen Menschen, nicht trivial und darum häufig zeitaufwändig. Beispiele für mehrdeutige Personendaten sind unterschiedliche Schreibweisen von Nachnamen (`Maier` vs `Mayer`), vertauschte Vor- und Nachnamen, deren die Zuordnung nicht eindeutig ist (`Peter`, `Michel`, `Moritz`) oder ausländische Namen, wo die Schreibweise unbekannt ist. Dabei verlassen sich die Unternehmen in ihren Geschäftsprozessen auf diese Daten. Weshalb die Qualität der Abbildungen maßgeblichen Einfluss auf die Qualität eines Produktes oder einer Dienstleistung hat. In der Informatik werden die Manifestationen der abgebildeten Objekte als Datensätze in Datenbanken (o.ä.) bezeichnet. Zur Verbesserung der Qualität der gesammelten Daten wird in der Praxis eine Datenbereinigung (engl. data cleaning) durchgeführt. Ein wichtiger Aspekt der Datenbereinigung ist, alle Datensätze zu finden welche derselben Entität entsprechen. Verfahren, die Abbildungen der Entitäten finden und verlinken bzw. zusammenführen, werden üblicherweise Entity Resolution, Duplicate Detection oder Record Linkage genannt. Beispiele, wo mehrere Datensätze auf dieselbe Entität verweisen, sind Patientenakten in einer Krankenhausdatenbank oder ein Wählerverzeichnis, in welches eine Person öfters eingetragen ist. Für den Fall, dass diese Informationen nicht zusammengeführt oder verlinkt werden können, folgen teils schlimme Konsequenzen. Beispielsweise trifft ein Arzt, aufgrund unvollständiger Informationen, die falsche Entscheidung zur Behandlung eines Patienten oder ein Wahlberechtigter gibt mehrere Stimmen ab, was zu Wahlunstimmigkeiten führt. Weitere Einsatzbereiche sind Betrugserkennung, Bonitätsprüfung und Inkasso, hierbei sind die Konsequenzen finanzieller Art. Das Thema dieser Masterarbeit wird für den Problembereich der UNIVERSUM Group in Frankfurt am Main untersucht. Die UNIVERSUM Group bietet Online-Händlern an, die Einkäufe ihrer Kunden zu versichern. Das bedeutet, dass nach Ablauf einer Zahlungsperiode, bei Ausbleiben der Zahlung durch den Kunden, der Betrag durch die Versicherung gezahlt wird. Die UNIVERSUM Group wird in diesem Fall zum Gläubiger der Forderung und wird im Inkassoverfahren das Geld vom Kunden einfordern. Beim Inkassoverfahren müssen aufgrund gesetzlicher Bestimmungen mehrere Forderungen derselben natürlichen

Person zusammengefasst werden. Die meisten Forderungen fallen hierbei einmal täglich mit dem Ablauf der Zahlungsfrist an. Dadurch können Entity Resolution Verfahren eingesetzt werden, die periodisch (etwa jede Nacht) auf einem statischen Datenbestand operieren, der sich während der Laufzeit der Entity Resolution nicht verändert. Für jede Forderung im Datenbestand wird dadurch geprüft, ob es einen übereinstimmenden Schuldner gibt. Hierbei spielt hauptsächlich die Qualität eine entscheidende Rolle. Die Laufzeit ist lediglich durch die Periode (= 1 Tag/Nacht) begrenzt. Für Onlinedienste, wie die Betrugserkennung bzw. die Bonitätsprüfung, sind diese Verfahren nicht geeignet, weil Entity Resolution hierbei häufig nur ein Teilprozess eines Gesamtprozesses ist, sodass diese oft im Subsekundenbereich stattfindet muss. Bei der Bonitätsprüfung wird die Entity Resolution auf Anfrage durchgeführt und erst mit dem Ergebnis der Anfrage kann der Onlinehändler die Bestellung abschließen. Dementsprechend spielt neben der Qualität auch die Laufzeit eine wichtige Rolle. Technisch gesehen handelt es sich bei den meisten Onlinediensten um Event Stream Processing (ESP) Systeme, welche einen Datenstrom von Anfragendatensätzen in nahe Echtzeit bearbeiten müssen. Da ein solcher Datenstrom kein definiertes Ende hat, werden Änderungen am Datenbestand zur Laufzeit vorgenommen. Das bedeutet, dass der zu prüfende Datenbestand dynamisch ist und sich mit jeder Anfrage verändern kann. Zum einen die Laufzeitanforderungen und zum anderen die dynamischen Daten stellen Entity Resolution Verfahren hierbei vor eine Herausforderung.

Unabhängig von den eingesetzten Verfahren gibt es bei Entity Resolution stets die Schwierigkeit, die Parameter der Verfahren auf die Domäne der Daten anzupassen. Beispielsweise unterscheidet sich die Struktur eines Datensatzes mit Personendaten gravierend von dem einer Produktdatenbank, weshalb gute Parameter einer Domäne oft nicht übertragbar sind. Die Anpassung der Konfiguration ist ein aufwändiger Prozess, der selbst einen Domänexperten vor eine große Herausforderung stellt, da die Anzahl der Parameter leicht in den zweistelligen Bereich wächst und die Auswirkungen von Parametern auf Performanz und Güte schwer abzuschätzen sind. Wünschenswert wäre deshalb ein System oder Framework, das selbstkonfigurierend ist und folglich möglichst automatisiert die Parameter auf die Datendomäne anpasst. Da das Ausstellen von Versicherungen der UNIVERSUM Group ausschließlich durch deren Onlinedienst erfolgt, in welchem Bonitätsprüfung und Betrugserkennung durchgeführt werden müssen, liegt der Schwerpunkt dieser Arbeit auf Verfahren für Event Stream Processing Systeme. In Kapitel 2 werden zunächst die Grundlagen für Entity Resolution Verfahren erläutert und anschließend die genauen Anforderungen an Entity Resolution Verfahren für ESP-Systeme erläutert. Im übrigen Kapitel 2 werden ähnliche Arbeiten vorgestellt, die sich mit den diversen Teilbereichen der Entity Resolution befassen. In Kapitel 3 werden die vorgestellten Verfahren in Bezug auf die gestellten Anforderungen, sowie die Konfigurierbarkeit ihrer Parameter, analysiert und bei Bedarf angepasst. Kapitel 4 stellt auf Basis der Analyse ein System vor, dass sich selbst konfigurieren kann, um Entity Resolution für ESP-Systeme durchzuführen. Danach wird in Kapitel 5 eine

Evaluation dieses Systems durchgeführt. Dabei werden die Laufzeitanforderungen, die Qualität und die Effektivität der eingesetzten Verfahren überprüft und ausgewertet. Abschließend wird in Kapitel 6 ein Ausblick auf mögliche Weiterentwicklungen gegeben.

2 Grundlagen und ähnliche Arbeiten

Die Grundlagen geben einen Überblick zum Ursprung und der Funktionsweise der Entity Resolution (Abschnitt 2.1). Auf dieser Basis werden anschließend Anforderungen an einen Entity Resolution Workflow für Entity Stream Procession Systeme gestellt (Abschnitt 2.2). Damit die Anforderungen umgesetzt werden können werden in den Abschnitte 2.3, 2.4, 2.5 ähnliche Arbeiten vorgestellt. Am Ende des Kapitels werden geeignete Verfahren und Metriken zum Messen der Qualtität und Effizienz beschrieben (Abschnitt 2.6).

2.1 Entity Resolution

Die Methoden zur Duplikatserkennung stammen ursprünglich aus dem Gesundheitsbereich und wurden erstmal 1969 von Felegi & Sunter [1] formal formuliert. Je nach Fachgebiet gibt es unterschiedliche Fachbegriffe. Statistiker und Epidemiologen sprechen von *record* oder *data linkage*, während Informatiker das Problem unter *entity resolution*, *data* oder *field matching*, *duplicate detection*, *object identification* oder *merge/purge* kennen. Identifiziert werden sollen dabei beliebige Entitäten, welche oft in Form von Datensätzen einer Datenbank vorliegen. Die Schwierigkeit dabei ist allerdings, dass Entitäten nicht durch ein einzigartiges Attribut identifiziert werden können, beispielsweise Produkte, bibliografische Einträge oder selbsterfasste Onlineauskünfte. Zudem sind die Datensätze oft fehlerhaft, beispielsweise durch Rechtschreibfehler, welche durch Tippfehler, Hörfehler oder OCR-Fehler enstehen. Eine andere Fehlerquelle sind unterschiedliche Konventionen, beispielsweise bei Endungen von Straßennamen `strasse`, `straße` oder `str`. Auch denkbar sind Fehler aufgrund von Betrug. Entity Resolution (ER) Verfahren vergleichen meist eine oder mehrere Datenbanken, indem Datensatzpaare gebildet werden. Als Ergebnis wird eine Menge von übereinstimmenden Datensatzpaaren, d.h. zwei Datensätze, welche die selbe Entität beschreiben, geliefert. Damit eine Übereinstimmung zwischen zwei oder mehr Entitäten festgestellt werden kann, müssen diese verglichen werden und ein Ähnlichkeitswert (engl. similarity score) bestimmt werden. Dieser Ähnlichkeitswert gibt die Intensität der Übereinstimmung an. Das ER Problem wird formal von Köpcke & Rahm in [2] folgendermaßen beschrieben. Gegeben sind zwei Mengen von Entitäten $A \in S_A$ und $B \in S_B$ zweier Datenquellen S_A und S_B, welche semantisch dem selben Entitätstypen entsprechen. Das ER Problem ist es, alle Paare in $A \times B$ zu identifizieren, welche derselben Entität entsprechen. Als Spezialfall ist dabei die Suche in einer Daten-

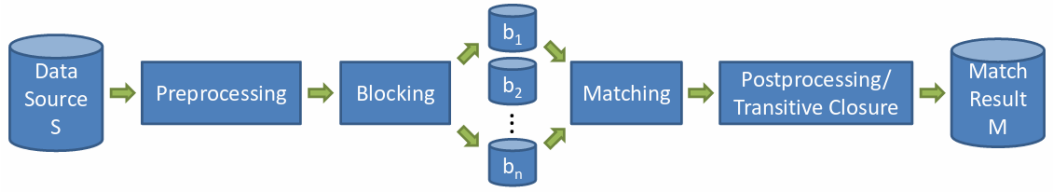

Abbildung 2.1 Vereinfachter Entity Resolution Workflow aus [3]. Die Datenquelle S wird vorverarbeitet und in kleinere Submengen gegliedert. Innerhalb dieser werden Datensatzpaare miteinander verglichen und paarweise bestimmt, ob diese der selben Entität entsprechen. Abschließend werden aus Paaren Gruppen von Duplikaten ermittelt und als Ergenisse M geliefert.

quelle $A = B, S_A = S_B$ zu betrachten. Eine Übereinstimmung $u = (e_i, e_j, s)$ verknüpft zwei Entitäten $e_i \in S_A$ und $e_j \in S_B$ mit einem Ähnlichkeitswert $s \in [0, 1]$.

In der klassischen Variante arbeitet Entity Resolution auf statischen Daten, d.h., dass während des ER-Prozesses keine neuen Daten hinzukommen. Hierbei werden die zwei Disziplinen Deduplizierung und Entity-Linking unterschieden. Die Deduplizierung wird auf einer Datenquelle durchgeführt und hat den Zweck, alle Duplikate in dieser Datenquelle zu finden. Anschließend werden die gefundenen Duplikate automatisch oder manuell zusammengeführt. Entity Linking hingegen wird auf mindestens zwei verschiedenen Datenquellen durchgeführt. Das Ziel ist es, nicht Duplikate zusammenzuführen, sondern Entitäten zwischen den Datenquellen zu verlinken. Damit die Links eindeutig sind, wird in der Regel vorausgesetzt, dass die einzelnen Datenquellen dedupliziert sind.

Die Ausführung der Vergleichsmethoden ist enorm teuer, da diese jedes Attribut zweier Datensätze miteinander vergleichen. Bei einem Vollvergleich aller Datensätze, führt dies zu einer quadratischen Komplexität pro Attribut, was dafür sorgt, dass bei großen Datenmengen die Ausführungszeit unakzeptabel lang wird. Um die Ausführungszeit zu reduzieren, wird versucht, den Suchraum auf die wahrscheinlichsten Duplikatsvorkommen zu begrenzen. Diese Vorgehen werden als Blocking oder Indexing bezeichnet.

Abbildung 2.1 zeigt einen vereinfachten typischen Entity Resolution Workflow. Zunächst werden die Datensätze einer Datenquelle S vorverarbeitet, um typische Fehler zu entfernen. Dazu gehört das Korrigieren von Rechtschreibfehler, Ignorieren von Groß- bzw. Kleinschreibung, beispielsweise durch Konvertierung in Kleinschreibung, und das Ersetzen von bekannten Abkürzungen. Durch die Vorverarbeitung kann die Qualität des Matchings verbessert, indem verhindert wird, dass offensichtliche Abweichungen den Ähnlichkeitswert beeinflussen. Der nächste Schritt, das Blocking, teilt die Gesamtmenge in Submengen $b_1, b_2, \cdots, b_n$ zur Reduzierung der Gesamtkomplexität, da nur die jeweiligen Blöcke voll verglichen werden. In Abschnitt 2.3 werden detailliert verschiedene Blockingverfahren erläutert. Auf das Blocking folgt das Matching, hierbei werden innerhalb der Submengen von Datensatzpaaren Ähnlichkeitswerte bestimmt. Die Möglichkeiten der Ähnlichkeitsbestimmung werden in Abschnitt 2.4 beschrieben. Anhand

der Ähnlichkeitswerte wird anschließend für jedes Datensatzpaar entschieden, ob es sich um ein Match (beide Datensatze beschreiben dieselbe Entität) oder ein Non-Match (die Datensätze beschreiben unterschiedliche Entitäten) handelt. Diese Klassifikation wird genauer in Abschnitt 2.5 erklärt. Abschließend findet noch die Berechnung der transitiven Hülle statt, um beispielsweise aus Paaren von Matches Gruppen zu bilden, welche derselben Entität entsprechen $M = (a, b), (b, c) \implies M = (a, b, c)$.

Laut Köpcke & Rahm [2] gibt es keine Methode zur Entity Resolution, welche allen anderen überlegen ist. Vielmehr ist der Erfolg unterschiedlicher Methoden abhängig von der Datendomäne. Deshalb wurde Anfang der 00er Jahre begonnen, Frameworks zu entwickeln, welche verschiedene Methoden miteinander kombinieren. Ein Vergleich dieser Frameworks wurde durch Köpcke & Rahm [2] durchgeführt. Ein Framework besteht hierbei aus verschiedenen Matchern und Matching Strategien. Ein Matcher ist dabei ein Algorithmus, welcher die Ähnlichkeit zweier Datensätze ermittelt. Köpcke & Rahm unterscheiden zwischen attributs- und kontextbasierenden Matchern. Als Kontext wird die semantische Beziehung bzw. Hierarchie zwischen den Attributen bezeichnet, beispielsweise in Graphstrukturen, welche es erlauben Ähnlichkeitswerte über Kanten zu propagieren. Um die Matcher miteinander zu kombinieren, nutzen die Frameworks mindestens eine Matching-Strategie. Durch die Match-Strategie werden verglichene Datensatzpaare in die Mengen *Matches* und *Non-Matches* klassifiziert.

Ein Großteil der Forschung in Entity Resolution konzentriert sich auf die Qualität der Vergleichsergebnisse. Die von Köpcke & Rahm betrachteten Frameworks konzentrieren sich hierbei alle darauf, zwei statische Mengen zu miteinander vergleichen. Bei großen Datenmengen kann dies durchaus mehrere Stunden dauern. Daher gibt es in den letzten Jahre einige Ansätze und Frameworks, welche MapReduce-Algorithmen nutzen, um die Laufzeit zu verkürzen [4]. Einen Ansatz, die Laufzeit für Anwendungen mit Laufzeitanforderungen zu optimieren, präsentieren Whang et al. [6]. Anstatt eine Übereinstimmungsmenge nach Abschluss eines Algorithmus zu liefern, zeigen sie Möglichkeiten, partielle Ergebnisse, während der Laufzeit des Algorithmus zu erhalten. Dabei modifizieren sie die Blocking-Algorithmen, sodass zunächst die wahrscheinlichsten Kandidaten miteinander verglichen werden. Dabei wird in relativ kurzer Zeit ein Großteil der Duplikate gefunden.

Neben den statischen Verfahren gibt es zunehmend Bedarf an dynamischen Verfahren. Dynamisch bedeutet hier, dass während der Laufzeit neue Datensätze hinzugefügt werden können. Das Finden gleicher Entitäten erfolgt dabei auf Anfrage, weshalb die gesamte Datenmenge vorab nicht bekannt ist. Beispielsweise müssen Kreditauskunfteien auf Anfrage prüfen, ob ein Kunde kreditwürdig ist. Dazu müssen die passenden Entitäten möglichst schnell gefunden werden, um eine Entscheidung treffen zu können. Zudem ist es notwendig, eine Historie der unveränderten Anfragen aller Entitäten vorzuhalten, da diese Beweise über frühere Anfragen liefern. Ramadan et al. [7] formulieren die Pro-

blemstellung für dynamische ER-Verfahren folgendermaßen: für jeden Anfragedatensatz q_j eines Anfragestroms Q sollen alle Datensätze M_{q_j} in einem Datensatz R gefunden werden, welche dieselbe Entität wie q_j beschreiben.

$$M_{q_j} = \{r_i | r_i.eid = q_j.eid, r_i \in R\}, M_{q_j} \subseteq R, q_j \in Q,$$

wobei eid ein eindeutiger Identifikator einer Entität ist, welcher so nicht existiert. Die Herausforderung für dynamische ER-Verfahren ist weiter nach Ramadan et al., Indexing-Verfahren zu entwickeln, welche es erlauben den Index dynamisch zu erweitern und eine kleine Zahl qualitativer Ergebnisse in nahe Echtzeit (Subsekundenbereich) zu liefern. Ein dynamisches ER System ist ähnlich einer Suchmaschine, doch anstatt einer gewerteten Liste möglicher Treffer soll es alle gleichen Entitäten finden, welche zur Anfrage passen. Das bedeutet insbesondere, dass die Anfrage die gleiche Datenstruktur haben muss, wie die zu durchsuchende Datenquelle. Zudem kann eine Anfrage während der Abfrage als neuer Datensatz aufgenommen werden. Erste Ergebnisse, Entity Resolution in nahe Echtzeit zu erreichen, präsentieren Christen & Gayler in [8], unter Verwendung von Inverted Indexing Techniken, welche normalerweise bei der Websuche Anwendung finden. Die dynamischen Verfahren werden in Abschnitt 2.3.2 behandelt.

2.2 Anforderungen an eine ER-System für ESP

Das Event Stream Processing besteht aus drei Teilen[1]:

- Event – Ein Ereignis irgendeines Vorkommnis, zu einer eindeutig definierten Zeit, welches durch eine Menge von Attributen aufgezeichnet wird.
- Stream – Ein Datenstrom ist ein konstanter Fluss oder stetiger Ansturm von Daten, die von einer beliebigen Anzahl verbundener Geräte zu einem Dienst fließen.
- Processing – Ist die Handlung, welche die Daten analysiert.

Für ein Entity Resolution Framework, das Teil eines Event Stream Processing Systems ist, werden Ereignisse als Anfragen bezeichnet. Dabei besteht eine Anfrage aus einem Datensatz, der strukturell identisch zu den Bestandsdaten ist. Das Ergebnis einer Anfrage ist eine Menge von Datensätzen, die derselben Entität entsprechen. Unabhängig von den Datenstrukturen verschiedener Domänen, muss ein Entity Resolution Framework für ESP-Systeme stets abwägen zwischen *Qualität* und *Effizienz*. Diese Abwägung führt zu den folgenden drei Problemen:

[1] https://www.sas.com/en_us/insights/articles/big-data/3-things-about-event-stream-processing.html

- **Niedrige Latenzen**. Der Zeitraum vom Stellen einer Anfrage bis zu deren Beantwortung sollte im Subsekundenbereich liegen. Die Gründe dafür sind vielfältig, beispielsweise bietet eine Produktsuche eine bessere Benutzererfahrung, je schneller die Daten verfügbar sind, weil Benutzer dazu geneigt sind einen Dienst nicht mehr zu nutzen, wenn die Benutzererfahrung durch zu lange Wartezeiten getrübt wird. Eine Kreditauskunft ist meist Teil eines größeren Prozesses, sodass deren Laufzeit die Benutzererfahrung von anderen Diensten beeinflussen kann, weshalb eine solche Auskunft seinen Kunden Garantien für die Laufzeit der Anfragen gibt, um sich gegenüber der Konkurrenz hervorzuheben. Hauptverantwortlich für niedrige Latenzen sind in einem Entity Resolution Framework die Blocking Verfahren. Während einer Anfrage dauert die Berechnung der Ähnlichkeitswerte zwischen Attributen des Anfragedatensatzes und durch Blocking gefilterter Datensatzkandidaten am längsten. Um niedrige Latenzen zu erreichen, muss das Blocking die zu durchsuchende Kandidatenmenge stark reduzieren und kann außerdem versuchen den Aufwand von Ähnlichkeitsvergleichen, beispielsweise durch Caching, zu reduzieren.
- **Datenmodifikation zur Laufzeit**. Der Datenbestand kann mit jedem Ereignis modifiziert werden, sodass ständig neue Entitäten hinzukommen oder modifiziert werden. Das Entfernen wird nur in äußersten Ausnahmen vorgenommen und findet deshalb keine weitere Beachtung. Sobald eine Enitität hinzugekommen ist bzw. modifiziert wurde, muss die Änderung sofort für die darauffolgenden Anfragen zum Abgleich auffindbar sein. Werden die Anfragen stets sequentiell bearbeitet, ist diese Anforderung realtiv einfach zu erfüllen. Da aber auch bei hoher Last die Latenzen niedrig bleiben müssen, kann ein Entity Resolution Framework die Bearbeitung auf mehrere Threads, Prozesse oder Knoten skalieren, um die Anfragen parallel zu bearbeiten. Erhält das System nacheinander mehrere Anfragen, die derselben Entität zuzuweisen sind, dann ist die Wahrscheinlichkeit hoch, dass diese parallel vorarbeitet werden. Damit allerdings die zweite Anfrage, die die erste als Teil ihrer Ergebnissemenge enthält, müssen diese, bei Nutzung von Paralellisierung, sequentialisiert werden, ohne dadurch einen Flaschenhals für das Gesamtsystem zu bilden.
- **Hohe Trefferrate**. Unter den Latzen darf naturlich nicht die Qualität des Ergebnisses leiden. Das Ziel ist es zu einer Entität möglichst alle im Datenbestand existierenden Datensätze zu finden und zurückzugeben. Dazu wird die Kandidatenmenge untersucht, die durch ein Blocking Verfahren gebildet wurde. Da diese Menge auch Datensätze enthält die nicht derselben Entität entsprichen, müssen diese nach der Ähnlichkeitsberechnung von einem Klassifikator in *Matches* und *Non-Matches* klassifiziert werden. Damit dieser möglichst wenige Missklassifikationen unternimmt, d.h. Matches als Non-Matches deutet bzw. umgekehrt, muss der Ähnlichkeitswert eine hohe Aussagekraft haben. Deshalb ist es wichtig für jedes Attribut eine geeignete Ähnlichkeitsfunktion zu finden, die dem Klassifikator erlaubt, auf den Attributsähnlichkeiten, möglichst eindeutige Entscheidungen zu treffen.

Um diese drei Probleme zu lösen, werden im weiteren Kapitel die Problembereiche *Blocking*, *Ähnlichkeitsberechnung* und *Klassifikation* betrachtet, welche Teil jedes Entity Resolution Workflows sind. Neben weiteren Grundlagen, werden zu diesen Problembereichen vor allem ähnliche Arbeiten vorgestellt.

2.3 Blocking

Blocking dient der Reduzierung der quadratischen Komplexität eines ER Verfahrens. Im Folgenden werden Verfahren unterschieden die entwickelt wurden, um auf statischen oder dynamischen Datenquellen angewandt zu werden.

2.3.1 Statisches Blocking

Für die Duplikatserkennung in zwei Datenquellen A und B sind $|A| \cdot |B|$ Paarvergleiche notwendig. Bei einer einzelnen Datenquelle A müssen $\frac{1}{2} \cdot |A| \cdot (|A| - 1)$ Vergleiche durchgeführt werden. In beiden Fällen ist die Anzahl der Vergleiche quadratisch zur Eingabemenge [3]. In der Studie [9] zeigen Köpcke et al., dass das kartesische Produkt für große Datenmengen nicht skaliert. Aus diesem Grund reduzieren moderne Entity Resolution Frameworks den Suchraum auf die wahrscheinlichsten Kandidaten, die sogenannten Match-Kandidaten. Diese Methoden zur Reduzierung des quadratischen Suchraum werden übergreifend als Blockingmethoden bezeichnet. Neben Blocking werden auch Windowing- und Indexing Verfahren eingesetzt. Während Blockingverfahren die Anzahl der notwendigen Vergleiche drastisch reduzieren, indem Non-Matches ausgeschlossen werden, besteht dennoch die Gefahr, dass fälschlicherweise tatsächliche Matches ausgefiltert werden. Daher ist es notwendig die Güte des Blockingverfahrens zu bestimmen. Zwei Kennziffern werden dazu erhoben. Zum einen die *Reduction Ratio*, welche die Reduzierung der Vergleiche im Gegensatz zum Kartesischen Produkt ausdrückt und zum anderen die *Pairs Completeness*, welche den Anteil der tatsächlich ausgewählten Duplikate, die sich nach dem Blocking in der Kandidatenmenge befinden, beschreibt. Eine detaillierte Beschreibung der Komplexitätsmaße für Blocking wird in Abschnitt 2.6 vorgenommen.

Prinzipiell erfolgt Blocking entweder durch Gruppierung oder Sortierung. Mögliche Duplikate sollen dadurch in gegenseitige “Nähe” gebracht werden. Zur Durchführung der Gruppierung oder Sortierung müssen sog. Block- bzw. Sortierschlüssel für jeden Datensatz erzeugt werden. Die Ableitung der Schlüssel erfolgt anhand von den Attributswerten oder einem Teil der Attributswerte. Die Ableitung stellt eine Signatur des Datensatzes dar. Eine beliebte Variante für Schlüssel sind etwa phonetische Enkodierung.

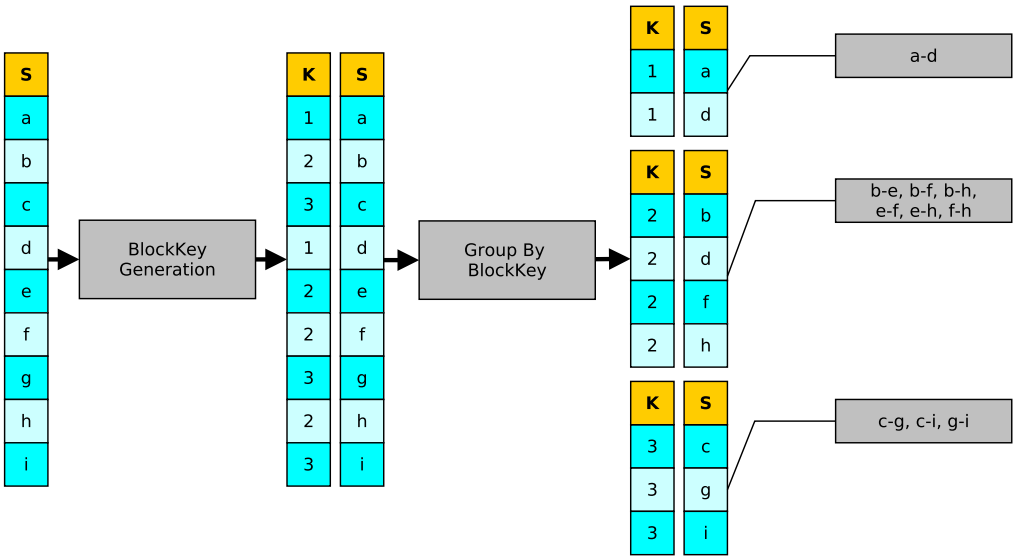

Abbildung 2.2 Beispielhafte Standard Blocking Ausführung nach [3]. Für jeden Datensatz in S wird ein Blockschlüssel K erzeugt, anhand derer werden Blöcke erzeugt und innerhalb der Blöcke werden Paare gebildet.

Standard Blocking

Standard Blocking ist eine der ersten und populärsten Blockingmethoden [1]. Die Idee des Verfahrens ist eine Menge von Datensätzen in disjunkte Partitionen (genannt Blöcke) zu teilen und diese anschließend nur Datensätze der jeweiligen Blöcke miteinander vergleichen. Dazu wird jedem Datensatz ein Blockschlüssel zugeordnet. Die Qualität des Blockingverfahrens hängt daher maßgeblich vom gewählten Blockschlüssel ab, da dieser die Anzahl und Größe der Partitionen bestimmt. In einer Menge von Personen wäre ein schlechter Blockschlüssel das Geschlecht, weil dadurch lediglich zwei große Blöcke erzeugt werden. Ein besserer Blockschlüssel ist beispielsweise die Postleitzahl oder die ersten drei Ziffern der Postleitzahl [10]. Abbildung 2.2 zeigt die Ausführung des Blockingverfahrens beispielhaft an einer Datenquelle S. Zunächst wird jedem Datensatz (a - i) ein Blockschlüssel (hier 1, 2, 3) zugeordnet. Anschließend wird anhand dieses Schlüssels gruppiert. Die Größe der einzelnen Blöcke bestimmt das Reduction Ratio, dieses hängt allerdings immer von der Datenquelle ab und kann daher nicht pauschalisiert werden. Bei der Generierung der Blockschlüssel können fehlerhafte Werte einzelner Attribute dazu führen, dass Duplikate in unterschiedlichen Blöcken landen. Damit diese Duplikate dennoch gefunden werden, können für jeden Datensatz mehrere Blockschlüssel, anhand unterschiedlicher Attribute, generiert werden. Dieser Ansatz nennt sich Multipass Blocking [3]. Im Folgenden werden mit Q-Gram Indexing und Suffix Array Indexing zwei Verfahren diskutiert, die ein unscharfes Matching der Schlüssel erlauben und dadurch etwa Tippfehler auflösen können.

Q-Gram Indexing

Das Q-Gram Indexing basiert auf der Idee, Datensätze unterschiedlicher aber ähnlicher Blockschlüssel, miteinander zu vergleichen. Ein Blockschlüssel wird dazu in eine Liste G von q-Grammen überführt. Ein q-Gram ist ein Substring der Länge q des ursprünglichen Blockschlüssels. Beispielsweise erzeugt $q = 2$ angewendet auf den Blockschlüssel banana die Liste $G = ba, an, na$. Alle Kombinationen der q-Gram Liste mit einer Mindestlänge $l = max(1, \lfloor \#G \cdot t \rfloor)$ werden konkateniert und dienen als Schlüssel der Blöcke, wobei t ein Schwellwert zwischen 0 und 1 ist. Für $t = 0.9$ werden die Sublisten $(ba, an), (ba, na), (an, na), (ba, an, na)$ der Längen 2 und 3 gebildet. Datensätze werden dabei mehreren Blöcken zugewiesen. Dieses Verfahren kann als Alternative zum Multi-pass Verfahren beim Standard Blocking genutzt werden. Ist $t = 1$ wird lediglich ein Blockschlüssel erzeugt, was dem Standard Blocking entspricht. Der große Nachteil ist der hohe Aufwand bei der Berechnung aller möglichen Sublisten. Ein Blockschlüssel mit n Zeichen muss in $k = n - q + 1$ q-Gramme zerlegt werden. Insgesamt müssen dadurch $\sum_{i=max\{1,[k \cdot t]\}}^{k} \binom{k}{i}$ Sublisten berechnet werden [11].

Suffix Array Indexing

Das Suffix Array Indexing [12] leitet, ähnlich wie das Q-Gram Indexing, mehrere Schlüssel aus einem Blockschlüssel ab. Grundidee ist es, alle Suffixe mit einer Mindestlänge von l zu bestimmen. Ein Datensatz mit Blockschlüssellänge n wird in $n - l + 1$ Blöcke eingeordnet. Ist $n < l$ wird der Ausgangsschlüssel als einziger Schlüssel verwendet. Durch die größere Menge an Kandidatenpaaren ist i.Allg. die *Pairs Completeness* höher (vgl. Multi-pass). Zudem ist der Aufwand der Berechnung der Schlüssel im Gegensatz zu Q-Grammen deutlich geringer. Im Gegensatz zum Standard Blocking ist die Menge an Kandidatenpaaren jedoch deutlich höher. Dadurch ist auch die Wahrscheinlichkeit, dass zwei Datensätze unnötigerweise mehrfach miteinander verglichen werden hoch. Aus Blöcken, welche einen bestimmten Schwellwert überschreiten, werden daher alle Datensätze entfernt, die mindestens einen weiteren längeren Blockschüssel haben.

Sorted Neighborhood

Das Sorted Neighborhood Verfahren, ist ein Sortierverfahren, welches 1995 von Hernández & Stolfo [13] zur Erkennung von Duplikaten in Datenbanktabellen vorgestellt wurde. Es besteht aus drei Phasen. Zunächst bekommt jeder Datensatz einen Sortierschlüssel zugewiesen, dabei muss der Sortierschlüssel nicht einzigartig sein. Um die Berechnung des Schlüssels gering zu halten, soll dieser durch Verkettung von Attributen bzw. Teilen der Attribute bestimmt werden. Attribute die vorne im Schlüssel stehen haben dadurch eine höhere Priorität. In der zweiten Phase werden die Datensätze anhand des Schlüssels sortiert. In der dritten Phase wird ein Fenster (engl. Window) über die sortierten

Datensätze geschoben und alle Datensätze innerhalb des Windows werden miteinander verglichen. Dieses Verfahren eignet sich besonders gut zur Erkennung von Duplikaten innerhalb einer Datenquelle. Sollen Duplikate in mehreren Datenquellen gefunden werden, müssen die Einträge beim Sortieren gemischt werden. Dadurch besteht allerdings die Gefahr, dass vorrangig Datensätze einer Datenquelle miteinander verglichen werden. Die Vorteil gegenüber dem Standard Blocking ist, dass die Anzahl der Vergleiche lediglich von der Größe der Datenquelle und der gewählten Fenstergröße abhängen. Ein großer Nachteil ist, dass Datensätze, die sich in der ersten Stelle des Schlüssels unterscheiden, weit voneinander entfernt sind und dadurch nicht als Matches identifiziert werden. Um dennoch eine hohe Pairs Completeness zu erreichen, werden mehrere Schlüssel pro Datensatz generiert und ein Fenster mit kleiner Größe über die verschieden sortierten Listen geschoben. Dieses Verfahren entspricht im Grunde dem Multi-pass Verfahren beim Standard Blocking.

Ein großes Problem bei der klassischen und der Multi-pass Variante des Sorted Neighborhood Verfahrens ist, dass die zu wählende Fenstergröße w größer als die Anzahl der Datensätze mit dem am häufigsten vorkommenden Sortierschlüssel sein muss, um eine gute Pairs Completeness zu erreichen. Sei n die Menge an Datensätzen mit dem am häufigsten vorkommenden Schlüssel k und m die Menge der Datensätze des darauffolgenden Schlüssels $k + 1$, dann ist $w = n + m$. Nur dadurch kann sichergestellt werden, dass alle Datensätze aus n mit den "nahen" Datensätzen aus m verglichen werden. Sortierschlüssel sind für gewöhnlich nicht gleichverteilt, daher gibt es meist wenige große und viele kleine Mengen an Datensätzen mit dem gleichen Sortierschlüssel, wodurch Datensätze mit seltenen Sortierschlüsseln unnötig oft mit "weit" entfernten Datensätzen verglichen werden. Zudem dominiert der am häufigsten vorkommenden Schlüssel die Ausführungszeit des Algorithmus, genauso wie beim Standard Blocking,

In [10] schlagen Draisbach & Naumann eine optimierte Variante des Sorted Neighborhood Verfahrens vor. Dabei zeigen sie, dass Standard Blocking und Sorted Neighborhood zwei extreme von Überlappungen bei Partitionen sind. Gegeben sind zwei Partitionen P_1 und P_2, dann ist die Überlappung $U_{P_1,P_2} = P_1 \cap P_2$ und $u = |U_{P_1,P_2}|$. Ihre Idee ist es diese Überlappung zu optimieren, welche groß genug sein muss, um tatsächliche Matches zu finden, aber gering genug, um die Anzahl der Vergleiche zu reduzieren. Zunächst wird wie beim klassischen Verfahren sortiert. Danach werden angrenzende Datensätze in disjunkte Partitionen zerlegt und schließlich wird ein Überlappungsfaktor u gewählt. Innerhalb jedes Blockes wird analog zum Standard Blocking jeder Datensatz mit jedem anderen verglichen. Innerhalb des Overlap-Window $w = u + 1$, wird jeweils das erste Element mit allen anderen verglichen. Ist $w = 0$ entspricht das Verfahren dem Standard Blocking, hat jede Partition nur ein Element entspricht es der Sorted Neighborhood Methode. Um zu vermeiden, dass eine Partition dominiert, können größere Partitionen in Subpartitionen geteilt werden.

Record ID	First name	Double-Metaphone
r1	tony	tn
r2	cathrine	k0rn
r3	tony	tn
r4	kathryn	k0rn
r5	tonya	tn
r6	cathrine	k0rn
r7	linda	lnt
r8	tonia	tn

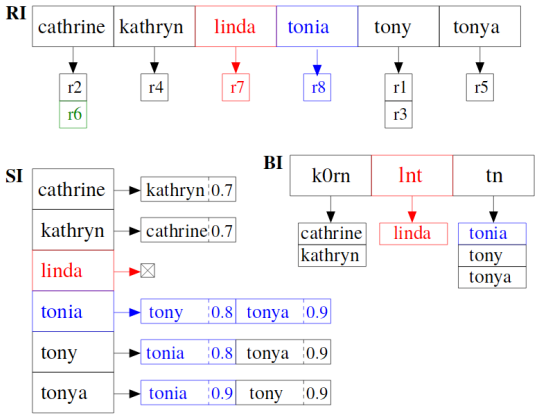

Abbildung 2.3 Ein DySimII-Index, welcher aus der Tabelle links erzeugt worden ist. Die Beispieldatensätze enthalten das Namensattribut und eine Double-Metaphone Enkodierung, welche als Blockingschlüssel genutzt wird. **RI** ist der Record Identifikatoren Index, **BI** der Block Index und **SI** der Similarity Index. Das Beispiel ist aus [15] entnommen.

Canopy Clustering

Cohen & Richman [14] schlagen ein Clustering-Verfahren zum Blocken, auf Basis von Canopies, vor. Die Idee des Canopy Clustering ist es, Datensätze anhand einer einfachen Vergleichsmetrik in überlappende Cluster (=Canopies) zu partionieren. Zur Generierung wird eine Kandidatenliste gebildet, welche initial aus allen Datensätzen besteht. Ein Cluster wird gebildet, indem zufällig ein Cluster-Zentroid gewählte wird und alle Datensätze innerhalb des Mindestabstandes d_1 diesem zugewiesen werden. Zusätzlich werden alle Datensätze dieses Clusters mit einem weiteren Mindestabstandes $d_2 < d_1$ aus der Kandidatenliste entfernt. Dieser Algorithmus wird wiederholt, bis die Kandidatenliste leer ist. Die *Pairs Completeness* hängt hierbei stark von der gewählten Abstandsfunktion ab. Anschließend werden alle Datensätze eines Cluster miteinander verglichen.

2.3.2 Dynamisches Blocking

Für die Dupliaktserkennung in einer Datenquelle A, sind bei einer Anfrage $|A|$ Vergleiche notwendig. Dies würde zu ungewollt hohen Latenzen führen, weshalb auch im dynamischen Fall Blocking Verfahren eingesetzt werden. Besonders wichtig für dynamische Verfahren ist es, dass Anfragen möglichst schnell beantwortet werden. Erreicht wird dies, indem Verfahren einen Teil der Vergleiche vorausberechnen. Des Weiteren sollen die Anwortzeiten möglichst gleich sein, um zu verhindern das manche Anfragen um ein vielfaches länger benötigen. Das bedeutet, dass die Kandidatenpaare, die pro Anfrage in Frage kommen, möglichst gleich sein müssen.

Algorithm 1 DySimII - Build

Input:
- Data set: D
- Encoding functions: $E_i, i = 1 \cdots n$
- Similarity functions: $S_i, i = 1 \cdots n$

Output:
- Index data structures: RI, BI, SI

Initialize $RI = \{\}, BI = \{\}, SI = \{\}$
for records $r \in D$ **do**
  for attributes $a = 1 \cdots n$ **do**
    Append $r.id$ into $RI[r.a]$
    if $r.a \notin SI$ **then**
      $c = E_a(r.a)$
      Append $r.a$ to $BI[c]$
      for attribute $v \in BI[c]$ **do**
        $sim = S_a(r.a, v)$
        Append $(r.a, sim)$ to $SI[v]$
        Append (v, sim) to $SI[r.a]$
      end for
    end if
  end for
end for
return RI, BI, SI

Algorithm 2 DySimII - Query

Input:
- Query record: q
- Encoding functions: $E_i, i = 1 \cdots n$
- Similarity functions: $S_i, i = 1 \cdots n$
- Indexes: RI, BI, SI

Output:
- Matches: M

Initialize dictionary $M = \{\}$
for attributes $a = 1 \cdots n$ **do**
  if $q.a \notin RI$ **then**
    $Insert(q.a, q.id, E_a, S_a)$
  else
    Append $q.id$ to $RI[q.a]$
  end if
  for $r.id \in RI[q.a]$ **do**
    $M[r.id] = M[r.id] + 1.0$
  end for
  for $(r.a, sim) \in SI[q.a]$ **do**
    for $r.id \in RI[r.a]$ **do**
      $M[r.id] = M[r.id] + sim$
    end for
  end for
end for
Sort M according to similarities
return M

DySimII

Der Dynamic Similarity-Aware Inverted Index [15] ist die dynamische Erweiterung (Version 2) des Similarity-Aware Inverted Index von Christen & Gayler [8], deshalb abgekürzt DySimII. Die Funktionalität beider Verfahren ist identisch, bis auf die Ausnahme, dass der DySimII es erlaubt den Index während der Laufzeit zu erweitern. Die Grundidee ist es, die benötigten Ähnlichkeiten vorauszuberechnen, um diese während der Laufzeit nur nachschlagen zu müssen.

Der Index besteht aus drei Teilen. Dem **Record Identifikatoren Index (RI)**, welcher alle Attribute speichert und diese ihren Datensätzen zuordnet, dem **Block Index (BI)**, welcher Attribute anhand einer Enkodierungsfunktion gruppiert und zuletzt dem **Similarity Index (SI)**, welcher dieselben Schlüssel wie der Record Identifikatoren Index verwendet und die Ähnlichkeiten der Attribute im gleichen Block hält. Abbildung 2.3 zeigt ein Beispiel eines DySimII Index. Im RI wurden die Datensatzidentifikator von Tony (r1, r3) und Cathrine (r2, r6) als Attributsübereinstimmung gruppiert. Anschließend wurden im BI über die Double-Metaphone Enkodierung, welche einen identischen String für gleich klingende Wörter erzeugt, ähnliche Schreibweisen von Tony und Cathrine zusammengeführt, was dem Standard Blocking entspricht. Im SI wurden die Ähnlichkeiten von Tony, Tonia und Tonya bzw. Cathrine und Kathryn, welche sich in einem gemeinsamen Block befinden, untereinander mit ihren berechneten Ähnlichkeiten verknüpft.

Das Verfahren unterscheidet zwei Phasen. Die Bauphase (engl. Build-Phase), in welcher der Index, aus einem initialen Datenbestand, erzeugt wird (Algorithmus 1) und die Anfragephase (engl. Query-Phase), welche Anfragen aus einem Datenstrom beantwortet (Algorithmus 2).

Build Phase. Das Einfügen von Datensätzen läuft nach dem folgendem Schema ab. Die Einfügeprozudur wird für jedes Attribut eines Datensatzes r wiederholt (Zeile 3). Zuerst wird der Identifikatoren $r.id$ unter dem aktuellen Attribut $r.a$ in den RI eingefügt (Zeile 4). Im nächsten Schritt wird geprüft, ob sich $r.a$ bereits im SI befindet (Zeile 5). Ist dies der Fall gibt es für dieses Attribut nichts weiter zu tun. Andernfalls wird für das Attribut $r.a$, über die Enkodierungsfunktion E_a ein Blockschlüssel bestimmt (Zeile 6). Anhand dessen wird das Attribut in den entsprechenden Block im Block Index eingefügt (Zeile 7). Beinhaltet der Block mehr als ein Attribut, wird zu allen bereits im Block befindlichen Attributen die Ähnlichkeit sim bestimmt (Zeile 9). Diese Ähnlichkeit wird zwei Mal in den Similarity Index eingefügt. Zunächst für das Tupel $(r.a, sim)$, unter dem verglichenen Attribut v (Zeile 10) und anschließend das Tupel (v, sim), unter dem Attribut des einzufügenden Datensatzes $r.a$ (Zeile 11). Die Schritte der Zeilen 4-12, werden für jedes Attribut jedes Datensatzes wiederholt. Nachdem alle Datensätze in RI, BI und SI eingefügt wurden, werden diese drei Indicies zurückgegeben.

Query Phase. Zur Beantwortung einer Anfrage werden ebenfalls alle Attribute seperate betrachtet (Zeile 2). Zuerst wird die Ergebnisliste M initalisiert. Für jedes Attribut wird geprüft, ob dieses bereits im RI existiert (Zeile 3). Ist dies der Fall wird lediglich der Datensatzidentifikator $q.id$ in den RI unter dem Attribut $q.a$ eingefügt (Zeile 6). Andernfalls wird das Attribut $q.a$ nach dem Verfahren aus Algorithmus 1 zu den Indicies hinzugefügt (Zeile 4). Dadurch werden implizit die Ähnlichkeiten zu $q.a$ zu Attributen im gemeinsamen Block berechnet. Anschließend werden aus dem RI alle Identifikatoren ausgelesen (Zeile 8), welche dasselbe Attribut wie q besitzen. Diese werden in die Ergebnisliste mit dem Ähnlichkeitswert 1 aufgenommen bzw. wenn dort schon ein Eintrag für einen Datensatz vorhanden ist, wird die Ähnlichkeit aufsummiert (Zeile 9). Danach werden zu $q.a$, im SI, die Attribute desselben Blocks und deren Ähnlichkeit sim nachgeschlagen (Zeile 11). Für jedes Attribut werden aus dem RI, die Datensätze mit dem Attribut $r.a$ selektiert (Zeile 11) und in die Ergebnisliste mit ihrer Ähnlichkeit sim aufgenommen bzw. auf Ähnlichkeit eines bestehenden Eintrags addiert.

Im Gegensatz zum Standard Blocking können Anfragen deutlich schneller beantwortet werden, da im Optimalfall keine Ähnlichkeitsberechnungen stattfinden und lediglich Werte nachgeschlagen werden müssen. Auf der negativen Seite steht hingegen der deutlich erhöhte Speicherbedarf, welcher auf das Halten der Ähnlichkeitswerte zurückzuführen ist.

Similarity-Aware Index with Local Sensitive Hashing (LSH)

Dieses Verfahren ist eine Erweiterung des DySimII durch LSH, welches von Li et al. [16] vorgestellt wurde. Die hier genutzte Variante des Local Sensitive Hashing nutzt das Minhash Verfahren. Minhashing ist eine effiziente Abschätzung der Überlappung zwei-

er Mengen, bekannt als Jaccard-Ähnlichkeit. Anhand des Minhash-Algorithmus ist es möglich, für jeden Datensatz n, Signaturen der Länge k zu generieren, dazu werden n verschiedene zufällig gewählte Hashfunktionen genutzt. Um die Wahrscheinlichkeit zu erhöhen, dass nur gleiche Paare dieselbe Signatur haben, wird eine Technik namens Banding eingesetzt. Hierzu werden l Signaturen zu einem Band zusammengefügt und damit verundet. Mehrere Bänder sind logisch gesehen eine Veroderung. Dieses Verfahren teilt sich ebenfalls in Bau- und Anfragephase auf.

Build Phase. Anstatt eines *Record Identifier Index* (RI) hat dieses Verfahren einen *LSH-Index* (LI). Zur Erzeugung des LI werden für alle Datensätze zunächst die Minhash Signaturen erzeugt und zu Bändern verundet. Danach der LI wird in zwei Schritten gebaut. Zuerst werden Datensätze, anhand der Bänder, gruppiert, welcher als Schlüssel dienen. Innerhalb der Bänder gibt es weitere Subindicies, welche als Schlüssel die Minhash Signaturen nutzen. Den jeweiligen Signaturen, innerhalb der Bänder, wird der Datensatzidentifikator zugewiesen. Durch die Trennung, von Datensätzen mit gleichem Band aber unterschiedlichen Signaturen, wird die Wahrscheinlichkeit erhöht, dass unähnliche Datensätze einen gemeinsamen Block haben. Die Schritte zum Einfügen in den Block Index bzw. den Similarity Index sind analog zum DySimII Verfahren.

Query Phase. Für die Beantwortung einer Anfrage werden zunächst für den neuen Datensatz die Minhash Signaturen und Bänder erzeugt und mit dem Datensatzidentifikator in den LSH-Index eingefügt. Danach werden die Datensatzidentifikatoren mit den gleichen Signaturen, in den gleichen Bändern, als Kandidatenmenge ausgelesen. Nun müssen die Attribute der Kandidaten aus einer Datenquelle geladen werden. Mit diesen Attributen können aus dem Similarity Index die Ähnlichkeitswerte jedes Kandidaten bestimmt werden. Die Kandidaten, welche aus dem LSH Index erhalten wurden haben allerdings nicht zwingend Attribute in denselben Blöcken im Block Index wie der Anfragedatensatz. Deshalb gibt es zu einigen Attributen keine vorausberechnete Ähnlichkeit, die aus dem Similarity Index bezogen werden kann. In diesen Fällen wird die Ähnlichkeit mit dem Wert 0 versehen, da das Berechnen zur Laufzeit zu lange dauert. Dies mindert zwar die Genauigkeit, sorgt aber für gute Latenzen.

Im Gegensatz zum DySimII ist die Berechnung des Index aufwendiger, da für jeden Datensatz die Minhash Signaturen und Bänder berechnet werden müssen. Allerdings ist die potentielle Kandidatenmenge deutlich geringer als beim DySimII, wodurch die Anfragen schneller beantwortet werden können.

DySNI

Das DySNI Verfahren von Ramadan et al. [7] ist eine dynamische Umsetzung des Sorted Neighborhood Verfahrens, aus dem statischen Blocking. Anstatt eines Arrays wird eine

Baumstruktur verwendet, um Datasätze möglichst effizient zu selektieren. Der gewählte Baum ist ein BraidedTree (BRT), welcher eine Erweiterung eines balancierter binären AVL-Baums ist. Dieser unterscheidet sich, indem zusätzlich innerhalb des Baumes jeder Knoten jeweils einen Verweis auf seinen Vorgänger und seinen Nachfolger hat. Die Sortierung erfolgt alphabetisch nach einem gewählten Sortierschlüssel. Ein Knoten besteht dabei aus einem Sortierschlüsselwert (Sorting Key Value. kurz: SKV) und einer Liste von Datensätzen mit diesem SKV.

Build Phase. Beim Einfügen eines neuen Datensatzes wird zunächst dessen SKV erzeugt. Wenn der SKV noch nicht im BRT-Baum vorhanden ist, wird ein neuer Knoten erzeugt und der Datensatzidentifikator angehängt. Ist der Knoten bereits vorhanden, wird lediglich der Datensatzidentifikator zum existierenden Knoten hinzugefügt. Zusätzlich wird der Datensatz in einen Inverted Index D eingefügt, um ihn zum Attributsvergleich mit anderen Datensätzen schnell selektieren zu können.

Query Phase. Zunächst wird der Anfragedatensatz, nach dem Vorgehen aus der Build Phase, eingefügt. Der Knoten, in welchem, der Anfragedatensatz eingefügt wurde, heißt Anfrageknoten N_q. Ausgehend von N_q wird ein Fenster über die benachbarten Knoten gespannt. Alle Datensätze, welche in Knoten innerhalb des Fensters gespeichert sind, werden als Kandidatenmenge C selektiert. Aus D werden für jeden Kandidaten die Attribute gelesen und anschließend in einem Paarvergleich mit dem Anfragedatensatz die Attributsähnlichkeiten ermittelt. Für die Erzeugung des Fensters werden vier Methoden vorgestellt, welche sich an Varianten des statischen Sorted Neighborhood Verfahrens orientieren.

- **Fixed Window Size (DySNI-f)** ist das einfachste Verfahren, bei welchem das Fenster um einen festen Wert w in Vorgänger- und Nachfolgerichtung aufgespannt wird.
- **Candidates-Based Adaptive Window (DySNI-c)** erweitert das Fenster abwechselnd in Vorgänger- und Nachfolgerichtung, solange bis eine Mindestanzahl an Kandidaten gefunden wurde.
- **Similarity-Based Adaptive Window (DySNI-s)** nutzt die Ähnlichkeit zwischen SKVs, dabei wird ein Fenster in eine Richtung solange erweitert bis die Ähnlichkeit zwischen dem SKV von N_q und dem nächsten Vorgänger bzw. Nachfolger eine Mindestähnlichkeit Δ unterschreitet.
- **Duplicate-Based wpAdaptive Window (DySNI-d)** erweitert das Fenster auf Basis der gefundener Matches, in beide Richtungen unabhängig voneinander. Dabei wird das Fenster um jeweils einen Knoten erweitert und zwischen dem Anfragedatensatz und den Datensätzen des neuen Knoten, der Ähnlichkeitswert berechnet, sowie klassifiziert, ob es sich um ein Match oder Non-Match handelt. Sinkt der An-

teil an gefunden Matches unter eine Schranke δ, wird das Fenster in diese Richtung nicht weiter vergrößert.

Damit Ähnlichkeiten zwischen den Datensätzen nicht jedes Mal neu berechnet werden müssen, wird je nach gewählter Fensterberechnung, die Ähnlichkeit der SKVs berechnet und in den beteiligten Knoten abgespeichert. Dadurch wird allerdings die Auswahl an SKVs auf Konkatenation von Attributen beschränkt. Attribute, die nicht im SKV genutzt wurden, müssen bei diesem Verfahren trotzdem jedes Mal neu berechnet werden. Des Weiteren ist auch dieses Verfahren sensitiv auf Fehler am Anfang der SKVs. Dies kann korrigiert werden, indem ähnlich zum Multi-pass Verfahren des Sorted Neighborhood Verfahrens, mehrere BRT-Bäume mit unterschiedlichen SKVs erstellt werden.

In ihrer Auswertung zeigen Ramadan et al., dass das *DySNI-d* Verfahren im BRT-Baum nicht funktioniert, weil ein Großteil der Duplikate im Anfrageknoten N_q landet und damit eine Erweiterung des Fensters nicht zustande kommt. Die besten Recall Ergenisse wurden mit dem *DySNI-s* Verfahren erreicht, da der Baum nach den SKVs sortiert wurde und dieses Fenster sich am besten aufspannt. Die beiden anderen Verfahren *DySNI-f* und *DySNI-c* erzielen, ebenfalls gute Ergebnisse. Zusätzlich haben die Verfahren den Vorteil, dass über die Fenstergröße die Anzahl der Kandidaten kontrolliert werden kann, wodurch auch die Latenzen kontrolliert werden können.

2.3.3 Blocking Schema

Die Qualität aller Verfahren, ob statisch oder dynamisch, hängt maßgeblich von der Auswahl der richtigen Blockschlüssel bzw. Sortierschlüssel ab. Ein Verbund aus mindestens einem Blockschlüssel für einen Entitätstypen nennt man Blocking Schema. Wie genau diese Schemata auszuwählen sind, wird von vielen Blocking Verfahren offen gelassen. Die meisten Verfahren schränken jedoch ein, dass zu einem Datensatz nur ein Blockschlüssel oder Sortierschlüssel erzeugt werden darf. Bei der Verwendung von Multi-pass Ansätzen werden dementsprechend verschiedene Schemata gefordert. Ein adäquates Schema zu finden ist oft, auch von Domainexperten, nur durch ausprobieren herauszufinden. Oft genutzt werden phonetische Enkodierung und Konkatenation von Attributen. Weitere Beispiele sind Q-Gramme oder Suffixe aus den vorgestellten statischen Verfahren.

DNF-Blocking Schema

Um die Probleme des manuellen Auswählens von Block- und Sortierschlüsseln zu umgehen schlagen Kejriwal & Miranker [17] ein Verfahren vor, welches ein Blocking Schema in disjunktiver Normalform erzeugt, welches aus den folgenden vier Komponenten besteht.

Indizierungsfunktion. Das kleinste Element eines Blocking Schema ist eine *Indizierungsfunktion* $h_i(x_t)$. Diese akzeptiert einen Attributswert x_t eines Datensatzes und erzeugt eine Menge Y, welche 0 oder mehr Blockschlüssel (engl. *blocking key value*, kurz BKV) beinhaltet. Ein BKV identifiziert einen Block, welchem ein Datensatz zugeordnet wird. Ein Beispiel einer Indizierungsfunktion ist `Tokens`. Mit der Funktion Tokens wird ein Eingabestring, beispielsweise 'Marios Pizza', in eine Menge von Token mittels eines Trennzeichens getrennt, beispielsweise durch eine Leerzeichen in $Y = \{$'Marios', 'Pizza'$\}$.

General Blocking Predicate. Das allgemeine Blockingprädikat (engl. *general blocking predicate*) $p_i(x_{t_1}, x_{t_2})$ nimmt zwei Attribute unterschiedlicher Datensatze t_1 und t_2 und nutzt die i$^{\text{te}}$ Indizierungsfunktion, um zwei Mengen von BKV Y_1 und Y_2 zu erzeugen. Das Prädikat ist wahr, wenn beiden Mengen eine gemeinsame Schnittmenge haben $Y_1 \cup Y_2 \neq$. Angenommen die i$^{\text{te}}$ Indizierungsfunktion ist `Tokens`, dann ist das zugehörige Prädikat `EnthältGemeinsamenToken`, welches Wahr ist, wenn zwei Attribute mindestens einen gemeinsamem Token haben. Beispielsweise $p_{egt}('Marios\ Pizza','Tonys\ Pizza') = Wahr$, weil $Y_1 = \{$'Marios', 'Pizza'$\}$ und $Y_2 = \{$'Tonys', 'Pizza'$\}$ woraus folgt das $Y_1 \cup Y_2 = \{$'Pizza'$\}$ und damit ist das Prädikat erfüllt.

Specific Blocking Predicate. Das spezifische Blockingprädikat (engl. *specific blocking predicate*) ist ein Paar (p_i, f), dass ein allgemeines Blockingprädikat p_i mit einem Attribute f verbindet. Das spezifische Blockingprädikat nimmt dazu zwei Datensätze t_1 und t_2 und wendet p_i auf die entsprechenden Attribute f_1 und f_2 der Datensätze an. Ein solches Paar ist beispielsweise (`EnthältGemeinsamenToken`, `PLZ`). Dieses spezifische Prädikat ist wahr, wenn die Postleitzahl zweier Datensätze einen gemeinsamem Token haben.

DNF Blocking Schema. Das DNF Blocking Schema f_P ist eine Funktion, welche in der Disjunktiven Normalform ohne Negation durch eine Menge von P Ausdrücken erzeugt wird. Jeder Ausdruck in f_P besteht aus mindestens einem spezifischen Blockingpredikat, beispielsweise (`EnthältGemeinsamenToken`, `Name`) $\vee$ (`ExakteÜbereinstimmung`, `Stadt`). Mehrere spezifische Blockingpredikat in einem Ausdruck werden durch eine Konjunktion verbunden. Das DNF Blocking Schema ist dementsprechend Wahr, wenn einer seiner Ausdrücke Wahr ist. Ist ein Blocking Schema für zwei Datensätze Wahr, haben beide mindestens einen gemeinsamen Blockschlüsselwert. Für den Blockschlüssel entspricht die Konjunktion eines Ausdrucks dem Konkatenieren von Strings. Gegeben sei folgendes Blocking Schema f_P = (`ExakteÜbereinstimmung`, `Name`) $\wedge$ (`ExakteÜbereinstimmung`, `Stadt`) und der Datensatz $r =$ ('Peter', 'Frankfurt'). Daraus erzeugt f_P den Blockschlüssel 'PeterFrankfurt'. Die Disjunktion der Ausdrücke kann bei vielen Verfahren als Multi-pass Ansatz implementiert werden. Des Weiteren ist zu beachten, dass das Blocking Schema potentiell mehrere Schlüssel pro Datensatz erzeugt.

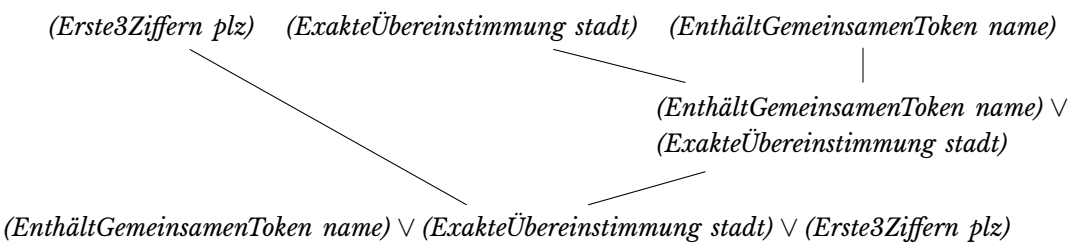

Abbildung 2.4 Konjunktion der drei Ausdrücke (`EnthältGemeinsamenToken`, `name`), (`ExakteÜbereinstimmung`, `stadt`) und (`Erste3Ziffern`, `plz`) zu einem zweistelligen und dreistelligen Ausdruck.

Lernen des DNF-Blocking Schema

Bevor das Verfahren von Kejriwal & Miranker [17] automatisiert ein Blocking Schema bestimmen kann, wird angenommen, dass eine Menge von bekannten Matches und Non-Matches vorhanden ist. Der erste Schritt zum Lernen eines DNF-Blocking Schema ist, eine Liste an spezifischen Blockingprädikaten zu benennen. Beispielsweise (`EnthältGemeinsamenToken`, `name`), (`ExakteÜbereinstimmung`, `stadt`) für Stringattribute und (`Erste3Ziffern`, `Postleitzahl`) für ein nummerisches Attribut. Anschließend wird für jedes Paar, der Matches bzw. Non-Matches, ein boolscher Featurevektor pro spezifischem Blockingprädikat erzeugt. Ein Wert ist Wahr, wenn ein Paar das Prädikat erfüllt. Dabei wird die Menge positiver Vektoren mit P_f und negativer Vektoren mit N_f bezeichnet.

Anschließend werden die Ausdrücke des Blocking Schema erzeugt. Da exponentiell viele Ausdrücke erzeugt werden können, muss der Anwender die Tiefe, d.h. Anzahl der Prädikate pro Ausdruck, beschränken. Abbildung 2.4 zeigt wie aus den Einzelausdrücken (`EnthältGemeinsamenToken`, `name`), (`ExakteÜbereinstimmung`, `stadt`) und (`Erste3Ziffern`, `plz`) ein zweistelliger und ein dreistelliger Ausdruck erzeugt werden. Zwei Ausdrücke a_1 und a_2 mit ihren Featurevektoren P_{f,a_1}, N_{f,a_1} und P_{f,a_2}, N_{f,a_2} werden zu einem neuen Ausdruck a_{1-2} konjugiert, indem die Vektoren verundet werden $P_{f,a_1-2} = P_{f,a_1} \wedge P_{f,a_2}$, respektive $N_{f,a_{1-2}} = N_{f,a_1} \wedge N_{f,a_2}$. Die teuren Prädikatsoperationen müssen dadurch lediglich auf die einstelligen Ausdrücken anwendet werden. Die Qualität der einzelnen Ausdrücke wird im Anschluss bewertet. Dazu nutzen Kejriwal & Miranker die Fisher-Score. Die Idee der Fisher-Score, nach [18], ist, eine Untermenge von Features (hier: Ausdrücke) zu finden, sodass die Datenpunkte der Klassen (hier: Matches und Non-Matches) möglichst weit voneinander entfernt und gleichzeitig die Datenpunkte innerhalb der Klasse möglichst nahe zusammen sind. Die Formel zur Berechnung der Fisher-Score des i^{ten} Ausdrucks sieht folgendermaßen aus

$$\rho_i = \frac{|P_{f,i}|(\mu_{p,i} - \mu_i)^2 + |N_{f,i}|(u_{\mu,i} - \mu_i)^2}{|P_{f,i}|\sigma_{p,i}^2 + |N_{f,i}|\sigma_{n,i}^2},$$

Algorithm 3 DisjunctiveBlockingScheme()

Input:
- Set of positively labeled feature vectors P_f
- List of conjuctive Terms T
- Maximum positive pairs that may be left uncovered ϵ

Output:
- Disjunctive Blocking Scheme DBS

1: Initialize set DBS to be empty
2: Initialize vector P_{dbs} as NULL vector
3: **for** term $t \in T$ **do**
4: Calculate the Fisher-Score
5: **end for**
6: Sort T according to scores
7: **while** more than ϵ Matches in P_{DBS} are uncovered **do**
8: Let i be feature in T with highest Fisher-Score
9: Remove i from T
10: **if** $P_f, i \vee P_{dbs}$ covers at least 1 new pair **then**
11: Add i to DBS
12: $P_{dbs} = P_{f,i} \vee P_{DBS}$
13: **end if**
14: **end while**
15: Return disjunction of Terms DBS

dabei ist $\mu_{p,i}$ bzw. $\mu_{n,i}$ der Anteil der wahren Werte in $P_{f,i}$ und $N_{f,i}$. Weiterhin ist μ_i der Anteil wahrer Werte in $P_{f,i}$ und $N_{f,i}$ zusammen und σ ist die positive bzw. negative Varianz.

Algorithmus 3 beschreibt vereinfacht die Auswahl des DNF Blocking Schema. Zunächst werden die Ausdrück anhand der Fisher-Score bewertet (Zeile 4). Die Liste der Ausdrücke T wird anschließend nach ihrer Fisher-Score sortiert (Zeile 6). Das zusammenfügen des Disjunktiven Blocking Schema erfolgt durch hinzugefügen des jeweils besten Ausdrucks $i \in T$, solange ein Ausdruck mindestens ein weiteres Match erfasst (Zeilen 7-14). Dazu werden die positiven Featurevektoren des Ausdrucks $P_{f,i}$ und der P_{DBS} verodert. Dies wird wiederholt, bis ein Minimum an Matches noch nicht erfasst wurde oder alle Ausdrücke verarbeitet sind.

2.4 Ähnlichkeitsmaße

In Abschnitt 2.3 wurde beschrieben wie Kandidaten für einen Vergleich gruppiert und selektiert werden. Über Ähnlichkeitsmaße (engl. similarity measures) wird die Ähnlichkeit zweier Datensätze bestimmt. Genauer wird die Ähnlichkeit der einzelnen Attribute bestimmt, aus welcher sich die Gesamtähnlichkeit der Datensätze bestimmen lässt. Die meisten Fehler, die zu unterschiedlichen Datensätzen führen sind typographische Variationen von Strings. Weshalb sich entsprechend viele Ansätze für den Vergleich von Stringattributen finden. Attributsähnlichkeiten werden nach Elmagarmid et al. [19] in vier Kategorien geordnet:

- zeichenbasierend
- tokenbasierend
- phonetisch
- nummerisch

zusätzlich werden noch die kernelbasierend Methoden betrachtet.

Zeichenbasierende Ähnlichkeit

Wie sich die Ähnlichkeit von Strings bestimmen lässt, wird seit den 60er Jahren [20] intensiv erforscht. Die Stärke von zeichenbasierten Ähnlichkeiten sind das Erkennen von typografischen Fehlern.

Der älteste und wohl auch bekannteste Algorithmus ist die Levenshtein Distanz [20]. Die Levenshtein Distanz ist eine **Editierdistanzen**, welche die minimalen Schritte berechnet, die benötigt werden um einen String σ_1 in einen anderen σ_2 umzuwandeln. Diese Schritte beinhalten das Einfügen, das Löschen, das Ersetzen und mit der Modifikation von Damerau [21] auch das Vertauschen von Zeichen. Je weniger Schritte für die Transformation notwendig sind, desto ähnlicher sind zwei Strings. Da potentiell alle Zeichen beider Strings miteinander vergleichen werden ist die Komplexität $O(|\sigma_1| \cdot |\sigma_2|)$. Needleman und Wunsch [22] erweitern die originale Editierdistanz dahingehend, dass bestimmte Operationen anders gewichtet werden. Dazu können die Kosten für die einzelnen Schritte, welche in der einfachsten Form 1 sind, auf einen beliebigen Gleitkommawert angepasst werden. Beispielsweise können Transpositionen, welche häufig einen Tippfehler darstellen, niedriger gewichtet werden. In dieser Variante entspricht das Lösen der Editierdistanz dem Traveling Salesmen Problem und ist daher NP-schwer.

Eine weitere Modifikation der Editierdistanz ist es Lücken zu erkennen [23], beispielsweise wenn ein Wort abgekürzt wurde, etwa *John R. Smith* statt *Johnathan Richard Smith*. Dementsprechend können Kosten für das Öffnen und das Erweitern einer Lücke festgelegt werden. Eine andere ist es, Fehler am Anfang oder am Ende des String mit geringeren Kosten zu versehen [24].

Eine weitere gängige Alternative ist die Jaro-Distanz, welche die Anzahl der gemeinsamen Zeichen m berechnet, wobei eine Verschiebung von $\frac{1}{2} \cdot min(|\sigma_1|, |\sigma_2|)$ zugelassen wird. Von den gemeinsamen Zeichen werden anschließend die Transpositionen t berechnet, d.h. wie viele gemeinsame Zeichen nicht in der gleichen Reihenfolge sind. Daraus berechnet sich die Jaro-Distanz $d_j = \frac{1}{3}\left(\frac{m}{|s_1|} + \frac{m}{|s_2|} + \frac{m-t}{m}\right)$.

Tokenbasierende Ähnlichkeit

Die tokenbasierende Ähnlichkeit bietet, im Gegensatz zu den zeichenbasierenden, den Vorteil, dass Vertauschungen von Wörtern erkannt werden, beispielsweise bei Vornamen und Nachnamen.

Monge und Elkan [25] schlagen einen Algorithmus auf Basis atomarer Strings vor. Für zwei Strings σ_1, σ_2 werden alle Token aus $\sigma_1 = (\sigma_{1_{t_1}}, \ldots, \sigma_{1_{t_i}})$ mit allen Token aus $\sigma_2 = (\sigma_{2_{t_1}}, \ldots, \sigma_{2_{t_j}})$ vergleichen. Zum Vergleich wird eine beliebige zeichenbasierende Ähnlichkeit sim gewählt werden. Dieser Algorithmus kann dadurch auch typographische Fehler erkennen. Anschließend wird für jeden Token t in σ_1 die Ähnlichkeit mit $s_t = max(sim(\sigma_{1_t}, \sigma_{2_{t_1}}), \ldots, sim(\sigma_{1_t}, \sigma_{2_{t_j}}))$ berechnet. Die Gesamtähnlichkeit ist der Durchschnitt der Tokenähnlichkeiten $m = avg(s_{t_1}, \ldots, s_{t_i})$. Das Problem dieses Algorithmus ist seine Komplexität, welche quadratisch zur Tokenmenge und damit zu sim ist.

Eine weitere Möglichkeit Token zu bilden sind Q-Gramme. Diese erlauben es ebenfalls, neben Vertauschungen von Wörtern, auch typographische Fehler zu erkennen. Über Q-Gramme wird ein String σ in k überlappende Token der Länge n zerlegt. Dazu wird ein Fenster der Länge n von Position 1 bis $|\sigma| - (n-1)$ geschoben. Für den Namen `Thomas` und $n = 3$ werden die Q-Gramme {`Tho`, `hom`, `oma`, `mas`} gebildet. Um aus Q-Grammen eine Ähnlichkeit zu bestimmen gibt es viele Möglichkeiten, beispielsweise Anteil der übereinstimmenden Q-Gramme.

Eine deutlich einfachere und schnellere Möglichkeit die Ähnlichekit von Token zu berechnen, ist der *Jarcard-Koeffizienten*. Dieser gibt die Ähnlichkeit zweier Mengen A, B mit $J(A,B) = \frac{|A \cap B|}{|A \cup B|}$ an. Ein ähnliches Maß bietet der *Simpson-Koeffizienten*, welcher die Überlappung zweier Mengen $S(A,B) = \frac{|A \cap B|}{\min(|A|,|B|)}$ bestimmt.

Ein weiteres Vorgehen auf Basis atomarer Strings ist WHIRL von Cohen [26]. Es kombiniert die Kosinus-Ähnlichkeit mit dem TF/IDF Gewichtungsschema. Dabei ist TF die Vorkommenshäufigkeit (engl. term frequency) und gibt an wie häufig ein Token in einem String vorkommt. IDF ist die Inverse Dokumenthäufigkeit (engl. inverse document frequency) und gibt an wie oft ein Token in den bekannten Strings eines Vokabulars D vorkommnt. Für alle atomaren Strings w wird ein Gewicht berechnet

$$\nu_\sigma(w) = log(tf_w + 1) \cdot log(idf_w).$$

Die Kosinus-Ähnlichkeit zweier String σ_1, σ_2 ist dementsprechend definiert als

$$sim(\sigma_1, \sigma_2) = \frac{\sum_{i=1}^{|D|} \nu_{\sigma_1}(j) \cdot \nu_{\sigma_2}(j)}{\|\nu_{\sigma_1}\| \|\nu_{\sigma_2}\|}$$

Mit WHIRL ist es möglich vertauschungssicher die Ähnlichkeit von Strings zu bestimmen. Ein großer Vorteil dieses Algorithmus ist, dass nach erheben des TF/IDF Index, die Ähnlichkeit in $O(1)$ berechnet werden kann, da lediglich Werte nachgeschlagen werden müssen. Dem entgegen steht, dass typographische Fehler nicht erkannt werden. Deshalb haben Gravano et. al [27] WHIRL erweitert und nutzen statt atomare Strings Q-Gramme. Dadurch lassen sich bei gleicher Komplexität auch Rechtschreibfehler erkennen, da ein

Großteil der Q-Gramme gleich ist. Allerdings geschieht dies zu Ungunsten von Speicherkosten, da der TF/IDF Index dementsprechend größer wird.

Phonetische Ähnlichkeit

Die phonetische Ähnlichkeit ist sowohl zeichen- als auch tokenbasiert. Es werden jedoch nicht die Zeichen oder Token miteinander verglichen, sondern die Sprechlaute von Wörtern. So ist es möglich gleich gesprochene Wörter mit unterschiedlicher Schreibweise zu finden. Dies ist vor allen Dingen bei Namen sehr nützlich, da es hier eine besonders hohe Dichte an gleich klingenden Worten mit kleinen Variationen in der Schreibweise gibt. Phonetische Enkodierungsschemata funktionieren allerdings meist nur für eine Sprache oder einen Akzent. Bekannte Algorithmen für die englische Sprache sind Soundex und NYSIIS, sowie Metaphone und Double Metaphone, welche sich zum Teil auch auf nicht englische Sprachen anwenden lassen. Ein Methode für die deutsche Sprache ist die Kölner Phonetik.

Nummerische Ähnlichkeit

Während es für Strings eine Vielzahl ein Vergleichmöglichkeiten gibt ist die Anzahl bei den nummerischen Überschaubar. Die offensichtlichste Methode ist, eine Nummer als String zu behandeln und entsprechende Algorithmen zu verwenden. Andere eher primitive Vorgehensweisen sind etwa, die ersten n oder letzten m Ziffern miteinander zu vergleichen, beispielsweise bei einer Postleitzahl. Für Mengenangaben zwischen zwei Werten n_1 und n_2 kann die maximale absolute Differenz genutzt werden $sim_{d_{max}} = 1.0 - \left(\frac{|n_1 - n_2|}{d_{max}}\right)$.

Kernel Ähnlichkeit

Anhand zweier gegebener Strings gibt es keine offensichtliche Antwort auf die Frage: Wie ähnlich sind σ_1 und σ_2? Im Gegensatz dazu kann dies für Vektoren in $\mathbb{R}^d$ eindeutig, beispielsweise über die Kosinus-Ähnlichkeit $\frac{\sigma_1 \cdot \sigma_2}{\|\sigma_1\| \|\sigma_2\|}$ berechnet werden [28]. Kernelfunktionen werden unter anderem in Support Vector Machines (SVM) eingesetzt. Diese Kernel weisen eine Reihe statistisch interessanter Eigenschaften auf, beispielsweise dass ihre Performanz unabhängig von der Dimensionalität ist, auf welcher die Berechnung stattfindet. Dadurch ist es möglich in hohen Dimensionalitäten zu arbeiten ohne Überanzupassen [29].

Der einfachste und meist genutzte String Kernel ist der Bag-of-Words Kernel. Dabei wird die Anzahl der vorkommenden Worte in σ gezählt und in einem dünnbesetzten Vektor, über die Menge aller bekannten Worte aller bekannten Strings, erzeugt. Wie bereits be-

kannt sind atomare String anfällig für typographische Fehler. Aus diesem Grund gibt es auch Variationen des Kernels, der Q-Gramme nutzt, um dieses Problem zu umgehen.

Lodhi et al. stellen eine Kernelfunktion vor, um Strings im Feature Space miteinander zu vergleichen, ohne diese vorher in Vektoren zu zerlegen. Der sogenannte String Subsequence Kernel (SSK) vergleicht Strings, indem er Stringvektoren erzeugt, welche einen bestimmten Substring beinhalten oder nicht. Dabei wird jedes Vorkommen eines Substrings anhand der Übereinstimmung gewichtet. Die Übereinstimmung erlaubt beispielsweise auch Lücken, sodass der Substring 'c-a-r' in den beiden Wörtern '**car**d' und '**cust**a**r**d' mit unterschiedlicher Gewichtung vorkommt.

2.5 Klassifikatoren

Die Aufgabe von Klassifikatoren oder Matching-Strategien (vgl. Köpcke & Rahm [2]) ist es, Datensatzpaare in zwei Mengen Matches und Non-Matches kategorisieren. Im Gegensatz zu den Attributesähnlichkeitsmaßen bewerten diese einen kompletten Datensatz, welcher im Normalfall aus mehreren Attributen besteht. Klassifikatoren können nach Elmagarmid et al. [19] in zwei Kategorien einordnen werden.

- Vorgehen, die *Trainingsdaten* benötigen um zu Lernen welche Datensätze übereinstimmen. Hierzu gehören überwachte, semi-überwachte, aktive und unüberwachte Lernstrategien.
- Vorgehen, die *Domänenwissen* oder *generische Distanzmaße* nutzen um Übereinstimmungen zu finden.

2.5.1 Distanzbasierende Verfahren

Nachdem die Ähnlichkeit zwischen den Attributen der Kandidatenpaaren berechnet wurden, gibt es für jedes Kandidatenpaar (t_1, t_2) einen Ähnlichkeitsvektor $(sim(t_{1,a_1}, t_{2,a_1}), \ldots, sim(t_{1,a_n}, t_{2,a_n}))$ über die Attribute $a_1, \ldots, a_n$ beider Datensätze. Dieser Vektor wird von den nachfolgenden Verfahren genutzt, um eine Match-Entscheidung zu treffen.

Schwellenwertbasierend

Die naivste Art und Weise um zu klassifizieren sind nach Christen [30, Kap. 6] Schwellenwerte. Hierfür werden die Ähnlichkeitsvektoren zu einer Gesamtähnlichkeit g aufsummiert. Anschließend werden je nach Ausprägung bis zu zwei Schwellen festgelegt. In der Variante mit einer Schwelle t, werden die Kandidatenpaare in zwei Klassen mit $g \geq t$

als Matches und $g < t$ als Non-Matches klassifiziert. Werden zwei Schranken $t1$ und $t2$ genutzt, wird in drei Klassen gegliedert, Matches mit $g \geq t1$, Non-Matches mit $g \leq t2$ und zusätzlich gibt es noch den Bereich $t2 < g < t1$, welcher ein potentielles Match bedeutet und manuelle klassifiziert werden muss. Der Nachteil dieser Methodik ist, dass beim Mitteln des Vektors alle Attribute mit gleichem Gewicht zum endgültigen Wert beitragen. Dadurch wird die Wichtigkeit der unterschiedlichen Attribute und ihre Wertstellung innerhalb des Datensatzes verworfen. Um dem entgegenzuwirken, können für jedes Attribut Gewichte vergeben werden, mit welchen die Wertstellung der Attribute angegeben werden kann. Dennoch gehen beim Aufsummieren von Ähnlichkeiten detaillierte Informationen über die einzelnen Ähnlichkeiten verloren.

Regelbasierend

Christen [30, Kap. 6] beschreibt die regelbasierende Klassifikation als Anwendung der Prädikatenlogik erster Stufe (PL1). Dabei wird ein Klassifikationspredikat in konjunktiver Normalfrom mit disjunktiven Ausdrücken geschrieben.

$$((sim(name)[r_i, r_j] \geq 0.9) \wedge (sim(stadt)[r_i, r_j] = 1.0)) \\ \vee (sim(plz)[r_i, r_j] \geq 0.7) \implies [r_i, r_j] \rightarrow Match \tag{2.1}$$

Formel 2.1 zeigt eine beispielhafte Regel, wobei $sim(\cdot)$ einer Attributsähnlichkeit entspricht und r_i, r_j zwei Datensätze sind. Diese Regel klassifiziert ein Paar als Match, wenn entweder der Ähnlichkeitswert für Name größer 0.9 und die Stadt gleich ist, oder der Ähnlichkeitswert der Postleitzahl größer 0.7 ist. Der Vorteil gegenüber den schwellenbasierenden Verfahren ist, dass Ausdrücke auf Attribute angewendet werden und dadurch die Informationen der einzelnen Attributsähnlichkeiten nicht verloren gehen. Mit der regelbasierten Klassifikation kann in beliebig viele Klassen kategorisiert werden. Typischerweise werden entweder zwei Klassen Match und Non-Match oder zusätzlich potentielles Match klassifiziert. Bei Match und Non-Match ist nur ein Prädikat P_m notwendig, da alle wahren Paare als Matches und alle falschen Paare als Non-Matches klassifiziert werden. Für potentielle Matches wird ein weiteres Prädikat P_{pm} benötigt, dementsprechend sind Attribute Non-Matches, wenn sowohl P_m als auch P_pm falsch ist. Für die Bestimmung der Prädikate gibt es zwei Möglichkeiten. Die erste ist einen Domänexperten das Prädikat festlegen zu lassen. Das ist allerdings ein sehr zeitintensiver Prozess, welcher bei aller Expertise, in den meisten Fällen durch ausprobieren gelöst werden muss. Die Alternative ist, ein Prädikat zu erlernen, was ähnlich zum Lernen eines Blocking Schema (vgl. Abschnitt 2.3.3) funktioniert.

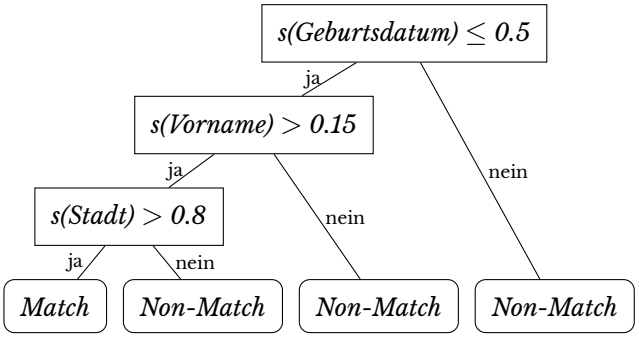

Abbildung 2.5 Beispiel eines Decision Tree. Der Baum tested an jedem Knoten den Ähnlichkeitswert eines bestimmten Attributes, durch die Funktion $s(\cdot)$. Die Blattknoten bestimmen das Klassifikationsergebnis Match oder Non-Match.

2.5.2 Überwachtes bzw. semi-überwachtes Lernen

Die Verfahren für überwachtes und semi-überwachtes Lernen benötigen eine Menge von klassifizierten Daten in der Form von Matches und Non-Matches. Anhand dieser Trainingsdaten kann ein Klassifikationsmodell erstellt werden. Das Modell für Entity Resolution muss Datensätze in die zwei Klassen Matches und Non-Matches einordnen, weshalb hierfür binäre Klassifikatoren benötigt werden. Für jedes klassifizierte Paar berechnet ein Klassifikator, anhand des trainierten Modells, die Wahrscheinlichkeit zwischen 0 % und 100 %, dass ein Paar ein Match ist. Die Entscheidung, ob ein Paar Match oder Non-Match ist, wird anhand einer Wahrscheinlichkeitsschwelle getroffen. Liegt diese beispielsweise bei 50 %, sind Paare mit einer Wahrscheinlichkeit größer Matches und Paare mit einer Wahrscheinlichkeit kleiner Non-Matches.

Decision Trees

Decision Trees sind als Klassifikatoren sehr beliebt, da ihre Funktionsweise anschaulich ist. Zudem kann ein Modell übersichtlich visualisiert werden, sodass es intuitiv, auch von Laien, interpretiert werden kann. Ähnlich zu dem regelbasierten Verfahren prüft auch der Decision Tree den Ähnlichkeitswert eines bestimmten Attributes, welches einem Wert im Vektor entspricht. Dementsprechend kann das Modell eines Decision Trees direkt in einem Prädikat formuliert werden. Abbildung 2.5 zeigt ein Beispiel eines Decision Tree. An jedem Knoten, ausgehend von Wurzelknoten, wird der Ähnlichkeitswert eines bestimmten Attributes über die Funktion $s(\cdot)$ geprüft. Weißt das Geburtdatum zweier Datensätze eine Ähnlichkeit kleiner 0.5 auf wird diese durch den darauffolgeden Blattknoten als Non-Match klassifiziert. Damit ein Blattknoten einen Datensatz als Match klassifiziert müssen zusätzlich noch der Vorname ähnlicher als 0.15 und die Stadt ähnlicher als 0.8 sein.

Name	*Address*	*City*	*Cuisine*
Fenix	8358 Sunset Blvd. West	Hollywood	American
Fenix at the Argyle	8358 Sunset Blvd.	W. Hollywood	French (new)

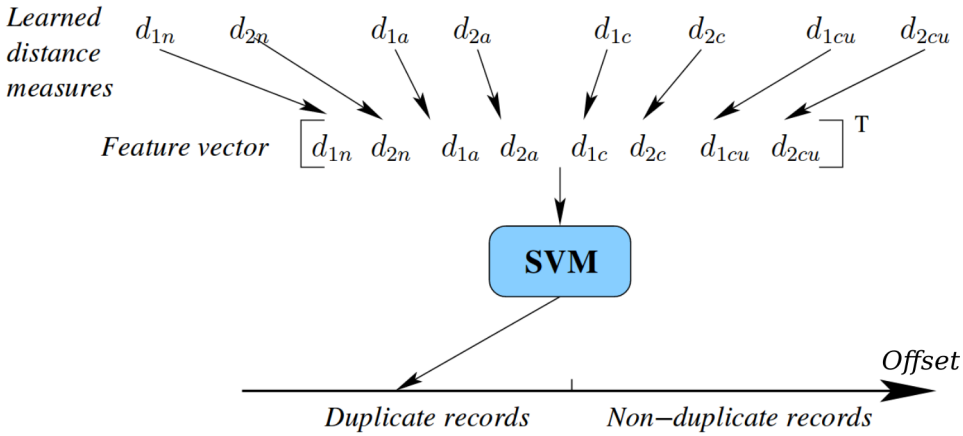

Abbildung 2.6 Datensatzklassifikation nach [32]. Der Featurevektor für die Klassifikation wird aus den Attributsähnlichkeiten von Name, Address, City und Cuisine erzeugt. Eine SVM klassifiziert diesen Vektor anschließend in Match (duplicate records) oder Non-Match (Non-duplicate records).

Support Vector Machines

Support Vector Machines wurden von Boser et al. [31] eingeführt. In ihrer einfachsten Form lernen SVMs eine separierende Hyperebene zwischen einer Menge von Punkten, welche den Abstand zwischen der Hyperebene und den nähesten Punkten der jeweiligen Klassen maximiert. Eine Kernelfunktion berechnet das innere Produkt zwischen Punkten im Hyperraum (Feature Space) [29].

Bilenko & Mooney [32] stellen ein Lernverfahren auf Basis der TF/IDF Ähnlichkeit von Cohen vor, welches einen SVM-Klassifikator nutzt. Dazu wird ein Vektor erzeugt, indem die Kosinus-Ähnlichkeit zwischen den Attributen eines Datensatzpaares berechnet wird. Dabei werden die Komponenten der bekannten Summe $\frac{x_i \cdot y_i}{\|x\|\|y\|}$, welche zum i$^{\text{ten}}$ Element des Vokabulars gehören, einem d-dimensionalen Vektor zugeordnet $\mathbf{p}(x, y) = \langle \frac{x_i \cdot y_i)}{\|x\|\|y\|} \rangle$. In Abbildung 2.6 werden zwei Datensätze gezeigt. Jedes Attributspaar ist dabei ein Teil eines Vektors. Die Vektoren werden dann von einem SVM-Model in Matches und Non-Matches klassifiziert. Trainiert wird das SVM-Model anhand von Match und Non-Match Vektoren.

Christen [33] erweitert dieses Verfahren, indem mehrere SVM Modelle trainiert werden. Die initiale SVM wird mit der Mengen der offensichtlichen Matches und Non-Matches der Gesamttrainingsmenge trainiert. Bei offensichtlichen Matches sind die Werte der Vektoren sehr nahe an 1 und bei offensichtlichen Non-Matches sehr nahe bei 0. Alle

nicht offensichtlichen Vektoren der Trainingsmenge werden anschließend mit der initialen SVM klassifiziert. Je nach Klassifizierungsergebnis werden diejenigen, die am weitesten von der seperierenden Hyperebene entfernt sind, zur Menge der offensichtlichen Matches bzw. Non-Matches hinzugefügt. Die zweite SVM wird dann mit den erweiterten Trainingsmengen trainiert. Diese Schritte werden solange wiederholt, bis ein Stopkriterium erfüllt ist.

2.5.3 Aktives Lernen

Ein großer Nachteil der überwachten Lernverfahren ist, dass die Trainingsmenge viele Beispiele benötigt und das diese repräsentativ für die zu Gesamtmenge von Entitäten sein muss. Wie Trainingsdaten akquiriert werden wird in Abschnitt 2.6 diskutiert. Als Alternative dazu gibt es die aktiven Lernverfahren, welche initial nur eine sehr kleine Trainingsmenge (*seed*) benötigen. Auf Basis des initialen Models werden in Interaktion mit einem erfahrenen Benutzer Datensatzpaare selektiert, die helfen das Klassifikationsmodell zu verbessern. Eine initiales Modell kann relativ einfach über offensichtliche Matches und Non-Machtes erzeugt werden. Anschließend ist aktives Lernen ein iterativer Prozess. Zunächst wird die Trainingsmenge mit dem Modell klassifiziert. Aus der Menge klassifizierte Daten werden die Interessantesten ausgewählt, die manuelle von einem Benutzer klassifiziert werden. Anschließend werden diese zu den initialen Daten hinzugefügt und es wird ein neues Modell trainiert. Diese Schleife wird solange wiederholt bis ein Stopkriterum (Anzahl von Iterationen oder minimale Genauigkeit) erreicht wurde.

Arasu et al. [34] kombinieren eine aktives Lernvorgehen mit einem Blockingmechanismus, welcher entweder mit einem Decision Tree oder einer SVM funktioniert. Dabei gibt der Benutzer als Stopkriterium die Mindestpräzision (siehe Abschnitt 2.6) an, welcher das finale Modell entsprechen muss. Der Lernprozess versucht dann ein Modell zu finden, welches einen hohen Recall liefert und gleichzeitig die Anzahl der manuell zu klassifizierenden Paare gering hält.

2.6 Messen von Qualität- und Effizienz[2]

Aus den bis hier vorgestellten Verfahren zu Entity Resolution stellt sich die Frage: Wie kann die Qualität und Komplexität dieser Verfahren gemessen werden? Die Messung dient dazu, ein Verfahren zu bewertet und gleichzeitig eine Vergleichbarkeit zwischen anderen Verfahren herzustellen. Damit die Qualität und die Komplexität der ER-Verfahren überprüft werden kann, ist es unerlässlich über die Ground Truth

[2] Dieser Abschnitt bezieht sich auf Analysen und Erklärungen zu Qualität und Komplexität von Entity Resolution Systemen aus Christen [30].

Daten (auch Gold Standard Daten) zu verfügen. Die Ground Truth beschreibt eine Menge gelabelter Daten, welche einer oder mehreren Klassen angehören. Für Entity Resolution sind die Daten, Datensatzpaare und die Klassen Matches und Non-Matches. Damit die Ground-Truth repräsenativ für die zu überprüfenden Daten ist, sollte diese möglichst deren Charakteristik wiederspiegeln. Daraus entsteht die nächste Frage: Woher kommen die Ground Truth Daten?

- Wird versucht einen entwickelten Algorithmus/Verfahren zu bewerten, dann empfiehlt es sich einen der frei verfügbaren Datensätze zu nehmen, zu welchen bereits Ground Truth Daten existieren und beispielsweise von Wissenschaftlern oder Domainexperten manuell klassifiziert wurden. Das Problem ist allderings, dass viele dieser Datensätze nur wenige Einträge (meist < 10.000) und daher kaum Bezug zu Realdaten haben. Diese Datensätze werden in Abschnitt 5.1.3 diskutiert.

- Soll ein Verfahren auf Daten einer Domäne angewendet werden, zu welcher keine Ground Truth existiert, müssen diese manuell erzeugt werden. Datensatzpaare werden dabei zufällig erzeugt und müssen anschließend von einem Prüfer in Matches und Non-Matches klassifiziert. Ein großer Nachteil dieser Methode ist es, dass selbst wenn ein Blockingverfahren angewendet wurde, dass die Anzahl der zu klassifizierenden Paare riesig ist. Hinzu kommt, dass die Menge der Matches nur einen Bruchteil der Paare betrifft, weshalb die Ground Truth ein deutliches Ungleichgewicht aufweisen wird. Ein weiteres Problem ist, dass in diesem Prozess Fehler gemacht werden. Dabei entstehen die Fehler nicht bei den offensichtlichen Match und Non-Matches, sondern bei den Paaren, die auch für den Menschen nur schwer zu bewerten sind. Des Weiteren kann es zu unterschiedlichen Klassifizierungen je nach Prüfer kommen und auch derselbe Prüfer kann je nach Gemütslage und Konzentrationslevel unterschiedliche Aussagen über dasselbe Paar treffen.

Vogel et al. [35] haben deshalb einen sogenannten *Annealing Standard* entwickelt, welcher das Erstellen einer Ground Truth über einen iterativen Prozess vereinfachen sollen. Dabei wird zunächst mit einem Klassifikatoren eine Baseline erzeugt, die den Annealing Standard darstellt. Anschließend werden mit einem weiteren Klassifikatoren, welcher der vorherige mit anderen Parametern sein kann, Paare erzeugt und mit der Baseline verglichen. Die Übereinstimmung der beiden bildet den neuen Annealing Standard. Die übrigen Paare werden zur manuellen Inspektion Prüfern vorgelegt, die dadurch erzeugten Matches und Non-Matches werden mit dem Annealing Standard verschmolzen. Diese Iteration wird solange wiederholt, bis das Delta der Klassifikatoren einen bestimmten Maximalwert an Paaren unterschreitet.

Algorithm 4 WeakTrainingSet w\o Ground Truth

Input:
- Dataset: D
- Upper Threshold: ut
- Lower Threshold: lt
- Blocking Window Size: c
- Maximum Duplicate Pairs: max_p
- Maximum Non-Duplicate Pairs: max_n

Output:
- A set of positive samples: P
- A set of negative samples: N

1: Initialize set $P = ()$, set $N = ()$
2: Initialize set of tuple pairs $C = ()$
3: Generate TFIDF statistics of D
4: **for** fields $f \in D$ **do**
5: **for** records $r \in D$ **do**
6: Tokenize r_f into BKV_f
7: Block r on generate tokens for field f
8: **end for**
9: **end for**
10: **for** block B generate in previous step **do**
11: Slide a window of size c over tupels in B
Generate all possible pairs within window and
add to C
12: **end for**
13: **for** pairs $(t_1, t_2) \in C$ **do**
14: Compute TFIDF similarity sim of (t_1, t_2)
15: **if** $sim \geq ut$ **then**
16: **if** $|P| < d$ **then**
17: add (t_1, t_2) to P
18: **else if** $sim >$ lowest sim in P **then**
19: Replace pair with lowest sim in P with (t_1, t_2)
20: **end if**
21: **end if**
22: **if** $sim < lt$ **then**
23: **if** $|N| < nd$ **then**
24: add (t_1, t_2) to N
25: **else if** $sim >$ lowest sim in N **then**
26: Replace pair with lowest sim in N with (t_1, t_2)
27: **end if**
28: **end if**
29: **end for**
30: Return P and N

Ein weiteres Verfahren haben Kejriwal & Mirankern [17] entwickelt. Ihre Idee ist es, eine Menge schwach klassifizierter Datenpaare zu generieren, dabei werden sowohl positive (Matches), als auch negative (Non-Matches) Datensatzpaare klassifiziert. Über zwei Schranken kann der Benutzer festlegen wie ähnlich (obere Schranke ut) bzw. wie verschieden (untere Schranke lt) die Paare sein sollen. Paare größer ut werden als Matches und Paare kleiner lt als Non-Matches klassifiziert. In Algorithmus 4 wird die Funktionsweise erklärt. Damit vor allem die erzeugten Ground Truth Non-Matches, der klassifizierten Paare, nicht beliebig groß werden, kann der Anwender festlegen, wie viele Matches max_p bzw. Non-Matches Paare max_n maximal erzeugt werden sollen. Zunächst wird die TF/IDF Statistik über D erzeugt (Zeile 3), welche für einen späteren Paarvergleich benötigt wird. Die in diesem Algorithmus genannten $fields$ beschreiben die Positionen eines Attributes im Tupel eines Datensatzes, beispielsweise für ein Tupel (`Kevin`, `Sapper`, `Hochschule RheinMain`) hat der Nachname die Position 2. Anschließend wird ein Blocking der Daten per Standard Blocking und Sorted Neighborhood durchgeführt. Jeder Datensatz wird (pro Attribute) in Token zerlegt (Zeilen 4-9). Anhand der Token wird das Standard Blocking durchgeführt, wobei jeder Datensatz in mehreren Blöcken vertreten sein kann. Die Menge von Blöcken ist jeweils nach Attributen gruppiert, sodass Token unterschiedlicher Attribute nicht als Blockschüssel desselben Blockes genutzt werden, da die Datensätze in den entsprechenden Blocken, trotz übereinstimmenden Token, vermutlich wenig Ähnlichkeit haben. Im Anschluss wird die

Kandidatenmenge C möglicher Matches bzw. Non-Matches, anhand der gruppierten Datensätze, generiert (Zeile 10-12). Um innerhalb der Blöcke den Paarvergleichsaufwand zu reduzieren, wird die Sorted Neighborhood verwendet und ein Fenster der Größe c über den Block geschoben. Nachdem das Fenster über jeden Block geschoben wurde, steht die Menge möglicher Kandidatenpaare fest. Diese Paare werden nun mit der TF/IDF-Ähnlichkeit sim (aus Cohen [26]) verglichen (Zeilen 15-21). Aufgrund der, über die kompletten Daten erfassten, TF/IDF Statistik beträgt die Komplexität des Vergleiches $O(1)$, da lediglich die entsprechenden TF und IDF Werte, der Attribute eines Paares, nachgeschlagen werden müssen. Ist die Ähnlichkeit $sim \geq ut$, wird das Paar als Match klassifiziert und zu P hinzugefügt (Zeilen 15-21). Analog, ist $sim < lt$ wird das Paar als Non-Match klassifiziert und zu N hinzugefügt (Zeilen 15-21). Von allen Matches werden jeweils die max_p mit der höchsten Ähnlichkeit sim ausgewählt (Zeilen 18-20). Analog werden ebenfalls die max_n Non-Matches mit der höchsten Ähnlichkeit sim gewählt (Zeilen 25-27). Bei den Non-Matches soll das verhindern, dass lediglich Paare mit $sim \approx 0.0$ ausgewählt werden, da diese für gewöhnlich zu niedrigen Klassifikationsraten führen.

- Werden schnell große Datensätze mit entsprechender Ground Truth benötigt„ bieten sich synthetisch generierte Datensätze an. Damit diese repräsentativ sind, sollten sie die gleichen Attribute haben wie die echten Datensätze. Dazu wird eine Datenbank möglicher Attributswerte benötigt, welche der Generator verwenden soll. Zusätzlich gibt es Parameter, um die Größe des Datensatzes und Anzahl der Duplikate, die Häufigkeitsverteilung der einzelnen Attribute und die Modifikationen der Duplikate gegenüber dem Original, in typographische, OCR oder phonetisch Fehler, zu bestimmen. Beispiele solcher Datensätze finden sich in Abschnitt 5.1.3.

- Anstatt synthetische Datensätze zu generieren und anschließend Fehler einzufügen, ist stattdessen auch möglich in einen bestehenden Datensatz Fehler einzubauen und diese als entsprechende Ground Truth zu verwenden. Dadurch werden allerdings die tatsächlichen Matches unterschlagen, was zu Konflikten bei der Entity Resolution führen kann.

2.6.1 Qualitätsmaße

Ist zu einem Datensatz die Ground-Truth verfügbar, so können die klassifizierten Datensätze einer der Kategorien in Tabelle 2.1 zugeordnet werden.

- True Positives (TP), sind alle Paare, die als Matches klassifiziert wurden und nach Ground Truth tatsächlich Matches sind.
- False Positives (FP), sind alle Paare, die als Matches klassifiziert wurden aber keine sind.

		Predicted classes	
		Match	Non-Match
Actual Matches	Match	True Positives (TP)	False Negatives (FN)
	Non-Match	False Positives (FP)	True Negatives (TN)

Tabelle 2.1 Matrix mit den vier Klassifikationszuständen. TP wenn tatsächliches und klassifiziertes Match, FN wenn tatsächlich Non-Match, aber klassifiziert als Match, FP wenn tatsächlich Match, aber klassifiziert als Non-Match und TN wenn tatsächliches und klassifiziertes Non-Match.

- False Negatives (FN), sind alle Paare, die als Non-Matches klassifiziert wurden aber tatsächlich Matches sind.
- True Negatives (TN), sind alle Paare, die als Non-Matches klassifiziert wurden und auch tatsächlich zwei verschiedene Entitäten identifizieren.

Das Ergebnis eines idealen Klassifikatoren ist, dass soviele Matches wie möglich True Positives sind und die Anzahl der False Positives, sowie False Negatives möglichst klein ist. Auf Basis der vier Klassifikationsklassen können Qualitätsmaße bestimmt werden. Die folgende Liste zeigt die beliebtesten Methoden und erklärt ihre Stärken und Schwächen.

- *Accuracy*.

$$acc = \frac{TP + TN}{TP + FP + TN + FN} \tag{2.2}$$

  Die Genauigkeit ist ein weit verbreitetes Qualitätsmaß für Binär- und Multi-Klassen Probleme im Maschine-Learning Bereich. Die Accuracy ist nützlich in Situationen, in denen die Klassen möglichst gleich verteilt sind. Für Entity Resolution ist dieses Maß daher nur bedingt geeignet, da zwischen Matches und Non-Matches fast immer ein Ungleichgewicht zugunsten von Non-Matches besteht. Daher sind die meisten klassifizierten Ergebnisse True Negatives, welche die Gleichung dominieren. Ein naiver Klassifikator der ausschließlich Non-Matches klassifiziert erhält dadruch eine hohe Genauigkeit.

- *Precision*.

$$prec = \frac{TP}{TP + FP} \tag{2.3}$$

  Precision wird oft als Qualitätsmaß von Suchergebnissen genommen, da es den Anteil der True Positives in den Matches berechnet. Für Entity Resolution misst die Precision den Anteil von korrekt bestimmten Matches.

- *Recall*.

$$rec = \frac{TP}{TP + FN} \tag{2.4}$$

  Recall misst den Anteil der tatsächlichen Matches (TP + FN), welche korrekt (TP) als Matches klassifiziert wurden. Zwischen Recall und Precision gibt es einen Kompromiss. Beispielsweise kann der Recall verbessert werden, indem die Precision abgesenkt bzw. die Precision verbessert, indem der Recall gesenkt wird.

- *F-Measure*. Auch bekannt als *f-score* oder *f_1-score*.

$$fmeas = 2 \cdot \left(\frac{prec \cdot rec}{prec + rec} \right) \tag{2.5}$$

  Das F-Measure berechnet das harmonische Mittel zwischen Precision und Recall. Eine guter F-Measure Wert ist daher ein Kompromiss zwischen den beiden.

Die oben genannten Qualitätsmaße berechnen alle einen exakten Wert für die Qualität eines Klassifikators. Aus den bekannten Verfahren für Blocking, Vergleich und Klassifizierung geht hervor, dass diese eine Reihe von Parametern haben, um das Ergebnis zu kalibrieren. Deshalb ist es sinnvoll eine Reihe von Werten zu erzeugen, um diese miteinander zu vergleichen. Ein solcher Vergleich funktioniert am einfachsten per Visualisierung. Die folgenden drei Visualisierungen werden dazu oft verwendet:

- *Precision-recall Graph*. Diese Visualisierung zeigt den Kompromiss zwischen Precision und Recall (siehe Abbildung 2.7[b]). Für jedes klassifizierte Datensatzpaar errechnet der Klassifikator eine Wahrscheinlichkeit. Mit Hilfe der Wahrscheinlichkeiten wird eine Reihenfolge bestimmt. Die Wahrscheinlichkeiten dienen dabei als Schwellwerte und für jede Schwelle wird Precision und Recall berechnet. Dabei ist die X-Achse stets der Recall und die Y-Achse die Precision. Durch den Kompromiss startet die Kurve meist in der oberen rechten Ecke mit hoher Precision und niedrigem Recall und endet in der linken unteren Ecke mit entgegengesetzten Werten. Dabei ist das Ziel die Kurve möglichst Nahe an die linke obere Ecke zu bekommen, in welcher Precision und Recall maximal sind.
- *Average Precision*. Der Precision-recall Graph kann effektiv die Qualität eines Verfahrens oder Klassifikators darstellen. Automatisiert eine Entscheidung anhand eines Graphen zu treffen ist jedoch schwierig, weshalb mit der Average Precsion die Fläche unter Precision-recall Kurve im Recallintervall 0 bis 1 ermittelt wird. Die Average Precision wird in der Praxis berechnet durch

$$\text{AveP} = \sum_{k=1}^{n} P(k)\Delta r(k), \tag{2.6}$$

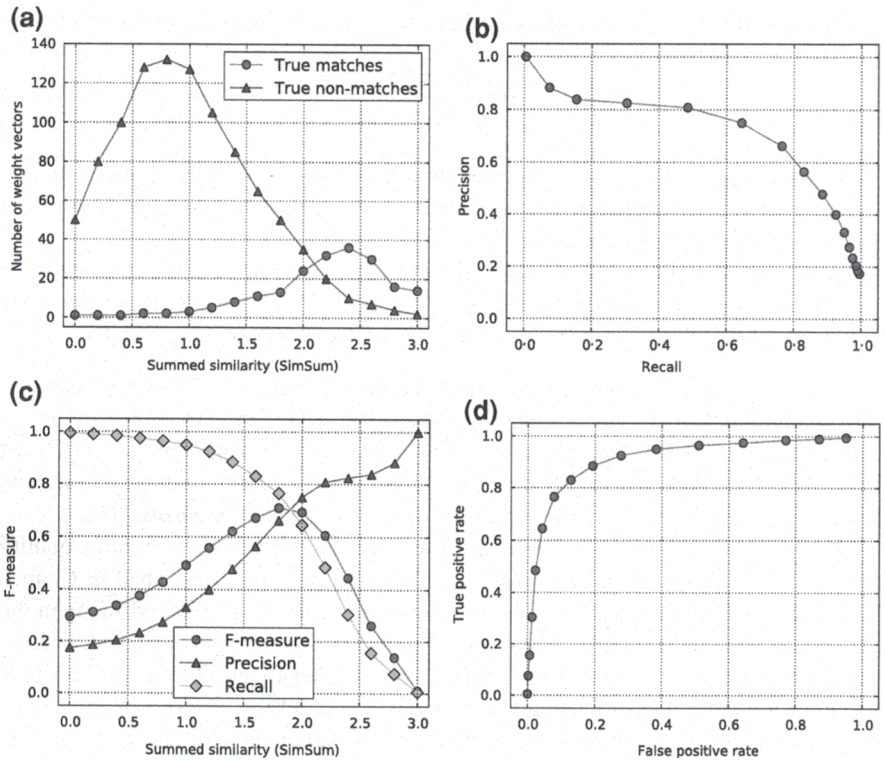

Abbildung 2.7 Beispiele von Qualitätsgraphen aus [30]. **b**. Precision-Recall, **c**. F-Measure, **d**. ROC Kurve.

wobei k die Schwelle ist, n die Anzahl der insgesamt klassifizierten Matches, $P(k)$ ist die Precision für die Schwelle k und $\Delta r(k)$ ist die Differenz des Recalls zwischen $k-1$ und k.

- *F-Measure Graph*. Anstatt zwei Qualitätmaße gegeneinander zu zeichen, kann man diese auch zusammen im Bezug auf einen bestimmten Parameter darstellen (siehe Abbildung 2.7[c]). Der F-Measure Graph plottet Precision, Recall und F-Measure gegen einen Parameter, wie etwa die akkumulierte Gesamtwahrscheinlichkeit, die bei den schwellenbasierten Klassifikatoren genutzt wird. Darüber kann dann abgelesen werden, bei welchem Wert (z.B. Schwelle) die beste Precision, der beste Recall und das beste F-Measure erreicht wird.
- *ROC Kurve*. Wie der Precision-recall Graph vergleicht die Receiver-Operating-Characteristic (ROC) Kurve zwei Qualitätmaße. Nämlich, auf der X-Achse die False-Positve-Rate und auf der Y-Achse den Recall (siehe Abbildung 2.7[d]). Obwohl die ROC Kurve robust gegen ungleichgewichtige Klassen ist, ist diese dennoch mit Vorsicht für Entity Resolution zu betrachten, da die False Positive Rate die True Negatives miteinbezieht und die Kurve dadurch etwas zu optimistisch Ausfallen kann. Verschiedene ROC Kurven verschiedener Klassifikatoren mit unterschiedlichen Parametern zu vergleichen, kann dennoch nützlich sein, um deren Qualität zu bewerten.

2.6.2 Effizienzmaße

Neben der Qualität bestimmt auch die Effizienz wie gut Entity Resolution Systeme funktionieren. Die offensichtlichste Art die Effizienz zu messen ist, die Laufzeit des Verfahrens zu messen und miteinander zu vergleichen. Dieser Ansatz ist allerdings abhängig von der genutzten Hardware und bietet keinen plattformübergreifenden Vergleich. Für die folgenden Maße müssen zunächst einige Mengen definiert werden. Zunächst wird in die Menge aller tatsächlichen Matches n_M und die Menge aller tatsächlichen Non-Matches n_N gegliedert. Dementsprechend ist $n_M + n_N = m \cdot n$ für Entity-Linking und $n_M + n_N = m(m-1)/2$ für Deduplizierung. Die Menge der durch Blocking gruppierten Datensatzpaare wird ebenso in Matches und Non-Matches geteilt und mit s_M bzw. s_N bezeichnet, wobei $s_M + s_N \leq n_M + n_N$.

- *Reduction Ratio*. Dieses Maß gibt an wie viele Datensatzpaare von einem Blockingverfahren generiert worden sind und setzt diese ins Verhältnis mit der Anzahl aller möglichen Datensatzpaaren, welche ohne Blocking generiert worden wären. Das Reduction Ratio ist definert als

$$rr = 1 - \left(\frac{s_M + s_N}{n_M + n_N} \right). \tag{2.7}$$

- *Pairs completeness*. Dieses Maß berechnet den Anteil der möglichen Matches. Es wird berechnet mit

$$pc = \frac{s_M}{n_M}. \tag{2.8}$$

  Pairs completeness ist mit dem Recall aus Formel 2.4 verwandt. Je geringer die Pairs completeness ist, desto geringer ist auch die Matchingqualität, da dieses Maß eine Obergrenze für einen möglichen Recall bestimmt. Denn tatsächliche Matches, die von einem Blocking Mechanismus nicht selektiert werden, können auch nicht klassifiziert werden. Zwischen Reduction Ratio und Pairs Completeness gibt es einen offensichtlichen Kompromiss, je mehr Datensatzpaare erzeugt werden, desto mehr tatsächliche Matches können gefunden werden.
- *Pairs quality*. Dieses Maß berücksichtigt die Qualität eines Blockingverfahrens, indem es die selektierten tatsächlichen Matches in Relation mit mit allen selektierten Paaren stellt. Es wird berechnet mit

$$pq = \frac{s_M}{s_M + s_N}. \tag{2.9}$$

  Die Pairs quality ist verwandt mit der Precision aus Formel 2.3. Eine hohe Pairs qualtiy bedeutet, dass ein Blockingverfahren hauptsächlich Paare erzeugt, welche tatsächlich Matches sind. Auch hier gibt es ähnlich zu Precision und Recall einen Kompromiss zwischen Pairs completness und Pairs quality.

3 Analyse

Ein kompletter Entity Resolution Workflow, wie in Kapitel 2 betrachtet, führt eine Reihe von Schritten aus. Diese Schritte lassen sich grob in vier Phasen gliedern. Zunächst die Vorverarbeitung, um offensichtliche Fehler zu korrigieren, gefolgt vom Blocking, welches die Komplexität der Suche reduziert, dem Matching, mit der Berechnung der Ähnlichkeiten im Paarvergleich und der Klassifizierung in Matches und Non-Matches und optional die Nacharbeitung, um Gruppen von Duplikaten zu erkennen. Ein System oder Framework zur dynamischen Entity Resolution besteht dementsprechend aus mindestens vier Komponenten. Einer Datenquelle, einer Vorverarbeitungspipeline, einem Indexer und einem Klassifikator. Neben der Wahl geeigneter Verfahren und Algorithmen für die genannten Komponenten, ist die größte Schwierigkeit die vielen Parameter auf die Datenquelle anzupassen. Werden beispielsweise die Parameter des DySimII Blocking Verfahren aus Abschnitt 2.3.2 betrachtet, so wird pro Attribut ein Blockschlüssel und eine Ähnlichkeitsfunktion benötigt. Bei einem Datensatz mit fünf Attributen, sind dies bereits 10 Parameter. Beim Matching kommen Parameter für die Ähnlichkeitsfunktionen, beispielsweise die Kosten der Operationen (einfügen, ersetzen, löschen) bei der Levenshtein Distanz, welche auf ein Attribut optimiert werden. Bei einem Attribut pro Ähnlichkeitsfunktion und beispielsweise abweichenden Werten für die Kosten der Einfügeoperation, kommen weitere fünf Parameter hinzu. Für das Matching kann ein simpler Schwellenwertklassifikator, mit lediglich einer Schwelle, genutzt werden. In dieser Konstellation (ohne Vorverarbeitung) mit Blocking Verfahren, verschiedenen Ähnlichkeitsfunktionen und einem Klassifikator, kommt das ER-System bereits auf 16 Parameter. Diese Parameter werden als freie Parameter bezeichnet und sind laut Wikipedia[1] folgendermaßen definiert:

> Ein freier Parameter ist eine Variable eines mathematischen Modells, die vom Modell nicht exakt vorhergesagt bzw. eingeschränkt werden kann und daher experimentell oder theoretisch geschätzt werden muss.

Die freien Parameter manuell zu bestimmen, ist selbst mit einer cleveren Strategie, die Parameter auszuprobieren, sehr zeit- und kostenintensiv. Besonders bei großen Datenmengen kann dieses Trial and Error Verfahren sehr lange dauern, da das Ausprobieren mehrere Stunden bis Tage in Anspruch nimmt. Noch schwieriger wird es, wenn keine Ground Truth Daten vorhanden sind. Dadurch entfällt größtenteils die Möglichkeit die

[1] https://en.wikipedia.org/wiki/Free_parameter

eingestellten Parameter effektiv und qualitativ zu überprüfen. Aufgrunddessen ist die manuelle Bestimmung der freien Parameter oft nicht pratikabel. Dem kann ein Entity Resolution System Abhilfe verschaffen, dass selbständig die freien Parameter bestimmt. Die von einem System bestimmten freien Parameter, werden im Folgenden als Konfiguration bezeichnet. Ein System zur selbstständigen Bestimmung seiner Konfiguration kann das Problem der freien Parameter jedoch nicht vollständig lösen, da die Verfahren zum Lernen ebenfalls parametrierbar sind und dadurch neue freie Parameter miteinbringen. Trotzdem können auf diese Weise die kritischsten Teile der Konfiguration ermittelt werden, die zum einen am meisten Zeit zum Einstellen und zum anderen maßgeblich die Qualitiät und die Effektivität des Gesamtsystems beeinflussen, sodass für ein solides System, Feinjustierungen ausgenommen, nicht zwangsweise ein Domänexperte benötigt wird.

Für den Erfolg einer Anfrage müssen zwei Eigenschaften erfüllt sein. Zunächst benötigt es ein Blocking Verfahren, dass bei gegebenen Anforderungen an die Latenzen, in der Lage ist die Duplikate in den existierenden Daten als Kandidaten zu selektieren. Anschließend benötigt es einen Klassifikator, der zuverlässig Matches von Non-Matches aus der Kandidatenmenge filtert. Dementsprechend sind die wichtigsten freien Parameter, die des Blocking Schema, die Auswahl geeigneter Ähnlichkeitsmaße, die für ein Attribut entscheidende Unterschiede zwischen Matches und Non-Matches messen und die Parameter eines Klassifikator, damit dieser die Ähnlichkeiten bestmöglich interpretieren kann.

Damit es möglich ist verschiedene Konfigurationen zu vergleichen und zu bewerten, benötigt es eine Ground Truth. In Abschnitt 3.1 werden zwei Verfahren beschrieben, die eine Ground Truth zu diesem Zweck syntetisieren. Des Weiteren wird in Abschnitt 3.2 ein Verfahren untersucht, um automatisiert ein Blocking Schema zu erlernen, dass von einem Indexer zum Blocking genutzt werden kann. Anschließend wird in Abschnitt 3.3 betrachtet, wie anhand eines Blocking Schema für einen Datensatz Blockschlüssel erzeugt werden können. In Abschnitt 3.4 werden Blocking Verfahren analysiert und entwickelt, die das erlernte Blocking Schema nutzen können und die Anforderungen aus Abschnitt 2.2 erfüllen. Danach wird ein Verfahren entwickelt, das geeignete Ähnlichkeiten für die Attribute auswählt und zum Schluss werden die Möglichkeiten zum Trainieren eines Klassifikators untersucht.

3.1 Generieren von Labels

Mit Hilfe einer Ground Truth können Metriken zu Qualität und Effizienz eines Verfahrens bzw. einer Kette von Verfahren berechnet werden, wodurch die Ergebnisse in Relation zu anderen Parametern, Komponenten oder Daten gesetztet werden können. Für

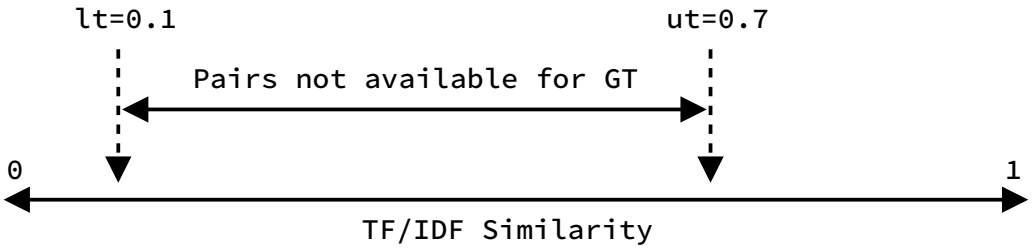

Abbildung 3.1 Darstellung der Lücke zwischen oberer und unterer Schwelle, des Algorithmuses des Label Generator ohne Ground Truth (GT), innerhalb welcher Paare nicht für die Ground Truth ausgewählt werden können.

den Fall, dass zu einem Datensatz keine Ground Truth existiert, ist die Bewertung des Ergebnisses entsprechend schwierig. Der bereits vorgestellte Algorithmus vom Kejriwal & Miranker [17] zur Erzeugung von schwachen Labels bietet hierfür zumindest die Möglichkeit eine Tendenz zu bekommen, die aussagt wie Qualität und Effizienz eines Verfahrens zu bewerten sind.

Der Algorithmus 4 nutzt dazu Token, in Form von Wörtern, für das Blocking der Datensätze, sowie zum Ermittlen der Ähnlichkeiten. Im Gegensatz zu einem Vollvergleich der Zeichenketten, beispielsweise durch eine Editierdistanz, wird dadurch die Komplexität auf Kosten der Genauigkeit reduziert. Die Genauigkeit ist gegenüber den Editierdistanzen niedriger, da lediglich auf Wörterebene miteinander vergleichen wird. Je mehr gemeinsame Wörter, desto ähnlicher ist ein Datensatzpaar. Hierdurch ist es beispielsweise nicht möglich Rechtschreibfehler zu erkennen, da der Algorithmus diese als zwei unterschiedliche Wörter behandelt. Die Gesamtkomplexität des Algorithmus beträgt $O(n + nm + nm)$, welche sich in die Erzeugung der TF/IDF Statistik ($O(n)$), die Erzeugung der Blöcke über m Attribute ($O(nm)$) und die Erzeugung der Kandidatenpaare $O(nm)$ gliedert. Bei diesem Algorithmus ist kritisch zu betrachten, dass ein Großteil der Datensatzpaare, aufgrund der Lücke zwischen den Schwellen, nicht für die Ground Truth ausgewählt werden kann. Abbildung 3.1 illustriert diese Lücke zwischen einer unteren Schwelle bei 0.1 und oberen Schwelle bei 0.7. Für alle Paare p gilt, wenn $lt \leq sim(p) < ut$, dann folgt $p \notin P \cup N$. Dadurch ist in vielen Fällen die generierte Ground Truth gegenüber dem Datensatz, nicht sonderlich repräsentativ, da viele aussagekräftige Paare ausgeschlossen werden. Kejriwal & Miranker haben in [17] die Werte für die Schwellen und der Fenstergröße empirisch an drei relativ kleine Datensätzen (< 10.000 Einträge) getestet. Ob die Ergebnisse sich auf deutlich größere Datesätze anwenden lassen, wird in Abschnitt 5.3 evaluiert.

Aufgrund der Verteilung von Matches und Non-Matches, die bis auf wenige Ausnahmen, immer ein deutliches Ungleichgewicht zugunsten der Non-Matches aufweist, werden für alle öffentlich verfügbaren Datensätze mit Ground Truth, lediglich die Matches angegeben. Dementsprechend sind alle Datensatzpaare, welche nicht in den Matches der

Algorithm 5 WeakTrainingSet with Ground Truth

Input:
- Dataset: D
- Ground Truth GT
- Blocking Window Size: c

Output:
- A set of positive samples: P
- A set of negative samples: N

```
Initialize set P = (), set N = ()
Initialize set of tuple pairs C = ()
Generate TFIDF statistics of D
for fields f ∈ D do
    for records r ∈ D do
        Tokenize r_f into BKV_f
        Block r on generate tokens for field f
    end for
end for
for block B generate in previous step do
    Slide a window of size c over tupels in B
    Generate all possible pairs within window and
    add to C
end for
for pairs (t_1, t_2) ∈ GT do
    Add (t_1, t_2) to P
    if p ∈ C then
        Remove p from C
    end if
end for
Initialize dictionary M = {}
for pairs (t_1, t_2) ∈ C do
    Compute TFIDF similarity sim of (t_1, t_2)
    M[(t_1, t_2)] = sim
end for
Calculate probability distribution over M
max_n = |GT| * 5
for i = 1 to max_n do
    Choose a pair p from M based on probability distribution
    Add p to N
end for
Return P and N
```

Ground Truth enthalten sind, als Non-Matches zu interpretieren. Das bedeutet, dass aus der riesigen Menge von Non-Matches eine repräsentative Stichprobe gezogen werden muss, anhand welcher Qualität und Effizienz, effektiv beurteilt werden können.

Um Non-Matches für eine Ground-Truth auszuwählen, müssen Paare gebildet und deren Ähnlichkeit berechnet werden. Die gesamten Non-Matches zu bilden und deren Ähnlichkeiten zu berechnen ist nicht effektiv, da diese Funktion bekannterweise quadratisch ($O(n^2)$) ist. Für ihr Verfahren zur Bestimmung einer schwachen Ground Truth haben Kejriwal & Miranker ein Verfahren entwickelt, das Paare bildet und deren Ähnlichkeit berechnet. Anstatt durch zwei Schwellen Matches und Non-Matches zu trennen, ist es bei vorhandener Ground Truth möglich, die Matches aus der Gesamtmenge der gebildeten Paare zu entfernen, sodass lediglich die Non-Matches übrig bleiben. Da das Blocking der Paare anhand von Token durchgeführt wurde, sind zum einen wenig substantielle Paare mit einer Ähnlichkeit nahe 0 ausgeschlossen worden und zum anderen substantielle Paare mit teils hoher Ähnlichkeit erzeugt worden. Ein Klassifikator beispielsweise kann relativ einfach Non-Matches mit geringer Ähnlichkeit von Matches trennen. Damit aber herausfordernde Paare, die sich etwa nur in einem Attribut unterscheiden, als Non-Match klassifiziert werden und nicht fälschlicherweise als Match betrachtet werden, wird eine Menge dieser uneindeutigen Paare benötigt. Aufgrund dessen bietet die Kandidatenerzeugung aus [17] eine gute Approximation, um schnell und effektiv gute

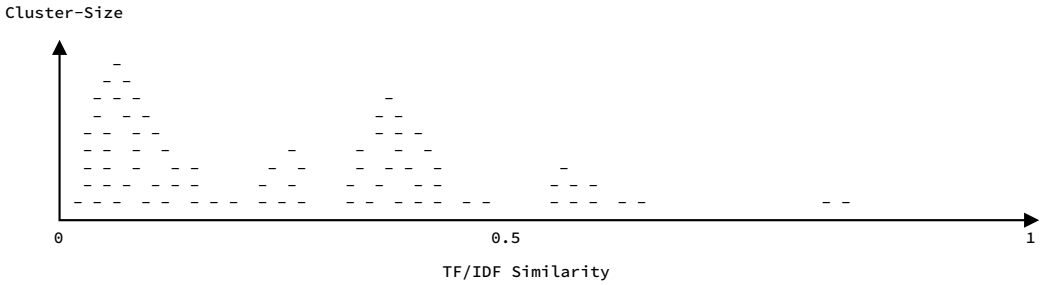

Abbildung 3.2 Beispielhalfte Verteilung von Non-Matches ('-' Symbol) auf der über die Ähnlichkeit (X-Achse) zwischen 0 und 1. Wie viele Non-Matches einen bestimmten Ähnlichkeitswert haben, wird durch die Häufung (Y-Achse) dargestellt.

Non-Matches zu erhalten. In Algorithmus 5 wird ein Verfahren beschrieben, das die Kandidatenerzeugung aus Algorithmus 4 nutzt, um damit eine Menge von Non-Matches zu bestimmen. Gegeben ist die Ground Truth in Form von Matches. Davon ausgehend wird eine repräsentative Menge von Non-Matches aus dem Datensatz D selektiert. Die ersten drei Schritte: das Generieren der TF/IDF Statistik, das Blocken durch die Token und das Erzeugen der Kandidatenmenge C (Zeilen 1-12 in grau), sind daher identisch zum ursprünglichen Algorithmus. Nachdem die Kandidatenmenge erzeugt wurde, werden zunächst alle Ground Truth Matches nach P übernommen (Zeilen 13-18). Zusätzlich werden alle Matches aus der Kandidatenmenge C entfernt, sodass diese ausschließlich Non-Matches beinhaltet. Anschließend wird ebenfalls die TF/IDF-Ähnlichkeit der Paare in C ermittelt und in M zwischengespeichert (Zeilen 20-23). Anhand dieser wird die Wahrscheinlichkeitsverteilung der Ähnlichkeiten in M ermittelt, beispielsweise durch ein Histogramm (Zeile 24). In Abbildung 3.2 ist beispielhaft eine Verteilung von Non-Matches (–Symbol) dargestellt. Die X-Achse gibt den Ähnlichkeitswert der Paare an. Die Häufung auf der Y-Achse illustriert, wie viele Paare eine entsprechende Ähnlichkeit haben. Eine Häufung ist vor allen Dingen im unteren Ähnlichkeitsbereich zu erwarten. Durchaus möglich sind allerdings auch größere Anhäufungen im mittleren Bereich, da durch das Blocking der Großteil der Paare mit Ähnlichkeit 0 ausgeschlossen worden ist. In Zeile 27 werden nun Non-Matches, anhand der Wahrscheinlichkeitsverteilung, zufällig aus M gezogen, sodass Paare innerhalb einer großen Anhäufung (z.B. unterer Bereich) häufiger ausgewählt werden als Paare in kleinen Anhäufungen (z.B. oberer Bereich). Damit wird erreicht, dass die Menge der für die Ground Truth gewählten Non-Matches möglichst repräsentativ ist. Die Ziehung wird max_n-Mal wiederholt bzw., solange bis keine Paare mehr übrig sind (Zeilen 26-29). Zum Schluss wird analog zum ursprünglichen Algorithmus die Grund Truth bestehend aus P und N zurückgegeben.

3.2 Lernen von Blocking Schemata

Kejriwal & Miranker haben ein Verfahren entwickelt, dass in Algorithmus 3 vorgestellt wurde. Darin wird ein Blocking Schema erzeugt, indem Ausdrücke anhand der Fisher-Score bewertet werden. Die berechnete Fisher-Score drückt für einen Ausdruck t, in Abhängigkeit der Groud Truth P und N, die Blockschlüsselabdeckung (engl. blocking key coverage) aus. Laut Ramadan & Christen [36] führt eine hohe Schlüsselabdeckung zu einer hohen Pair Completeness, sodass viele Matches in einen gemeinsamen Block gruppiert werden, während die Anzahl an Non-Matches, die zusammen einem Block zugeordnet werden, gering gehalten wird. Durch dieses Verfahren werden für einen Entity Resolution Workflow hochqualitative Blöcke erzeugt. Für dynamische Verfahren, die Anfragen im Subsekundenbereich beantworten und möglichst gleiche Latenzen haben sollen, ist aufgrund der niedrigen Dichte von Duplikaten (vgl. Anzahl gesamter Matches vs gesamter Non-Matches), die Fisher-Score als Bewertungskriterum ungeeignet. Zwar werden für die Duplikate Blöcke generiert, die es erlauben (möglichst) alle zur Anfrage passenden Entitäten schnell und präzise zu erhalten, allerdings ist der Großteil aller Anfragen ergebnislos. Ergebnislos in diesem Zusammenhang bedeutet, dass es zu einer Anfrage keinen Datensatz gibt, der derselben Entität entspricht. Das Problem ist, dass die Fisher-Score für diese Datensätze keine Aussage trifft, weshalb die generierten Blöcke, in welchen sich keine Matches befinden, zum Teil sehr groß werden. Aufgrund der Menge dieser Anfragen, wird die Effektivität des ER Systems dramatisch reduziert.

In [36] erweitern Ramadan & Christen den Algorithmus von Kejriwal & Miranker, um eine Bewertungsfunktion für Entity Resolution in nahe Echtzeit. Die Fisher-Score FS wird weiterhin genutzt um die Blockschlüsselabdeckung auszudrücken. Dies ist jedoch für dynamsiche Blocking Verfahren alleine nicht ausreichend, weil die Bewertung eines Ausdrucks zusätzlich von der Blockgröße und der Verteilung der Blöcke abhängt. Das Kontrollieren der Blöckgröße für alle Blöcke, auch solche die lediglich Non-Matches enthalten, sorgt dafür, dass alle Anfragen in nahe Echtzeit beantwortet werden können. Zur Bewertung der Blöckgröße wird die durchschnittliche Anzahl an Datensätzen pro Block ermittelt $S_{b_{(ave)}} = ave|b| : b \in B$, wobei mit B alle von einem Blocking Verfahren erzeugten Blöcke bezeichnet werden. Zusätzlich wird die maximale Blockgröße erhoben $S_{b_{(max)}} = max|b| : b \in B$. Erzeugt ein Ausdruck einen Block, der größer eines Schwellwertes ist, wird dieser Ausdruck verworfen. Über die Verteilung der Blöcke wird geprüft, dass alle Anfragen möglichst gleiche Latenzen haben. Dafür wird die Varianz in den Blockgrößen von B bestimmt $V = \frac{\sum_{b=1}^{|B|}(|b|-\mu_b)^2)}{|B|}$. Für einen Ausdruck k ergibt sich daraus die Bewertungsfunktion:

$$SC_k = \alpha \cdot (1 - FS_k) + \beta \cdot S_{b_{(ave)_k}} + (1 - \alpha - \beta) \cdot V_k,$$

Algorithm 6 LearnOptimalBS(S, k, d)

Input:
- Set of specific blocking predicates: S
- Maximum conjunctions per term: k
- Maximum disjunctions of terms: d

Output:
- Blocking Scheme: BS

Initialize set of blocking scheme candidates $BS_C = ()$
Initialize set of terms $T = ()$
for $i = 1$ **to** k **do**
  Generate combination of S with cardinality i and add to T
end for
for term $t \in T$ **do**
  $fmeasure, y_{true}, y_{pred} = evaluateTerm(t)$ ▷
  if $fmeasure = thres$ **then**
    Remove t from T
  end if
end for
for $i = 1$ **to** d **do**
  Generate combination C of T with cardinality i
  Add C to BS_C
end for
for Blocking scheme $bs \in BS_C$ **do**
  Initialize array y_{true} with length $|P \cup N|$
  Initialize array y_{pred} with length $|P \cup N|$
  for term $t \in bs$ **do**
    $y_{true} \vee t.y_{true}$
    $y_{pred} \vee t.y_{pred}$
  end for
  Score $s = fmeasure(y_{true}, y_{pred})$
  if $s > top_score$ **then**
    $BS = bs$
  end if
end for
return BS

Algorithm 7 EvaluateTerm(t, D, IX, P, N)

Input:
- Term: t
- Dataset: D
- Indexer: IX
- Set of positive pairs: P
- Set of negative pairs: N

Output:
- F-Measure: f
- Labels: y_{true}
- Predictions: y_{pred}

Build Index I over D using Indexer IX and Term t
Initialize set of pairs $C = ()$
Initialize $TP = 0$, $FP = 0$, $FN = 0$
for block $b \in I$ **do**
  Generate pair combinations pc for records in b and add them to C
  According to P and N calculate number of true positives, false positives and false negatives and sum them up with TP, FP and FN.
end for
Calculate F-Measure f according to TP, FN, FP
Initialize array y_{true} and y_{pred} with length $|P \cup N|$
for $i = 1$ **to** $|P|$ **do**
  Pair $p = P[i]$
  $y_{true} = True$
  if $p \in C$ **then**
    $y_{pred} = True$
  else
    $y_{pred} = False$
  end if
end for
for $i = |P| + 1$ **to** $|N|$ **do**
  Pair $p = N[i]$
  $y_{true} = False$
  if $p \in C$ **then**
    $y_{pred} = True$
  else
    $y_{pred} = False$
  end if
end for
return f, y_{true}, y_{pred}

wobei α und β genutzt werden, um die drei Kriterien zu gewichten. Je niedriger der Wert, desto besser wird ein Ausdruck gewertet. Gegenüber der ursprünglichen Bewertungsfuntion hat dieser Ansatz den Nachteil, dass für jeden Ausdruck von einem Blocking Verfahren die Blöcke B generiert werden müssen, um Durschnitt und Varianz zu berechnen. Hierdurch dauert die Bewertung der Ausdrücke deutlich länger. Ein weiterer Nachteil ist, dass die zwei freien Parameter (α und β) domänabhängig bzw. datensatzabhängig angepasst werden müssen. Der Wertebereich für die Fisher-Score liegt dabei zwischen 0 und 1, wohingegen der Blockdurchschnitt und die Blockvarianz deutlich darüber liegen können. In der Praxis ist es daher äußerst schwierig geeignete Gewichte zu wählen, sodass die Fisher-Score zur Geltung kommt und nicht von den anderen beiden Kriterien dominiert wird.

Zur Bewertung die Blöcke jedes Ausdrucks erzeugen zu lassen ist zwar aufwändig, da die generierten Blöcke jedoch vom genutzten Blocking Verfahren abhängig sind, ist dieser Schritt unvermeidlich. Verbesserungswürdig ist allerdings die Bewertungsfunktion. Die Anforderung an eine solche Funktion ist, Ausdrücke mit einer hohen Pairs Completeness und einer hohen Pairs Quality ausfindig zu machen und danach einzustufen. Aus Abschnitt 2.6 ist bekannt, dass die Pairs Completeness mit dem Recall verwandt ist und die Pairs Quality mit der Precision. Zudem ist bekannt, dass es einen Kompromiss zwischen Recall und Precision gibt, der durch das F-Measure ausgedrückt werden kann. Ein optimales F-Measure maximiert dementsprechend Recall und Precision. Aufgrund der

Verwandtschaft zwischen Recall und Precision zu Pairs Completeness und Pairs Quality kann das F-Measure ebenfalls genutzt werden, um diese beiden Attribute zu optimieren. Damit erfüllt das F-Measure die Anforderungen an eine gute Bewertungsfunktion. Die Evaluierung eines Ausdrucks, mit dem F-Measure als Bewertungsfunktion, ist in Algorithmus 7 beschrieben. Zur Bewertung eines Ausdrucks t benötigt der Algorithmus den Datensatz D, sowie die Matches P und die Non-Matches N, der Ground Truth. Zudem wird, das einzusetzende Blocking Verfahren benötigt. Dieses wird vertreten durch einen Indexer IX. Ein Ausdruck t wird dem Indexer IX als Blocking Schema übergeben, woraus dieser die zugehörigen Blöcke I, anhand der Datensätze aus D baut (Zeile 1). Durch die Betrachtung des konkreten Index, wird eine DNF erzeugt die auf den Indexer zugeschnitten ist. Als nächstes werden alle generierten Blöcke in I betrachtet (Zeile 4). Der Indexer muss dazu eine entsprechende Blockliste bereitstellen. Ein Block ist in diesem Zusammenhang nicht zwangsweise eine Gruppierung die über einen Blockschlüssel gebildet wurde, sondern jegliche Anhäufungen von Datensätzen, die bei einer Anfrage zusammen als Kandidatenmenge ausgewählt werden. Die Details hierzu werden in Abschnitt 3.4 erläutert. Für jeden Block werden zunächst die Paarkombinationen[2], aller dem Block zugehöriger Datensätze, ermittelt. Diese werden zu der Menge aller Paarkombinationen aller Blöcke C hinzugefügt (Zeile 5). Des Weiteren wird für einen Block b die Anzahl der Paare in den Klassifikationskategorien

- true positives, wenn ein Paar $p \in b$ und $p \in P$
- false positives, wenn ein Paar $p \in b$ und $p \in N$
- true negatives, wenn ein Paar $p \notin b$ und $p \in P$

über die Grund Truth ermittelt und in TP, FP und FN aufsummiert (Zeile 6). Anhand der Werte TP, FP, FN wird das F-Measure zur Bewertung des Ausdrucks t bestimmt (Zeile 8). Bei der späteren Disjunktion von Ausdrücken kann dieses F-Measure allderdings nur zur Vorauswahl der infrage kommenden Audrücke genutzt werden, da sich die F-Measure Werte verschiedener Ausdrücke aus unterschiedlichen Paarkombinationen berechnen. Damit die Ausdrücke effizient disjunktiert werden können, wird die Paarkombination auf die Ground Truth abgebildet. Dazu werden zwei Arrays y_{true} und y_{pred} mit der Länge $|P \cup N|$ erzeugt (Zeile 9). Diese können beim Zusammenfügen der DNF, ähnlich wie die Featurevektoren in [17], einfach verodert und daraus das F-Measure bestimmt werden. Für die Abbildung auf die Ground Truth müssen alles Matches P (Zeile 10) und alle Non-Matches (Zeile 19) betrachtet werden. y_{true} gibt an, welcher Klasse ein Paar p angehört: Match, wenn $p \in P$ (Zeile 12) oder Non-Match, wenn $p \in N$ (Zeile 21). y_{pred} gibt an, ob ein Datensatzpaar einen gemeinsamen Block in I hat $y_{pred}[p] = True$ (Zeilen 14,23) oder nicht $y_{pred}[p] = False$ (Zeilen 16,25). Die Bewertung des Ausdrucks über das F-Measure, y_{true} und y_{pred} werden zum Schluss an den Aufrufer zurückgegeben.

[2] 2-Tupel der Datensätze ohne festgelegte Reihenfolge

Die Änderungen an der Bewertungsfunktion führen dazu, dass das Verfahren zum Bilden eines Disjunktiven Blocking Schema aus [17] nicht mehr angewendet werden kann, da die Featurevektoren (welche dazu genutzt wurden) nicht länger erzeugt werden. Der Algorithmus zur Bestimmung des optimalen Blocking Schemas ist in Algorithmus 6 dargestellt. Die ersten Parameter sind die spezifischen Blockingprädikate S. Diese werden vom Aufrufer der Funktion bestimmt. Da die maximale Konjunktion der Blockingprädikate bzw. die maximale Disjunktion potentiell unendlich groß ist, werden diese über die Parameter k für die Konjunktionen und d für die Disjunktionen begrenzt. Ein weiterer Grund die Konjunktionen bzw. Disjunktionen nicht beliebig zu erhöhen ist, dass das ein Blocking Verfahren deutlich komplexere Blockschlüssel erzeugen muss. Denn für jedes spezifische Blockingprädikat muss, beim Erzeugen der Blockschlüssel eines Datensatzes, die Prädikatsfunktion aufgerufen werden. Demnach nimmt die Effizienz der Blockschlüsselgenerierung ab, je mehr Konjunktionen bzw. Disjunktionen ein Blocking Schema hat. Am Anfang werden die konjugierten Ausdrücke erzeugt, indem alle Kombinationen der Menge spezifischer Blockingprädikte S bis zur Länge k der maximalen Konjunktionen berechnet werden (Zeilen 3-5), somit ist

$$T = \{\{\text{2-Tupel von S}\}, \{\text{3-Tupel von S}\}, \ldots, \{\text{k-Tupel von S}\}\}.$$

Anschließend wird jeder Ausdruck t durch den oben erklärten Algorithmus 7 evaluiert. Ist der F-Measure Wert f für t kleiner einer Schranke $thres$, indiziert dies einen ungeeigneten Block und der Ausdruck wird entfernt (Zeile 9). Aus den in T noch vorhandenen Ausdrücken, werden durch Disjunktion bis zur Länge d mögliche Kandidaten eines Blocking Schemas generiert (Zeilen 12-14). Ein Blocking Schema wird ausgewählt, indem die Arrays y_{pred} und y_{true} der Ausdrücke in jedem potentiellen Blocking Schema verodert werden (Zeilen 20-21) und daraus das F-Measure berechnet wird (Zeile 23). Das potentielle Blocking Schema mit dem höchsten F-Measure wird abschließend ausgewählt und zurückgegeben.

3.3 Blockschlüsselgenerierung

Die Erzeugung von Blockschlüsseln aus einem einzelnen Attribut sind vielfältig. In Abschnitt 2.3.1 wurde ein Verfahren vorgestellt, dass Q-gramme bildet. Ein anderes Verfahren nutzt Attributssuffixe. Auch denkbar sind Attributsprefixe, welche bei der Sorted Neighborhood Anwendung finden. Zudem sind phonetische Enkodierungen beliebt, um unterschiedliche Schreibweisen zusammenzuführen. Damit diese Verfahren für ein DNF Blocking Schema nutzbar sind, werden diese als allgemeine Blockingprädikate abgebildet, beispielsweise `HasCommonQGram` oder `HasCommonSuffix`. Die Anwendung der erzeugten Blockschlüssel bei der Disjunktion von spezifischen Blockingprädikaten ist dabei intuitiv. Für jeden Blockschlüssel s, der anhand eines Datensatzes r erzeugt wurde, wird r einem Block zugeordnet. Die einzige Überlegung hierbei ist, ob die Blöcke

verschiedener spezifischer Blockingprädikate disjunkt sind oder nicht. Konkret bedeutet dies, erzeugen zwei Prädikate auf unterschiedlichen Attributen denselben Schlüssel, verweisen diese auf denselben Block oder werden unterschiedliche Blöcke adressiert. Dieses Problem muss vom jeweiligen Blocking Verfahren gelöst werden, die Konsequenzen daraus werden in Abschnitt 3.4 erläutert. Im Gegensatz dazu ist die Behandlung der konjunktiven Ausdrücke bei der Blockschlüsselerzeugung deutlich komplexer. Hierbei müssen zwei oder mehr Mengen von Blockschlüsseln miteinander verknüpft werden. Sei Σ das Alphabet über einem Datensatz D, Σ^* die Menge aller Wörter des Alphabets und $r.f$ ist ein Attribut eines Datensatzes r, dann ist $r.f \subseteq \Sigma^*$. Gegeben seien n spezifische Blockingprädikate $(p_1, f_1), \cdots, (p_n, f_n)$ und eine Relation $S((p_k, f_k), r) \subseteq r.f_k$, die anhand eines Prädikates (p_k, f_k) und eines Datensatzes r eine Menge von Blockschlüsseln erzeugt. Die Menge von Blockschlüsseln, die aus einem konjunktiven Ausdruck gebildet werden, besteht entsprechend aus der Konjunktion der Teilmengen S. Betrachtet man die Verundung der spezifischen Blockingprädikate strikt, so müssen die Mengen kommutativ verknüpft werden. Diese Eigenschaft ist für die Blockschlüsselbildung interessant, da deren Einhaltung oder Nichteinhaltung maßgeblich das Ergebnis eines Blocking Verfahrens beeinflusst. Wenn die Kommutativität nicht eingehalten wird, können die Blockschlüssel relativ einfach über das Kartesisches Produkt der Mengen erzeugt werden

$$BKV(r) = S_1 \times S_2 \times \cdots \times S_n.$$

Die kommutative Verknüpfung ist etwas komplizierter, hierzu definieren wird zunächst die Vereinigungsmenge $\mathbb{S} = S_1 \cup S_2 \cup \cdots \cup S_n$ und bilden die Blockschlüssel über das Kartesische Produkt dieser Menge

$$BKV(r) = \underbrace{\mathbb{S} \times \cdots \times \mathbb{S}}_{\text{n-Mal}}.$$

Als Beispiel wird der Ausdruck A = (`ExakteÜbereinstimmung`, `Vorname`) $\wedge$ (`ExakteÜbereinstimmung`, `Zweitname`) $\wedge$ (`ExakteÜbereinstimmung`, `Nachname`) betrachtet. Die Tabelle 3.1 beinhaltet vier Datensätze, die eindeutig über die ID identifiziert werden können. Jeweils zwei der Datensätze beschreiben dieselbe Entität. Zur Demonstration der Auswirkung der Kommutativität werden drei Paare betrachtet. Zunächst das Paar (1, 2). Beide Datensätze beschreiben dieselbe Entität, aber der Vorname wurde mit dem Zweitnamen vertauscht. Beim nächsten Paar (1, 3) gibt es ebenfalls eine Vertauschung, hier zwischen Vorname und Nachname. Allerdings beschreiben die beiden Datensätze unterschiedliche Entitäten. Das dritte Paar (3, 4) beschreibt wieder dieselbe Entität, jedoch gibt es in Datensatz 4 einen Schreibfehler im Vornamen. Angenommen $\wedge$ ist kommutativ, dann ist $BKV(1) \cup BKV(2) \neq \emptyset$, wodurch das Paar (1, 2), trotz vertauschter Attribute, einem gemeinsamen Block zugeordnet wird. Das Gleiche gilt aber auch für das Paar (1, 3), sodass $BKV(1) \cup BKV(3) \neq \emptyset$ und folglich wird dieses Paar ebenfalls einem gemeinsamen Block zugeordnet. Für den Fall, dass $\wedge$ nicht kommutativ ist, haben weder das Paar (1, 2) noch (1, 3) einen gemeinsamen Blockschlüs-

ID	Entitäts-ID	Vorname	Zweitname	Nachname
1	1	Peter	Moritz	Michel
2	1	Moritz	Peter	Michel
3	2	Michel	Moritz	Peter
4	2	Michle	Moritz	Peter

Tabelle 3.1 Beispieldatensätze zur Veranschaulichung der Kommutativität bei der Blockschüsselerzeugung. Das Attribut ID identifiziert den einzelnen Datensatz und die Entitäts-ID ordnet Datensätze Entitäten zu. Dementsprechend beschreiben Datensatz 1 und 2, sowie 3 und 4 die selbe Entität.

sel. Der Nachteil der kommutativen Blockschlüsselbildung anhand der obigen Formel ist, dass deutlich mehr Blockschlüssel erzeugt werden. Dementsprechend wird dadurch die Pairs Completeness auf Kosten der Pairs Quality erhöht werden. Ist die Erzeugung nicht kommutativ wird umgekehrt, die Pairs Quality auf Kosten der Pairs Completeness verbessert. Damit die Menge der Blöcke durch eine kommutative Schlüsselerzeugung nicht explodiert, kann diese simuliert werden. Dazu werden zunächst die Schlüssel nicht kommutativ erzeugt und anschließend werden die Tupel des Kartesischen Produktes sortiert. Die Kommutativität von $\wedge$ kann jedoch keinen Einfluss auf Rechtschreibfehler, wie im Falle von Paar (3, 4) nehmen. Eine Möglichkeit Rechtschreibfehler bei der Erzeugung der Blockschlüssel herauszufiltern ist, tokenbasierende Prädikate (z.B. Q-Gramme) zu verwenden. Der Nachteil hierbei ist, dass für die einzelnen Attribute mit einem solchen Prädikat deutlich mehr Teilschlüssel erzeugt werden. Als Folge dessen wird zum einen den Prozess der Schlüsselgenerierung verlangsamt und zum anderen die Chance erhöht, dass aufgrund einer größeren Menge von Blöckschlüsseln unähnliche Datensätze zusammen in einen Block gruppiert werden. Um bei gleicher Komplexität transpositive Rechtschreibfehler zu filtern, muss $\wedge$ kommutativ sein und zusätzlich auf Zeichenebene angewendet werden, beispielsweise indem die Teilschlüssel konkateniert und als Anagramme behandelt werden. Dadurch ist es wiederum möglich die Pairs Completeness zu erhöhen. Allerdings ebenfalls auf Kosten der Pairs Quality, da die Kollisionsrate durch Reduktion der möglichen Blockschlüssel erhöht wird. Im Vergleich zu einem tokenbasierenden Prädikat bleibt bestenfalls die Anzahl der Blockschlüssel gleich bzw. wird reduziert, anstatt zu wachsen.

Algorithmus 8 beschreibt ein nicht kommutatives Verfahren Blockschlüssel durch Konkatenation zu bilden. Jeder Ausdruck t des DNF Blocking Schema erzeugt für einen Datensatz r eine Menge von Blockschlüsseln. Die Blockschlüssel der Ausdrücke werden unabhängig voneinander gebildet. Der Algorithmus erhält daher als Eingabewert nur einen Ausdruck, bestehend aus mindestens einem spezifischen Blockingprädikat. Diese werden der Reihe nach betrachtet (Zeile 2). Anhand des Prädikats p werden alle Teilblockschlüssel für das verknüpfte Attribut $p.field$ generiert. Beispielsweise liefert das Prädikat `GemeinsamerToken` alle durch Leerzeichen getrennte Token des Attributes $r.field$

Algorithm 8 BlockingKeyValues(t, r)

Input:
- Term from Blocking Schema t
- Record r

Output:
- Blocking Key Values: BKV

```
1: Initialize list BKV = []
2: for specific blocking predicate p ∈ t do
3:     field f = p.field
4:     attribute keys ak = p.predicate(r.f)
5:     if BKV is empty then
6:         BKV = ak
7:     else
8:         bkv = []
9:         while BKV is not empty do
10:            Take key x from BKV
11:            for key y in ak do
12:                xy = concatenate(x, y)
13:                Append xy to BKV
14:            end for
15:        end while
16:        BKV = bkv
17:    end if
18: end for
19: return BKV
```

(Zeile 4). Ist die BKV Liste zu diesem Zeitpunkt leer, wird die Schlüsselliste ak als BKV Liste übernommen. Existieren in BKV allerdings schon Schlüssel, wird zunächst eine temporäre Liste bkv erzeugt. Anschließend werden aus BKV solange Schlüssel entnommen, bis diese leer ist (Zeile 9). Für jeden entnommenen Schlüssel x werden $|ak|$ neue Schlüssel erzeugt. Dazu wird der neue Schlüssel xy gebildet, indem ein Schlüssel y aus ak mit x konkateniert wird (Zeile 12). Alle auf diese Weise neu erzeugten Schlüsselpaare xy werden zu bkv hinzugefügt. Wenn die BKV Liste leer ist, wird die temporäre Liste bkv nach BKV übernommen (Zeile 16). Nachdem alle Prädikate bearbeitet wurden, wird die Liste der generierten Blockschlüssel BKV zurückgegeben.

3.4 Dynamische Blocking Verfahren

3.4.1 Similarity-Aware Inverted Index

Die Idee des DySimII (vgl. Abschnitt 2.3.2) ist ein Standard Blocking Verfahren, dass es erlaubt, die Ähnlichkeit von Datensätzen nachzuschlagen, indem Attributsähnlichkeiten vorausberechnet werden. Im Wesentlichen handelt es sich bei DySimII um einen Multi-pass Ansatz, da für einen Datensatz mehrere Blockschlüssel erzeugt werden. Zu diesem Zweck wird für jedes Attribut eine Enkodierungsfunktion genutzt, die genau einen Blockschlüssel erzeugt. Anhand dessen werden Attribute zusammen gruppiert und innerhalb der Gruppierung die Ähnlichkeit berechnet. Die errechnete Ähnlichkeit wird in einer Art Cache, dem Similarity Index, vorgehalten. Während einer Abfrage wird vermieden, dass ein Großteil der teuren Ähnlichkeitsberechnungen eingespart wird.

Beim Gruppieren von Attributen in Blöcke, führt der DySimII keine Trennung von Attributen durch, sodass Attributswerte von Vorname und Zweitname bzw. Nachname im

einen gemeinsamen Block gruppiert werden können. Dadurch wird zum einen die Abdeckung möglicher Attributskombinationen erhöht und zum anderen vermieden, dass Ähnlichkeiten doppelt berechnet werden. Diese Einsparung macht sich besonders bei ähnlichen Attributen mit großteils überlappenden Werten, wie Vorname und Zweitname, bemerkbar. Neben der Ähnlichkeitsvorausberechnung dienen die Blöcke auch zum Bilden einer Kandidatenmenge möglicher Duplikate. Für einen Anfragedatensatz q besteht die Kandidatenmenge C einerseits aus Datensätzen, die über den RI selektiert werden, weil diese in irgendeinem indizierten Attribut denselben Wert haben und andererseits aus Datensätzen, die über einen gemeinsamen Blockschlüssel in irgendeinem Attribut, im BI zusammen gruppiert wurden. Durch die Vermischung der Attributswerte, sowohl im RI als auch im BI, kann dadurch zwar die Pairs Completeness erhöht werden, da beispielsweise Attributsvertauschungen erkannt werden, je ähnlicher die Werte unterschiedlicher Attribute sind, desto mehr wird folglich die Pairs Quality gesenkt. Diese Verhalten korrespondiert mit der Kommutativität bei der Bildung von Blockschlüsseln, aus konjunktiven spezifischen Blockingprädikaten. Im Kontrast zum DySimII kann die Überlappung, beim Bauen des DNF Blocking Schema, jedoch mit Bedacht auf ausgewählte Attribute angewandt werden und wird nicht pauschalisiert. Die Möglichkeit der Überlappung von Attributswerten führt zu einer gravierenden Folge bei der Berechnung der Ähnlichkeit zwischen zwei Datensätzen. Da es in einem Block zu einer Vermischung von Werten unterschiedlicher Attribute kommen kann, ist es aufgrunddessen nicht möglich die vorausberechnete Ähnlichkeit einem bestimmten Attribut zuzuweisen. Diese müssten mit den tatsächlichen Attributen eines Datensatzes verglichen werden, um das korrekte Attribut zu finden. Der orginale Datensatz zum Nachschlagen steht dem DySimII jedoch nicht zur Verfügung, weshalb stattdessen die Ähnlichkeiten aufsummiert werden. Aus Abschnitt 2.5 ist allerdings bekannt, dass dabei für einen Klassifikator wertvolle Informationen verloren gehen, die entscheidend sind damit ein Match von einem Non-Match unterschieden werden kann. Die Entscheidung des DySimII Verfahrens, nur die Ähnlichkeiten aus Attributen, die einen gemeinsamen Block haben, in die Gesamtähnlichkeit mit einfließen zu lassen, reduziert den Ähnlichkeitsinformationswert weiter, da alle unähnlichen Attribute mit demselben Ähnlichkeitswert von 0 bestraft werden.

3.4.2 MDySimII

Eine weitere Einschränkung des DySimII Verfahrens ist die Vorgabe zu einer Enkodierfunktion pro Attribut, die genau ein Attribut liefert. Aufgrund dessen kann für DySimII kein DNF Blocking Schema verwendet werden. Im Folgenden wird ein Verfahren beschrieben, dass zum einen DNF kompatibel ist und zum anderen einige Nachteile, der Informationsdetailliertheit des Ähnlichkeitswertes, ausbessert. Dieser Ansatz wird Multi-Dynamic Similarity-Aware Inverted Index (MDySimII) genannt. Dabei steht das Multi für einen uneingeschränkten Multi-pass Ansatz. Das bedeutet, dass Blockschlüssel über mehrere Attribute generiert werden dürfen, dass nicht jedes Attribut zur Generierung

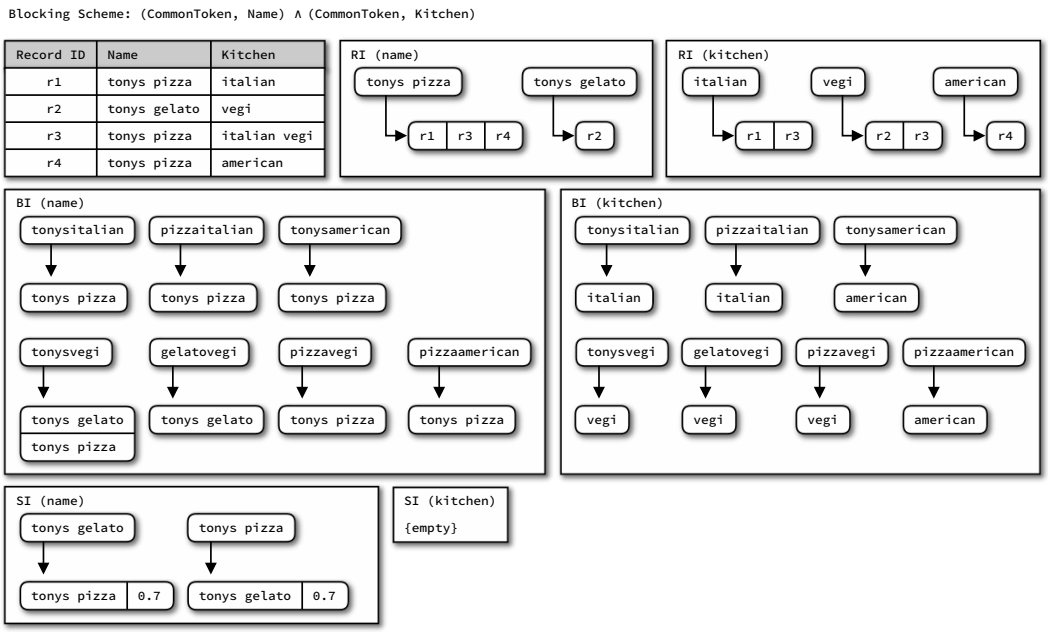

Abbildung 3.3 Ein beispielhafter MDySimII-Index, welcher aus der Tabelle links erzeugt worden ist. Die Beispieldatensätze enthalten das Namensattribut eines Restaurants und die Art der Küche. RI ist der Record Index, BI ist der Block Index, welcher aus dem Blocking Schema (CommonToken, Name) AND (CommonToken, Kitchen) erzeugt wurde. SI ist der Similarity Index.

genutzt werden muss und dass beliebig viele Blockschlüssel erzeugt werden können. Abbildung 3.3 zeigt beispielhaft die Indexstrukturen des MDySimII, welche aus der Tabelle und dem Blocking Schema (links oben) erzeugt wurden. Die RIs verknüpfen dabei jeweils ein Attribut mit den Datensatzidentifieren, die diesem Attribut entsprechen, etwa `tonys pizza` mit den Datensätzen `r1`, `r3` und `r4`. Die Blockschüssel für die BIs wurden durch Algorithmus 8 generiert. Für `r1` wurden, aus den Namenstoken `tonys` und `pizza`, sowie dem Token der Küchenart `italian`, die Blockschlüssel `tonysitalian` und `pizzaitalian` gebildet. Die beteiligten Attribute wurden, in ihren eigenen BI, in Blöcke mit den beiden Blockschlüsseln eingefügt. Der SI des jeweiligen Attributes wird angelegt, wenn in mindestens einem Block mehrere Attribute einfügt wurden. Das Beispiel zeigt die strikte Trennung von Attributswerten, sodass lediglich Werte desselben Attributes zusammen gruppiert werden können. Folglich ist der MDySimII in der Lage einen Vektor mit Ähnlichkeitswerten der einzelnen Attribute für Datensatze der Kandidatenmenge zurückzugeben.

Durch die erläuterten Anpassungen für den MDySimII ergeben sich einige Änderungen im Ablauf. In Algorithmus 9 ist die *Build-Phase* des MDySimII beschrieben. Zunächst werden die Index Datenstrukturen Record Identifier Index (RI), welcher alle Attribute speichert und diese ihren Datensätzen zuordnet, Block Index (BI), welcher Attribute anhand der Blockschlüssel gruppiert und Similarity Index (SI), welcher Attributsähnlichkeiten zwischen Attributen im gleichen Block hält, für jedes in der DNF vorkommende

Algorithm 9 MDySimII - Build

Input:
- Data set: D
- DNF Blocking Scheme: BS
- Fields used in BS: F
- Similarity functions: $S_i, i = 1 \cdots |F|$

Output:
- Index data structures: RI, BI, SI

1: **for** fields $f \in F$ **do**
2: Initialize $RI_f = \{\}, BI_f = \{\}, SI_f = \{\}$
3: **end for**
4: **for** records $r \in D$ **do**
5: **for** fields $f \in F$ **do**
6: insert $r.id$ into $RI_f[r.f]$
7: **end for**
8: **for** terms $t \in BS$ **do**
9: $bkvs = BlockingKeyValues(t, r)$
10: **for** $bkv \in bkvs$ **do**
11: **for** fields $f \in t.fields$ **do**
12: **if** $a \notin SI_f$ **then**
13: Append r.f to $BI_f[bkv]$
14: **for** attribute $a \in BI_f[bkv]$ **do**
15: $sim = S_f(r.f, a)$
16: Append $(r.f, sim)$ to $SI_f[a]$
17: Append (a, sim) to $SI_f[r.f]$
18: **end for**
19: **end if**
20: **end for**
21: **end for**
22: **end for**
23: **end for**

Algorithm 10 MDySimII - Query

Input:
- Query record: q
- DNF Blocking Schema: BS
- Fields used in BS as: F
- Similarity funcitons: $S_i, i = 1 \cdots n$

Output:
- Matches: M

1: Initialize dictionary $M = \{\}$
2: Insert q into Index
3: **for** fields $f \in F$ **do**
4: $ri = RI_f[q.f]$
5: **for** $r.id \in ri$ **do**
6: $M[(r.id, f)] = 1.0$
7: **end for**
8: $si = SI_f[q.f]$
9: **for** $(r.f, sim) \in si$ **do**
10: $ri = RI_f[r.f]$
11: **for** $r.id \in ri$ **do**
12: $M[(r.id, f)] = sim$
13: **end for**
14: **end for**
15: **end for**
16: return M

Attribut erzeugt (Zeilen 1-3). Als Nächstes werden alle Datensätze in D nacheinander den Indexstrukturen des MDySimII hinzugefügt. Dazu wird zuerst der Datensatzidentifier unter dem entsprechenden Attribut, in alle initialisierten RIs, eingefügt (Zeilen 5-7). Anschließend werden die disjunkten Ausdrücke der DNF einzeln betrachtet. Zu jedem Ausdruck werden für den Datensatz r die Blockschlüsselwerte $bkvs$ erzeugt (Zeile 9). Für jedes durch den aktuellen Ausdruck erfasste Attribut werden die Attributswerte von r, unter dem Blockschlüssel bkv, in den entsprechenden Block Index eingefügt (Zeilen 13). Nachdem ein Attribut in einen Block eingefügt wurde, werden analog zum ursprünglichen DySimII Verfahren, die Ähnlichkeiten zwischen $r.f$ und den bisherigen Attributen des Blocks ermittelt (Zeile 15). Der Algorithmus endet, wenn alle Datensätze $r \in D$ hinzugefügt wurden. Wird später in der *Query-Phase* ein einzelner Datensatz hinzugefügt, werden lediglich die Schritte in Zeile 5-22 durchgeführt.

Wird der MDySimII Index zur Bewertung eines DNF Ausdrucks für Algorithmus 7 gebaut, werden anstatt der Attributswerte die Datensatzidentifier in die Blöcke der BIs eingefügt. Zudem wird die Erzeugung der SIs ausgespart, da diese für die Bewertung nicht benötigt werden. Neben den BI Blöcken, werden zudem die RI Blöcke übergeben, da diese maßgeblich die Kandidatenmenge eines Anfragedatensatzes beeinflussen.

Die *Query-Phase* des MDySimII wird in Algorithmus 10 beschrieben. Dieser Algorithmus ist fast identisch zum DySimII Verfahren. Die Änderungen sind, dass im entsprechenden RI bzw. SI des Attributes nachgeschlagen und dass ein Ähnlichkeitsvektor erstellt wird, anstatt die einzelnen Ähnlichkeiten aufzusummieren. Aus Effizienzgründen erhebt der MDySimII, ebenso wie der DySimII, keine Ähnlichkeiten zwischen Attributen, die nicht zusammen gruppiert wurden. Durch das DNF Blocking Schema ist allerdings nicht mehr

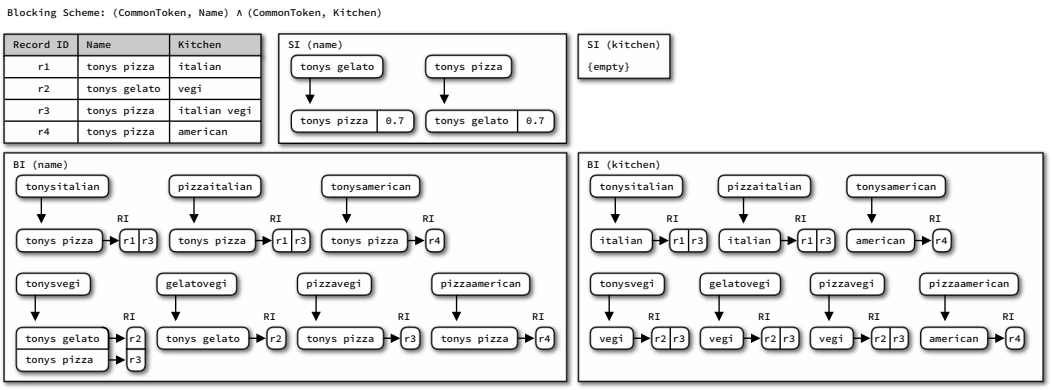

Abbildung 3.4 Ein beispielhafter MDySimIII-Index, welcher aus der Tabelle links erzeugt worden ist. Die Beispieldatensätze enthalten das Namensattribut eines Restaurants und die Art der Küche. BI ist der Block Index, welcher aus dem Blocking Schema (CommonToken, Name) AND (CommonToken, Kitchen) erzeugt wurde. SI ist der Similarity Index.

garantiert, dass jedes Attribut berücksichtigt wird, da der Ähnlichkeitsvektor der Kandidaten maximal in den Stellen besetzt ist, die auch im DNF Blocking Schema genutzt werden. Hierdurch entfällt eine komplette Attributsabdeckung, bei der Vorausberechnung der Ähnlichkeiten. Im schlimmsten Fall besteht das Blocking Schema nur aus einem einstelligen Ausdruck, wodurch das Blocking und die Vorausberechnung lediglich auf einem Attribut durchgeführt wird.

Das ursprüngliche Design des Similarity-Aware Inverted Index von Christen & Gayler [8] hat eine fundamentale Schwachstelle, die auch in den Varianten DySimII und MDySimII vorhanden ist. Diese Schwachstelle ist der Record Identifier Index (RI), welcher den Index enorm ausbremst, wenn ein Attribut nur wenige Auswahlmöglichkeiten bietet, beispielsweise Geschlecht oder Bundesland, bzw. wenn wenige Attributswerte besonders häufig vorkommen, beispielsweise der in Deutschland weit verbreitete Nachname `Müller`. Wenn ein solches Attribut im DNF Blocking Schema vorkommt, wird zwangsweise über den Record Identifier Index eine riesige Menge an Kandidaten selektiert, wovon nur ein Bruchteil tatsächlich relevant ist. Der schlimmste Fall tritt ein, wenn ein Datensatz ein Attribut hat, dass für alle Datensätze stets dasselbe ist. Beispielsweise das Geschlecht (männlich/weiblich), für einen Datensatz der nur weibliche Personen enthält. Dafür erzeugen alle Variationen des Similarity-Aware Inverted Index eine Kandidatenmenge mit 100% der Datensätze. Aufgrund der offensichtlich vielen False Positives in der Kandidatenmenge, wird der Algorithmus zur Bewertung von Ausdrücken, diejenigen mit diesen Attributen, sowohl alleinstehend als auch in Konjunktion mit anderen, sehr schlecht bewerten. Im ungünstigsten Fall wird dadurch ein Ausdruck verworfen, welcher für den BI qualitativ hochwertige und effiziente Blöcke erzeugt.

Algorithm 11 MDySimIII - Build

Input:
- Data set: D
- DNF Blocking Scheme: BS
- Fields used in BS as: F
- Similarity funcitons: $S_i, i = 1 \cdots n$

Output:
- Index data structures: BI, SI

```
1: for fields f ∈ F do
2:     Initialize BI_f = {}, SI_f = {}
3: end for
4: for records r ∈ D do
5:     for terms t ∈ BS do
6:         bkvs = BlockingKeyValues(t, r)
7:         for bkv ∈ bkvs do
8:             for fields f ∈ t.fields do
9:                 Add r.f to BI_f[bkv]
10:                ri = bi[r.f]
11:                Add r.id to ri
12:                BI_f[bkv] = ri
13:                Initialize inverted index list si = ()
14:                for attribute a ∈ bi do
15:                    if a ∉ SI_f then
16:                        sim = S_f(r.f, a)
17:                        Append (r.f, sim) to SI_f[a]
18:                        Append (a, sim) to si
19:                    end if
20:                end for
21:                SI_f[r.f] = si
22:            end for
23:        end for
24:    end for
25: end for
```

Algorithm 12 MDySimIII - Query

Input:
- Query record: q
- DNF Blocking Schema: BS
- Fields used in BS as: F
- Similarity funcitons: $S_i, i = 1 \cdots n$

Output:
- Matches: M

```
1: Initialize dictionary M = {}
2: Insert q into Index
3: for terms t ∈ BS do
4:     bkvs = BlockingKeyValues(t, r)
5:     for bkv ∈ bkvs do
6:         for fields f ∈ t.fields do
7:             bi = BI_f[bkv]
8:             for attribute r.f ∈ bi do
9:                 if q.f = r.f then
10:                    for identifier r.id ∈ bi[q.f] do
11:                        M[(r.id, f)] = 1.0
12:                    end for
13:                else
14:                    si = SI_f[q.f]
15:                    sim = si[r.f]
16:                    for identifier r.id ∈ bi[r.f] do
17:                        M[(r.id, f)] = sim
18:                    end for
19:                end if
20:            end for
21:        end for
22:    end for
23: end for
24: return M
```

3.4.3 MDySimIII

Der MDySimIII ist eine Modifikation des MDySimII Verfahrens, der dahingehend verändert wurde, dass die Anzahl der Kandidaten, auch bei spezifischen Blockingprädikaten mit wenigen Optionen bzw. häufig vorkommenden Werten, effizient eingeschränkt werden kann. Dadurch ist es möglich, im Gegensatz zur bisherigen Familie von Similarity-Aware Inverted Indizes, deutlich mehr spezifische Blockingprädikate, die zu einem guten Blocking Schema führen, zu betrachten. Damit dies möglich ist, wurde der globale bzw. attributsglobale Record Identifier Index (RI), in seiner bisherigen Form, aufgesplittet und in den Block Index (BI) verschoben. Der Similarity Index bleibt unverändert und verlinkt weiterhin Attribute eines Blockes mit ihren Ähnlichkeiten. In Abbildung 3.4 ist ein solcher Index aus den Datensätzen der Tabelle links oben und dem Blocking Schema `(CommonToken, Name)` $\wedge$ `(CommonToken, Kitchen)` erstellt worden. Ein Datensatz besteht aus den beiden Attributen Restaurantname `Name` und der Küchenart `Kitchen`. Anhand des Blocking Schema wurden durch Algorithmus 8 fünf Blockschlüssel generiert, nämlich `tonysitalian`, `tonysvegi`, `pizzaitalian`, `pizzavegi` und `gelatovegi`. Jeder Block beinhaltet weiterhin die Attribute, welche zum entsprechenden Blockschlüssel gehören. Die große Veränderung des MDySimIII ist , dass jedes Attribut eines Blockes einen eigenen Record Identifier Index besitzt.

Aufgrund der Änderungen am RI hat sich die Build-Phase an zwei Stellen leicht verändert. In Algorithmus 11 ist das Bauen des Index gezeigt. In grau sind die Schritte markiert, welche sich gegenüber Algorithmus 9 nicht verändert haben. Die erste Veränderung ist, dass das initiale Hinzufügen der Attribute, in ihren attributsspezifischen RI wegfällt, da

dieser so nicht mehr existiert. Die zweite Veränderung (Zeilen 10-12) ist beim Hinzufügen eines Attributes in einen Block. Nachdem das Attribut wie gewohnt in den Block eingefügt wurde, wird dessen RI geholt (Zeile 10), zu welchem anschließend der Datensatzidentifier hinzugefügt wird (Zeile 11). Wird der MDySimIII für den DNF Blocks Lerner gebaut, werden analog zum MDySimII lediglich die Datensatzidentifier in die Blöcke eingefügt. Die Verknüpfung mit den Attributen ist dabei irrelevant, da jeder Datensatz, der sich im Block befindet, ungeachtet seines zugehörigen Attributswertes in der Kandidatenmenge landet. Zudem wird auch hier die Erzeugung der SIs übersprungen.

Im Gegensatz zur Build-Phase hat sich die Query-Phase grundlegend verändert. Um an die Identifikatoren in den Blöcken zu gelangen, müssen für den Anfragedatensatz, die Blockschlüssel generiert werden (siehe Algorithmus 12). Diese werden nacheinander aus den Termen t erzeugt (Zeilen 3-4). Für jeden Blockschlüssel werden anschließend die entsprechenden Blöcke, der an dem Ausdruck beteiligten Attribute, betrachtet (Zeilen 5-7). Für jedes Attribut wird zunächst überprüft, ob dies dem eigenen entspricht. Ist dies der Fall, wird der Ähnlichkeitsvektor der Kandidaten im RI $bi[q.f]$, mit dem Ähnlichkeitswert 1.0 ergänzt (Zeilen 9-12). Falls die Attribute unterschiedlich sind, wird die Ähnlichkeit sim im SI nachgeschlagen (Zeile 15) und der Ähnlichkeitsvektor der Kandidaten im RI $bi[r.f]$, mit dem Ähnlichkeitswert sim ergänzt. Zum Schluss wird eine zum MDySimII identische Kandidatenmenge M zurückgegeben.

Anstatt wie vorher, alle Datensatzidentifier des gleichen Attributes global zu gruppieren, werden hier ausschließlich diejenigen in denselben RI eingefügt, die auch denselben Blockschlüssel haben. Beispielsweise werden in Abbildung 3.4, im BI des Namen, über den Blockschlüssel `tonysitalian` `r1` und `r3` zusammen einem RI zugeordnet, nicht jedoch `r4`, welcher keinen gemeinsamen Blockschlüssel zu `r1` oder `r3` hat. Die Anzahl der Blöcke sind identisch zum MDySimII Verfahren. Allerdings wird ein Attribut $r.f$ k-Mal mit seinem Datensatzidentifier verknüpft (vgl. MDySimII $|F|$-Mal und DySimII 1-Mal), wobei

$$k = \sum_{t \in BS} |BlockingKeyValues(t, r.f)|.$$

Beispielsweise `r1` viermal, `r2` fünfmal, `r3` neunmal und `r4` ebenfalls viermal. Das hat zur Folge, dass der MDySimIII mehr Speicherplatz als seine Vorgänger benötigt. Wie viel größer der Bedarf ist hängt von der konkreten Verteilung der Daten ab.

3.5 Lernen von Ähnlichkeitsmaßen

Aus der Vielfalt der möglichen Ähnlichkeitsmaße, gibt es keines das allen anderen klar überlegen ist. Es ist daher sehr domainabhängig welches Maß gute Ergebnisse liefert. Beim Vergleich von Datensätzen sind diese Domänen meist durch die unterschiedlichen Attribute getrennt. Daher ist es notwendig herauszufinden, für welches Attribute welche

Algorithm 13 PredictSimilarities(D, BS, S, P, N)

Input:
- Dataset D
- Blocking Scheme BS
- Fields used in BS: F
- Similarity functions: $S_i, i = 1 \cdots |F|$
- Set of positive pairs: P
- Set of negative pairs: N

Output:
- : $SIMS_i, i = 1 \cdots |F|$

Initialize list $SIMS = []$
for field $f \in F$ **do**
  Initialize bestScore = 0
  for similarity function $sim \in S$ **do**
    score = $FieldScore(BS, sim, field, P, N)$
    if $score > bestScore$ **then**
      $bestScore = score$
      $bestSim = sim$
    end if
  end for
  $SIMS[field] = bestSim$
end for
return $SIMS$

Algorithm 14 FieldScore(BS, sim, field, P, N)

Input:
- Blocking Scheme BS
- Similarity function: sim
- Attribute field: $field$
- Set of positive pairs: P
- Set of negative pairs: N

Output:
- field score: $score$

Initialize $y_{true} = [], y_scores = []$
for pair $(p_1.id, p_2.id) \in P \cup N$ **do**
  $p_1.f = D[p_1.id][field]$
  $p_2.f = D[p_2.id][field]$
  for term $t \in BS$ **do**
    if $field \in term$ **then**
      $p1_{bkv} = BlockingKeyValues(t, p_1.f)$
      $p2_{bkv} = BlockingKeyValues(t, p_2.f)$
      if $p1_{bkv} \cup p2_{bkv} \neq \emptyset$ **then**
        if $p \in P$ **then**
          Append 1 to y_{true}
        else
          Append 0 to y_{true}
        end if
        Append $sim(p_1.f, p_2.f)$ to y_{score}
      end if
    end if
  end for
end for
return $average\_precision\_score(y_{true}, y_{score})$

Ähnlichkeitsmetrik besonders gut funktioniert. Bevor ein Maß für die Qualität verschiedener Ähnlichkeitsmaße gefunden werden kann, muss das Problem der Vergleichbarkeit gelöst werden. Für zwei Strings a und b liefert der Jaccard-Koeffizient beispielsweise Werte zwischen 0 und 1, die Levenshtein-Distanz hingegen Werte zwischen 0 und $maxlen(a, b)$. Deshalb ist es notwendig die verschiedenen Ähnlichkeitsmaße zu normieren. Dafür wird das Intervall von 0 bis 1 gewählt, wobei 1 totale Übereinstimmung und 0 keine Übereinstimmung bedeutet. Dieses Intervall ist sinnvoll, da viele Klassifikatoren für diese Werte ihre Ausführungszeit optimiert können. Ähnlichkeitsmaße, die einen höheren Wert berechnen (je unähnlicher zwei Strings sind) werden als Abstandsfunktionen bezeichnet. Eine wichtige Eigenschaft von Abstandsfunktionen ist es, dass diese die Dreiecksungleichung in metrischen Räumen erfüllt. Der metrische Raum ist definiert als (X, d), wobei X hier die Menge an Wörtern einer Sprache ist und d eine beliebige Abstandsfunktion. Die Dreiecksungleichung sagt aus, dass der Abstand von x nach z nicht größer sein darf, als die Summe beliebiger Zwischenschritte über andere Wörter, x nach y und y nach z, daraus folgt $d(x, z) \leq d(x, y) + d(y, z)$, für alle $x, y, z \in X$. Bei der Normalisierung muss berücksichtigt werden, dass die Dreiecksgleichung nicht verletzt wird. Ansonsten ist nicht garantiert, dass sich alle normalisierten Werte im Intervall 0 bis 1 befinden. Um korrekt zu normalisieren muss für eine Abstandsfunktion, der größtmögliche Wert bestimmt werden. Dafür wird meist der länger der beiden Strings genutzt. Bei den Editierdistanzen, beispielsweise der Levenshtein-Distanz, ist dies die Anzahl der Zeichen im längeren String $maxlen(a, b)$. In diesem Fall wird der String in jeder Postition modifiziert, um a in b zu wandeln. Beispielsweise ist $Levenshtein(Liste, Baum) = 5 = maxlen(Liste, Baum)$, weil $(L \rightarrow B, i \rightarrow a, s \rightarrow u, t \rightarrow m, del\ e)$. Daraus folgt die Normalisierungsfunktion für Editierdistanzen:

$$sim_{norm}(sim, a, b) = 1 - \frac{sim(a, b)}{max\_sim\_val(a, b)}$$

Diese Normalisierungfunktion, anhand des größt möglichen Wertes der Ähnlichkeitsfunktion zwischen a und b, hat den Nachteil, dass die errechnetet Ähnlichkeiten z.T. zu optimistisch sind. Eine verbreitete Alternative, Abstandsfunktionen zu normalisieren, ist den größtmöglichen Wert anhand des kürzen Strings zu berechnen. Dadurch wird zwar überschwinglicher Optimismus, beim Normalisieren korrigiert, jedoch verstößt diese Art der Normalisierung gegen die Dreiecksungleichung.

Durch die Normalisierung ist der Similarity Lerner in der Lage für jedes Attribut das beste Ähnlichkeitsmaß auszuwählen. Algorithmen 13 und 14 beschreibt das Vorgehen. In Abschnitt 2.4 wurde gezeigt, dass die verschiedenen Ähnlichkeitsfunktionen oft mindestens einen Parameter haben. Da diese Parameter teilweise die Dreiecksungleichung aufheben, beispielsweise die Gewichte bzw. Kosten bei den Editierdistanzen, prüft der Similarity Lerner nur die Ähnlichkeitsfunktionen mit den Standardparametern. Von der Engine bekommt er dazu eine Auswahl an Ähnlichkeitsfunktionen S, das Blocking Schema BS, sowie die Ground Truth P und N. Der Algorithmus prüft nur die Attribute, welche im Blocking Schema enthalten sind (Zeile 2). Auf das gewählte Attribut werden die Ähnlichkeitsfunktionen in S (Zeile 5) durch den Algorithmus 14 angewandt und bewertet (Zeile 5). Die Ähnlichkeitsfunktion, die für ein Attribut die beste Bewertung liefert, wird für den Indexer als Ähnlichkeitsfunktion ausgewählt. Die Liste der ausgewählten Attribute wird abschließend an die Engine zurückgegeben. Für die Bewertung der Ähnlichkeiten eines Attributes wird die Average Precision Score genutzt. Diese wird berechnet anhand der Fläche unter der Precision-Recall Kurve. Die Average Precision Funktion benötigt zur Berechnung für jedes Paar der Ground Truth die Klasse, Match oder Non-Match, sowie den errechneten und normalisierten Ähnlichkeitswert. Diese Werte werden in den beiden Array y_{true}, für die Klassen und y_{score}, für die Ähnlichkeitswerte, gesammelt (Zeile 1). Für jedes Paar $(p_1.id, p_2.id)$ der Ground Truth, wird zunächst das entsprechende Attribute aus dem Datensatz D geholt (Zeile 2). Anhand der Attribute werden für jeden Ausdruck im Blocking Schema BS, die Blockschlüssel $p_{1_{bkv}}$ und $p_{2_{bkv}}$ erzeugt (Zeilen 7, 8). Falls die Schnittmenge der beiden Blockschlüsselmengen mindestens einen gemeinsamen Blockschlüssel beinhaltet (Zeile 9), wird die Ähnlichkeit zwischen den beiden Attributes des Paares berechnet 15. Der Ähnlichkeitswert wird in die Liste y_{scores} aufgenommen. Ist die Schnittmenge leer, so haben die Attribute des Paares keinen gemeinsamen Block und werden folglich vom Indexer nicht miteinander verglichen, weshalb der Vergleich auch hier übersprungen wird. Zusätzlich wird der Liste y_{true} eine 1 angefügt, wenn das Paar ein Match ist bzw. eine 0, wenn das Paar ein Non-Match ist. Abschließend wird die Average Precision Score berechnet und zurückgegeben.

3.6 Lernen der Hyperparameter der Klassifikatoren

Für jeden Klassifikator sollen, anhand eines geeigneten Qualitiätsmaßes, die besten Parameter bestimmt werden. Die einfachste Möglichkeit dafür ist eine Grid Search. Diese erzeugt ein Parameternetz aller möglichen Parameterkombinationen und sucht dieses vollständig ab. Dadurch ist zwar sichergestellt, dass die besten Parameter gefunden werden, je mehr Parameter es gibt, desto länger dauert diese Suche allerdings. Eine einfache Möglichkeit den Suchraum zu reduzieren ist, per Zufall nur eine bestimmte Anzahl an Parameterkombinationen zu bestimmten und nur diese zu vergleichen. Neben diesen beiden dynamischen Methoden zum Lernen der Hyperparamter, welche modellübergreifend funktionieren, gibt es in Scikit-learn auch spezialisierte Parametersuchen, beispielsweise für Logistic Regression, welche effizienter die Parameter für ihr Klassifikationsmodell finden. Der Fusion-Lerner sucht für jeden Klassifikator seperat die besten Parameter. Dabei wird das Ergebnis der besten Konfiguration mit den besten Konfigurationen anderer Klassifikatoren verglichen. Zum Schluss wird der Klassifikator ausgewählt, dessen beste Konfiguration, bei gegeben Qualitätsmaß das beste Ergebnis liefert.

Egal welches Suchverfahren genutzt wird, wichtig ist, dass beim Vergleichen, Validieren und Auswählen der Parameter bzw. der Modelle darauf geachtet wird, dass das Modell nicht überanpasst und dadurch ausschließlich auf den Trainingsdaten gute Ergebnisse erzielt werden. Um dies zu unterbinden, werden Kreuzvalidierungsverfahren genutzt, welche ebenfalls in Scikit-learn implementiert sind. Dabei unterscheidet man zwischen vollständiger Kreuzvalidierung und nicht-vollständiger Kreuzvalidierung. Ein Beispiel für die vollständige Kreuzvalidierung ist Leave-one-out. Dabei werden aus den Trainingsdaten mit n Objekten n Untermengen gebildet, bei welchen jeweils ein Element fehlt. Eine dieser Menge wird als Validierungsmenge ausgewählt, anhand welcher ein trainiertes Modell überprüft wird. Die anderen werden als Trainingsdaten genutzt. Das Verfahren wird n-Mal wiederholt, bis jede Menge als Validierungsmenge genutzt wurde. Das Ergebnis ist das Mittel aller Durchläufe. Da dies z.T. sehr lange dauert gibt es unvollständige Verfahren, wie das K-Fold. Dieses bildet zufällig k gleichmächtige Untermengen der Trainingsdaten. Eine dieser Mengen wird, analog zum Leave-one-out, zum Validieren ausgewählt. Die anderen $k - 1$ Mengen werden als Trainingsdaten genutzt. Das ganze wird k-Mal wiederholt, bis jede Menge einmal als Validierungsmenge genutzt wurde. Das Ergebnis der k Durchläufe wird ebenfalls gemittelt und als ein Wert zurückgegeben. Ist $k = n$ entspricht K-Fold dem Leave-one-out Verfahren. Eine beliebte Erweiterung des K-Fold ist der Stratified K-Fold. Dieser unterscheidet sich lediglich in der Generierung der Untermengen. Dabei werden Objekte ebenfalls zufällig aus der Trainingsmenge selektiert, jedoch wird darauf geachtet, dass das Verhältnis der Klassenzugehörigkeit der Objekte bestehen bleibt. Befinden sich in der Ausgangstrainingsmenge, beispielsweise 30% Matches und 70% Non-Matches, dann haben alle k Untermengen ebenfalls dieses 30% zu 70% Verhältnis. Damit wird sichergestellt, dass jede Untermenge

eine gute Repräsentation des Ganzen ist und folglich die Ergebnisse mehr Aussagekraft haben.

Hauptverantwortlich für die Qualität eines Modells ist die Ground Truth. Je mehr herausfordernde Matches und Non-Matches diese enthält, desto genauer kann ein Modell trainiert werden. Herausfordernd sind Paare, wo nicht offensichtlich ist, ob diese zu den Matches oder Non-Matches gehören. Allerdings kann für bestimmte Klassifikatoren die Größe der Ground Truth dazu führen, dass die Effizienz stark beeinträchtigt wird. Dies ist der Fall bei einer SVM, weil durch eine größere Ground Truth entsprechend mehr Stützvektoren erzeugt werden, die bei jeder Klassifikation verglichen werden müssen. Eine Lösung für dieses Problem ist das sog. Subsampling. Hierbei wird ein Kompromiss zwischen Qualtität und Effizienz getroffen, indem die Anzahl der Paare der Ground Truth reduziert wird. Soll beim Subsampling die Repräsentativität der Ground Truth erhalten bleiben, können Paare aus den Matches bzw. Non-Matches nach ihrer Ähnlichkeitsverteilung, wie in Abschnitt 3.1, ausgewählt werden.

4 Umsetzung eines selbstkonfigurierenden Systems

In diesem Kapitel wird zunächst das Design des selbstkonfigurierenden Systems beschrieben. Anschließend wird die Implementierung in Bezug auf Optimierungen betrachtet, die nötig sind damit große Datenmengen verarbeitet werden können.

4.1 Design

In diesem Abschnitt werden zunächst die Prozesse eines sich selbst konfigurierenden Entity Resolution System für dynamische Datenquellen, zur Bearbeitung von Anfrageströme, beschrieben. Anschließend wird die zu lernende Konfiguration formal definiert. Danach werden die Komponenten des Systems vorgestellt und deren Schnittstellen beschrieben. In Bezug auf den jeweiligen Prozess werden dann die Details zu den Komponenten erläutert, falls diese noch nicht aus Kapitel 3 bekannt sind.

4.1.1 Prozesssicht

Die in Kapitel 2 und 3 vorgestellten Verfahren, für dynamische Entity Resolution, trennen zwischen Build-Phase und Query-Phase. Diese Trennung wird auch für das selbstkonfigurierende System aufrecht erhalten. Zusätzlich gibt es noch eine weitere Phase zum Erlernen der Konfiguration, im Folgenden als *Fit-Phase* bezeichnet. Je nach Phase befindet sich bzw. wechselt das System in einen von drei Zuständen, die in Abbildung 4.1 dargestellt sind. Ein neu erzeugtes System ist *unangepasst* und kann durch das Lernen

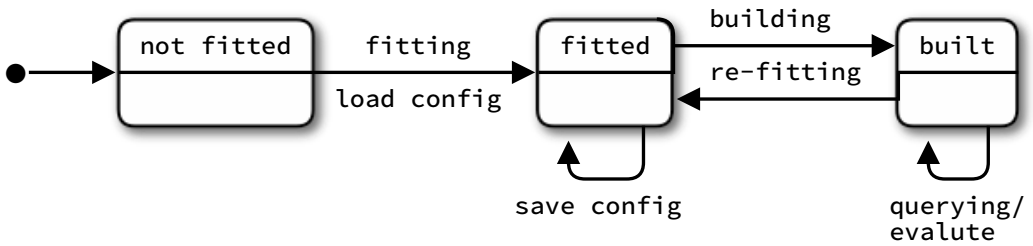

Abbildung 4.1 Zustandsdiagramm des selbstkonfigurierenden System. Lernen der Konfiguration versetzt das System von unangepasst nach angepasst. Wurde der Index gebaut, ist das System im Zustand gebaut und kann Anfrage entgegennehmen.

der Konfiguration (engl. fitting) in den Zustand *angepasst* wechseln. Alternativ kann der Zustandsübergang durch das Laden einer bereits gelernte Konfiguration durchgeführt werden. Anhand dieser Konfiguration kann der Index auf einem initialen Datenbestand gebaut (engl. building) werden. Danach befindet sich das System im Zustand *gebaut*. In diesem Zustand kann die eigentliche Entity Resolution, durch stellen von Anfragen aus einem Datenstrom (engl. querying), durchgeführt werden. Da die Möglichkeit besteht jede Anfrage in den Datenbestand (den Index) aufzunehmen, liegen nach einer gewissen Zeit genügend neue Daten vor, sodass sich auf Basis derer auch die optimale Konfiguartion verändert haben kann. Während des erneuten Lernens (engl. refitting) können weiterhin Anfragen beantwortet werden. Wenn der Lernvorgang abgeschlossen ist, muss der Index erneut gebaut werden, bevor das System wieder anfragen entgegen nehmen kann. Wenn Komponenten für das System entwickelt werden, ist es notwendig deren Qualität und Effektivität auszuwerten. Weshalb das System im Entwicklungsbetrieb entsprechende Metriken erheben und auswerten kann. Die Auswertung erfolgt nachdem mindestens eine Anfrage durchgeführt wurde.

In der Fit-Phase nimmt die Engine die Konfiguration des Systems vor. Eine Konfiguration ist ein Tupel (BS, S, M) bestehend aus dem Blocking Schema, den Ähnlichkeitsfunktionen und dem Klassikationsmodell. Die Teilkonfigurationen werden anhand einer Ground Truth erlernt, diese ist definert als $GT = (P, N)$ und ist ebenfalls ein Tupel, dass sich in die Menge der positive Datensatzpaare, die tatsächlichen Matches (true positives), sowie die Menge der negativen Datensatzpaare, die tatsächlichen Non-Matches (true negatives) teilt. Ein Datensatz wird definiert als n-Tupel, wobei n die Anzahl der Attribute ist. Ein Tupel hat die Form $t = (a_1, a_2, \ldots, a_n)$. Ein Attribut hat eine feste Position im Tupel, die als Feld oder Datenfeld f bezeichnet wird. Ein Datensatzensatzpaar ist definiert als $p = (t_j, t_k), j \neq k$, wobei j und k zwei beliebige Tupel desselben Datensatzes sein können. Weiterhin gilt $\forall p \in P, p \notin N$ und umgekehrt $\forall p \in N, p \notin P$. Das Blocking Schema entspricht der Definition aus Abschnitt 2.3.3, $BS = (term_1 \wedge \cdots \wedge term_j) \vee \cdots \vee (term_k \wedge \cdots \wedge term_n)$. Eine Ähnlichkeitsfunktion wird während des Lernens der Konfiguration mit einem Attribut verknüpft. Die Menge der gelernten Ähnlichkeitsfunktionen werden entsprechend als Tupel angegeben $S = (f_1, sim), \ldots, (f_n, sim)$. Die Ähnlichkeitsfuntion sim ist eine von m möglichen Ähnlichkeitsfunktionen $sim_1, \ldots, sim_m$, die vom System implementiert wurden. Das Klassifikationsmodell M ist spezifisch für den eingesetzten Klassifikator und entspricht, bespielsweise einem trainierten Entscheidungsbaum.

4.1.2 Komponentenmodell

Die Engine ist das Herzstück des selbstkonfigurierenden Systems und besteht aus einzelnen Komponenten, die bei Schnittstellenkompatibilität beliebig ausgetauscht werden. Die Komponenten und Schnittstellen der Engine sind in Abbildung 4.2 dargestellt.

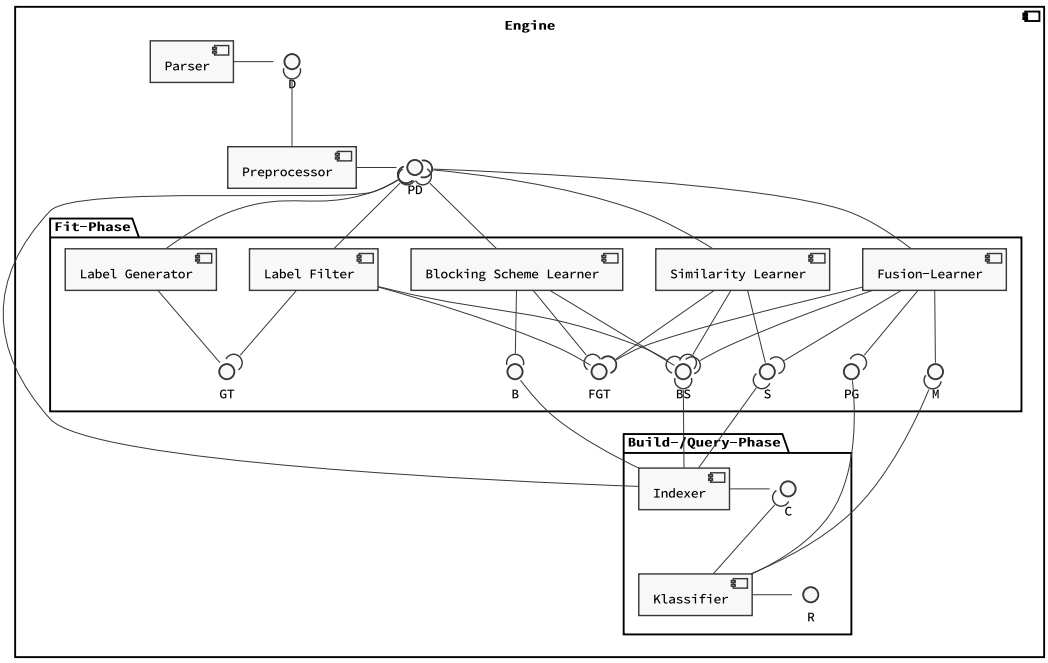

Abbildung 4.2 Komponentenmodell des selbstkonfigurierenden Systems. Bestehend aus dem Ground Truth Generator, dem Blocking Scheme Lerner, dem Similarity Lerner und dem Fusion-Lerner, welche für das Erlernen der Konfiguration (Fit-Phase) nötig sind, dem Indexer, welcher anhand der gelernten Konfiguration gebaut wird und dem Klassifikator. Der Parser, um Daten einer Datenquelle zu laden und der Präprozessor, um die geladenen Daten für Entity Resolution zu manipulieren

- **Parser**. Der Parser liest Datensätze aus einer Datenquelle und bietet eine Menge von Tupeln D an.
- **Präprozessor**. Der Präprozessor vorverarbeitet jedes Attribut jedes Datensatzes aus D in einer Pipeline, anhand einer Reihe von benutzerdefinierten Operationen, welche sequentiell angewendet werden. Ein einfaches Beispiel ist eine Rechtschreibprüfung. Das Ergebnis ist die vorverarbeitete Menge an Tupeln PD.
- **Label Generator**. Der Label Generator erzeugt eine geeignete Ground Truth GT zum Einstellen der Parameter in den folgenden Komponenten. Er konsumiert dazu die vorverbeiteten Tupel PD.
- **Blocking Schema Lerner**. Der Blocking Schema Lerner erzeugt eine Blocking Schema BS in distributiver Normalform nach [17]. Zur Bewertung eines Ausdrucks, konsumiert er die generierten Blöcke B eines Indexers.
- **Label Filter**. Der Label Filter ist fester Bestandteil der Engine und modifiziert die Ground Truth GT, indem nur Paare durchgelassen werden, die zum Blocking Schema BS passen. Das Ergebnis ist die gefilterte Ground Truth FGT.
- **Similarity Lerner**. Der Similarity Lerner bestimmt für jedes Attribut eine geeignete Ähnlichkeitsfunktion S.

- **Fusion-Lerner**. Der Fusion-Lerner ermittelt die besten Parameter für den verwendeten Klassifikator. Von diesem erhält der Fusion-Lerner mögliche Parameter PG, anhand welcher das Klassifikationsmodell M trainert wird.
- **Indexer**. Der Indexer wendet ein Blocking Verfahren auf die vorverarbeiteten Daten PD an und bietet bei einer Anfrage eine Kandidatenliste C mit möglichen Duplikaten an.
- **Klassifikator**. Der Klassifikator ordnet die Kandidaten aus C in Matches und Non-Matches. Die Menge an klassifizierten Matches R ist das Ergebnis einer Anfrage.

Die Hauptaufgabe der Engine ist es, die Interaktionen zwischen den Komponenten zu steuern. Dazu werden im simpelsten Fall die Daten von einer Komponente zur nächsten weitergereicht. Zum Teil muss die Engine zunächst jedoch die Rückgabewerte für die nächste Komponente aufbereitet (vgl. Label Filter). Die Engine dient weiterhin als Schnittstelle für den Benutzer. Alle drei Phasen haben den Schritt der Vorverarbeitung gemeinsam.

Vorverarbeitung

Die Vorverarbeitung der Daten ist in allen drei Phasen notwendig und macht die Datensätze robuster gegenüber Missklassifikationen, indem offensichtliche Fehler korrigiert und eventuelle, für die Identifikation von Entitäten irrelevante, Varianzen bereinigt weren. In Abbildung 4.3 sind die beteiligten Komponenten Parser und Präprozesser mit ihren Aktivitäten visualisiert. Jede Phase beginnt mit der Auswahl des korrekten Parsers durch die Engine.

Parser. Der Parser ist eine einfache Komponente, welche Datensätze aus einer Datenquelle liest und eine Menge von Tupeln D an die Engine übergibt. Je nach Phase kann die Datenquelle ein beliebiges Format haben, weshalb für jede Phase ein eigener Parser bestimmt werden kann. Für die *Fit-* und *Build-Phase*, wo große Datenmengen bearbeitet werden, liest der Parser beispielsweise aus einer CSV-Datei oder selektiert die Datensätze aus einer Datenbank. Währenddessen werden in der *Query-Phase* nur einzelne oder kleine Datenmengen gelesen, weshalb der Parser hier aus einer Message Queue (MQ) Datensätze erhalten könnte. Während der *Fit-Phase* hat der Parser zudem dafür Sorge zu tragen, dass der Engine die Attribute des Datensatzes, sowie deren Datentypen bekannt gemacht werden. Anhand der Datentypen können die Komponenten der Fit-Phase effizienter eine gute Konfigurationen bestimmen. Wenn der Parser diese Information nicht bereitstellt, werden alle Attribute als Zeichenketten behandelt.

Präprozessor. Der Präprozessor bzw. die Präprozessor-Pipeline besteht aus einer Reihe von Funktionen, die nacheinander auf alle Tupel aus D angewandt werden, um diese für die Entity Resolution vorzubereiten und robuster zu machen. Je nach Datentyp des Attri-

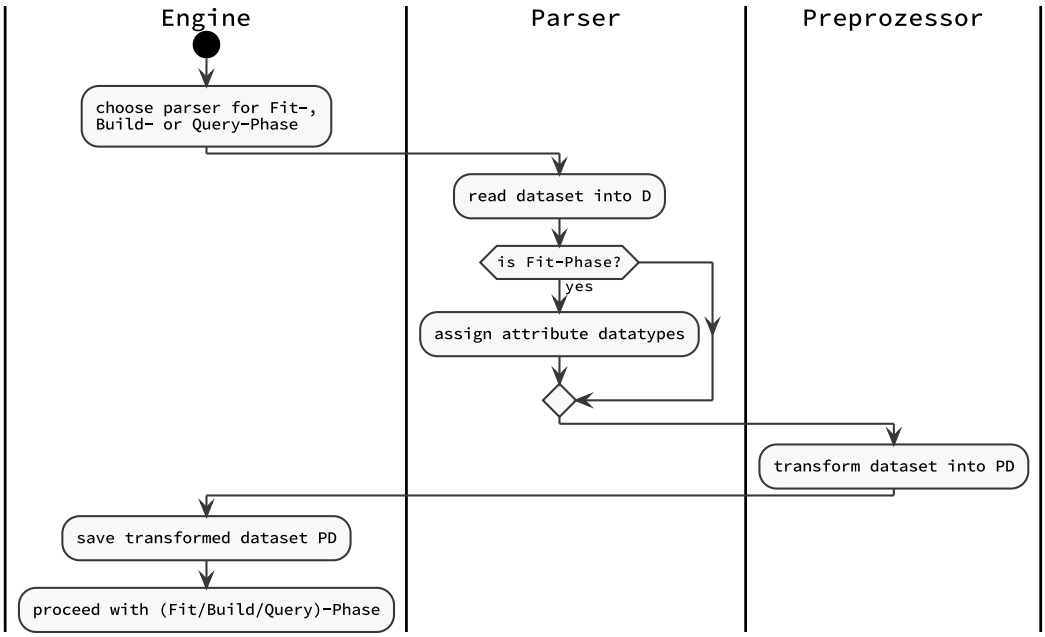

Abbildung 4.3 Aktivitätsdiagramm der Vorverarbeitung. Der Parser liest einen Datensatz, welcher vom Präprozessor transformieren wird. Der transformierte Datensatz wird von der Engine abgespeichert.

buts wird dabei eine andere Pipeline verwendet. Dieselbe Präprozessor-Pipeline muss in allen drei Phasen verwendet, damit die vorverarbeiteten Datenmenge PD stets die gleichen Charakteristiken aufweist. Werden vom Benutzer keine Operationen vorgegeben, beschränkt sich die Pipeline auf generische Modifkationen. Der Standardpräprozessor der Engine ist automatisiert lediglich in der Lage Zeichenketten in Kleinschreibweise zu konvertiert. Andere Operationen wie das Entfernen von Stopwörtern[1] (z.B. *und*, *oder*) benötigt das Kontextwissen über die Sprache der Attribute. Während die Sprache noch relativ leicht ermittelt werden kann, gibt es andere Vorverabeitungen, die aktuelle Daten benötigen. Ein komplexere domänenspezifische Anwendung hierfür ist, beispielsweise die Überprüfung der postalischen Adresse, welche zum einen länderspezifische Daten benötigt und zum anderen auch ständig auf dem aktuellen Stand gehalten werden muss. Neben zusätzlichen Funktionen muss der Benutzer auch die Reihenfolge der Funktionen vorgeben. Die Reihenfolge ist wichtig, da beispielsweise das Resultat einer Konvertierung in Kleinschreibweise von einer Rechtspreibprüfung in Teilen wieder aufgehoben werden kann.

Die vorverarbeiteten Tupel PD des Präprozessors, werden abschließend abgespeichert, um zu einem späteren Zeitpunkt geladen zu werden.

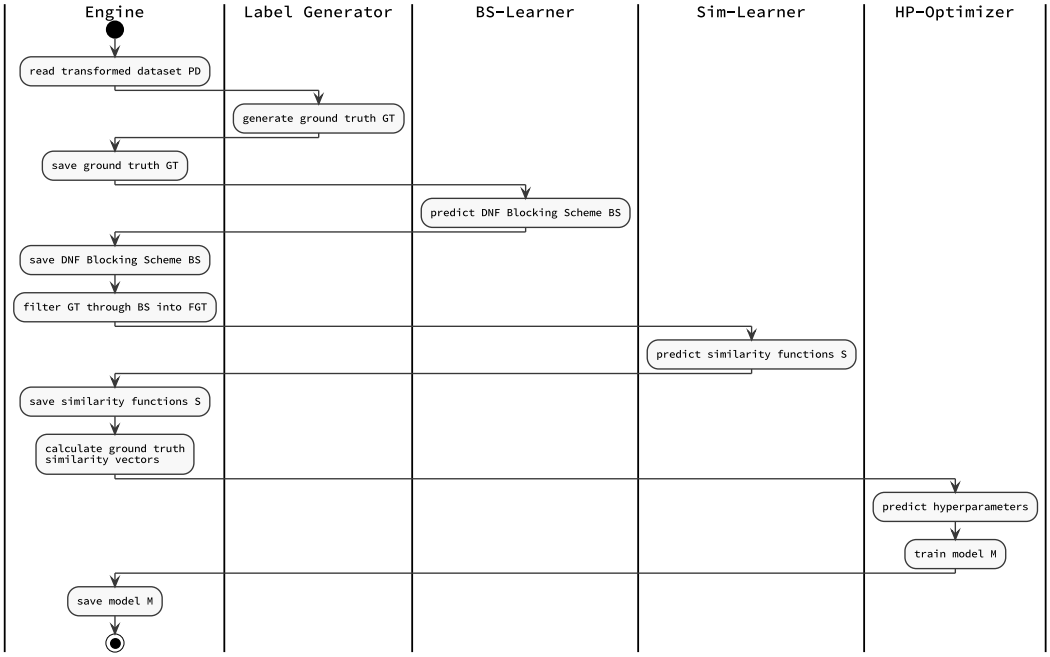

Abbildung 4.4 Aktivitätsdiagramm der Fit-Phase. Die Engine kontrolliert den Datenfluss zwischen den Komponenten, speichert Konfigurationen und bereitet Daten für Komponenten auf. Der Label Generator erzeugt die Ground Truth, durch welche ein DNF-Blocking Schema vom BS-Lerner erzeugt wird. Auf einer durch das Blocking Schema gefilterten Liste werden anschließend die Ähnlichkeitsfunktionen bestimmt. Anhand dieser Funktionen können Ähnlichkeitsvektoren auf der Ground Truth berechnet werden und vom Fusion-Lerner dadurch die Hyperparameter für den Klassifikator bestimmt, sowie abschließend das Klassifikationsmodell trainiert werden.

Fit-Phase

Die für die Fit-Phase relevanten Komponenten und Schnittstellen sind in Abbildung 4.2 in der Box “Fit-Phase” gruppiert. Bei großen Datensätzen kann diese Phase sehr lange dauern, weshalb die Engine die Teilkonfigurationen der Komponenten direkt sichert. Dadurch kann die Fit-Phase im Falle eines Abbruchs, z.B. durch einen Systemneustart, fortgesetzt werden und nur die unterbrochene Komponente muss wiederholt werden. Wurde die Fit-Phase abgeschlossen, ist es möglich die ermittelte Konfiguration einzulesen, wodurch die Fit-Phase übersprungen wird. Abbildung 4.4 zeigt das Aktivitätsdiagramm der Fit-Phase, ohne existierende Konfiguratation.

Label Generator. Der Label Generator erzeugt die Ground Truth GT, in Form von klassifizierten Matches und Non-Matches, für später in der Fit-Phase folgenden Komponenten. Dazu nutzt der Label Generator die vorverarbeiteten Tupel PD und bildet Datensatzpaare (Abbildung 4.4). Dabei gibt es zwei Ausprägungen. In der ersten Ausprägung erhält der Label Generator vorklassifizierte Matches für den vom Parser eingelesenen Datensatz (vgl. Abschnitt 3.1). In der zweiten Ausprägung stehen dem Label Generator keine

[1] Häufig auftretende Wörte, ohne Relevanz für die Erfassung des Inhaltes.

Algorithm 15 FilterGT(BS, D, P, N, AN, max_n)

Input:
- Blocking Scheme: BS
- Dataset: D
- Set of positive pairs: P
- Set of negative pairs: N
- Set of all generated negative pairs: AN
- Maximum Non-Duplicate Pairs: max_n

Output:
- Set of filtered positive pairs: fP
- Set of filtered negative pairs: fN
- Predictions: y_{pred}

1: Initialize empty sets $fP = ()$, $fN = ()$
2: **for** pair $(p_1.id, p_2.id) \in P$ **do**
3: **if** $HasCommonBlock(BS, D, (p_1.id, p_2.id))$ **then**
4: Append $(p_1.id, p_2.id)$ to fP
5: **end if**
6: **end for**
7: **for** pair $(p1, p2) \in N$ **do**
8: **if** $HasCommonBlock(BS, D, (p_1.id, p_2.id))$ **then**
9: Append $(p_1.id, p_2.id)$ to fN
10: **end if**
11: **end for**
12: **while** $|fN| < max_n$ and $|AN| > 0$ **do**
13: Draw pair $(p_1.id, p_2.id)$ from AN
14: **if** $HasCommonBlock(BS, D, (p_1.id, p_2.id))$ **then**
15: Append $(p_1.id, p_2.id)$ to fN
16: **end if**
17: **end while**
18: return fP, fN

Algorithm 16 HasCommonBlock(BS, D, p)

Input:
- Blocking Scheme: BS
- Dataset: D
- Pair: $p = (p_1.id, p_2.id)$

Output:
- True if p has common block, false otherwise

1: Initialize empty sets $p1_{bkvs} = (), p2_{bkvs} = ()$
2: **for** term $t \in BS$ **do**
3: $r_1 = D[p_1.id], r_2 = D[p_2.id]$
4: Add $BlockingKeyValues(t, r_1)$ to $p1_{bkvs}$
5: Add $BlockingKeyValues(t, r_2)$ to $p2_{bkvs}$
6: **end for**
7: **if** $p1_{bkvs} \cup p2_{bkvs} \neq \emptyset$ **then**
8: return True
9: **else**
10: return False
11: **end if**

vorklassifizierten Matches zur Verfügung. Weshalb die Ground Truth vollständig automatisiert bestimmt werden muss (vgl. Abschnitt 3.1). Ein Label Generator kann beide Ausprägungen implementieren. Falls nur die erste Ausprägung implementiert ist, kann die Engine, ohne existierende Ground Truth, die Fit-Phase nicht durchführen. Sollte nur die zweite Ausprägung vorhanden sein, werden die vorklassifizierten Matches ignoriert.

Blocking Schema Lerner. Der Blocking Schema Lerner ermittelt ein Blocking Schema in disjunktiver Normalform nach [17], welches in Abschnitt 3.2 vorgestellt wurde. Dafür benötigt der Blocking Schema Lerner die vorverarbeiteten Tupel PD, sowie die Ground Truth Daten GT des Label Generators. Zudem werden die Blöcke des genutzten Indexers IX zu jedem, zu analysierenden Audruck, benötigt. Daraus wird schlussendlich ein DNF Blocking Schema BS gebildet.

Label Filter. Anhand des Blocking Schema werden die Kandidaten, welche für die Entity Resolution infrage kommen, eingeschränkt. Damit der Similarity Lerner und der Fusion-Lerner ihre Konfiguration lediglich auf relevanten Paaren ermitteln, muss die Ground Truth durch das Blocking Schema gefiltert werden. Das Filtern der Ground Truth auf Basis des ermittelten Blocking Schema wird von der Engine durchgeführt bevor der Similarity Lerner aufgerufen wird (siehe Algorithmus 15). Dabei werden nacheinander alle Matches und Non-Matches der Ground Truth betrachet (Zeilen 2, 7). In Algorithmus 16 werden für jedes Paar $(p_1.id, p_2.id)$ die Blockschüssel, anhand des gegeben Blocking Schema, generiert (Zeilen 4, 5). Gibt es dabei eine Überlappung, dann gibt es für mindestens ein Attribut einen gemeinsamen Block, in welchem das Paar zusammen vorkommt. In diesem Fall gibt der Algorithmus “Wahr” zurück (Zeile 8). Gibt es keine Überlappung wird “Falsch” zurückgegeben (Zeile 10). Wurde durch Algorithmus

16 festgestellt, dass ein Match bzw. ein Non-Match einen gemeinsamen Blockschlüssel hat, dann werden diese zur gefilterten Ground Truth fP oder fN hinzugefügt (Zeilen 4, 9). Da das Ziel des Blocking Schema ist, möglichst nur gleiche Entitäten zu gruppieren, werden beim Filtern sehr viele Non-Machtes, im schlimmsten Fall alle, herausgefiltert. Dadurch ist die gefilterte Ground Truth zugunsten der Matches unbalanciert. Damit der Similarity Lerner und der Fusion-Lerner dennoch eine sinnvolle und aussagekräftige Konfiguration ermitteln können, werden die Non-Matches künstlich angereichert. Dazu werden die vom Label Generator zuvor verworfenen Non-Matches AN benötigt. Es wird nun versucht zur Ground Truth hinzuzufügen, indem wie davor eine Überlappung der Blockschlüssel gesucht wird (Zeile 14). Gibt es eine Überlappung wird das Paar zu fN hinzugefügt (Zeile 15). Dies wird solange wiederholt, bis die Ground Truth max_n Non-Matches beinhaltet oder die gesamte Menge der Non-Matches des Label Generators erschöpft ist (Zeile 12).

Similarity Lerner. Der Similarity Lerner bestimmt aus einer Menge von Ähnlichkeitsfunktion für jedes Attribut die geeigneste, nach dem Verfahren aus Abschnitt 3.5. Dabei werden nur Attribute betrachtet die einen gemeinsamen Blockschlüssel durch das Blocking Schema BS erhalten. Zur Bewertung werden die Datensatzpaare der gefilterten Grund Truth FGT genutzt. Das Ergebnis ist die Tupelliste S, welche jeweils ein Ähnlichkeitmaß mit Datenfeld verknüpft.

Fusion-Lerner. Der Fusion-Lerner ermittelt für einen gegebenen Klassifikator die Parameter, die das Modell mit der besten Bewertung erzeugen. Bevor der Fusion-Lerner aufgerufen werden kann, erzeugt die Engine für jedes Paar der gefilterte Ground Truth FGT, anhand der Ähnlichkeitsfunktionen des Similarity Lerners S, einen Ähnlichkeitsvektor pro Paar. Die Ähnlichkeitsvektoren sind dabei nur an den Stellen besetzt, in welches die Attribute des Paares einen gemeinsamen Block im Blocking Schema BS haben. Die Ähnlichkeitsvektoren werden dann vom Fusion-Lerner genutzt, um ein Modell mit gegebenen Parametern zu trainieren. Das Parameternetz PG muss von der Klassifikatorkomponente bereitgestellt werden, beispielsweise die maximale Tiefe eines Decision Tree. Um einen optimalen Klassifikator für die Eingabedaten zu bekommen ist es, abgesehen von der Parameterliste, möglich eine Liste von verschiedenen Klassifikatoren anzugeben, beispielsweise einen Decision Tree und eine SVM. Das Ergebnis des Fusion-Lerners ist das Modell M eines Klassifikators mit den Parametern, die die beste Bewertung erzielt haben.

Build-Phase

Die Build-Phase dient der Vorbearbeitung der Daten, bevor das selbstkonfigurierte ER-System seinen Betrieb aufnehmen kann. Dazu wird der komplette Datenbestand, in welchem Entitäten gesucht werden sollen, betrachtet. Nachdem diese durch die Vorverar-

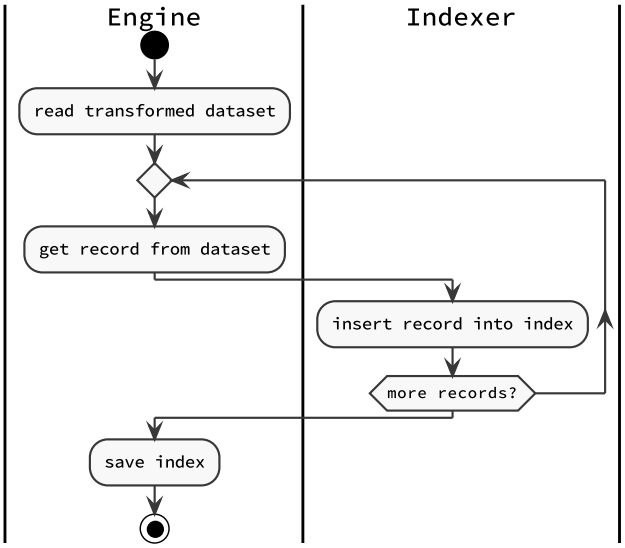

Abbildung 4.5 Aktivitätsdiagramm der Build-Phase. Der liest alle vorverarbeiteten Datensätze einer initalen Datensatzes ein und fügt diese seinem Index hinzu.

beitung gelaufen sind, wird auf den Daten ein Blocking-Verfahren durchgeführt. Der **Indexer** ist ein Blocking Mechanismus, der zum einen mit dynamischen Daten umgehen können muss und zum anderen das Blocking anhand des DNF Blocking Schema BS durchführt. In Abbildung 4.5 wird die Build-Phase erläutert. Die Engine liest zunächst alle vorverarbeiteten Datensätze PD ein. Anschließend werden die Datensätze einzeln dem Indexer übergeben, welcher diese zu seinem Index hinzufügt. Dabei besteht die Möglichkeit, dass der Index während des Einfügens, anhand der gelernten Ähnlichkeitsfunktionen S, bestimmte Ähnlichkeiten vorausberechnet. Das Bauen des Index kann einige Zeit in Anspruch nehmen, weshalb der Index nach dem Bauen gespeichert wird. Im Falle eines Neustarts der Engine müssen dann nur die Datensätze eingefügt werden, welche während der Query-Phase hinzugekommen sind.

Query-Phase

In der Query-Phase (siehe Abbildung 4.6) erhält die Engine von einem Query-Parser eine Menge von Anfragedatensätzen. Nachdem diese vorverarbeitet wurden, wird jeder Datensatz einzeln dem Indexer übergeben. Dieser erzeugt für den übergebenen Datensatz eine Kandidatenmenge C möglicher Matches. Diese Kandidaten werden dem Klassifikator übergeben. Das Modell des **Klassifikators** wurde während der *Fit-Phase* von dem Fusion-Lerner trainiert und kann nun in der *Query-Phase* genutzt werden, um die Kandidaten in Matches und Non-Matches zu klassifizieren. Das Ergebnis der Klassifikation wird von der Engine zwischengespeichert, bis alle Datensätze verarbeitet wurden. Abschließend werden die gesammelten Ergebnisse R an den Benutzer übergeben.

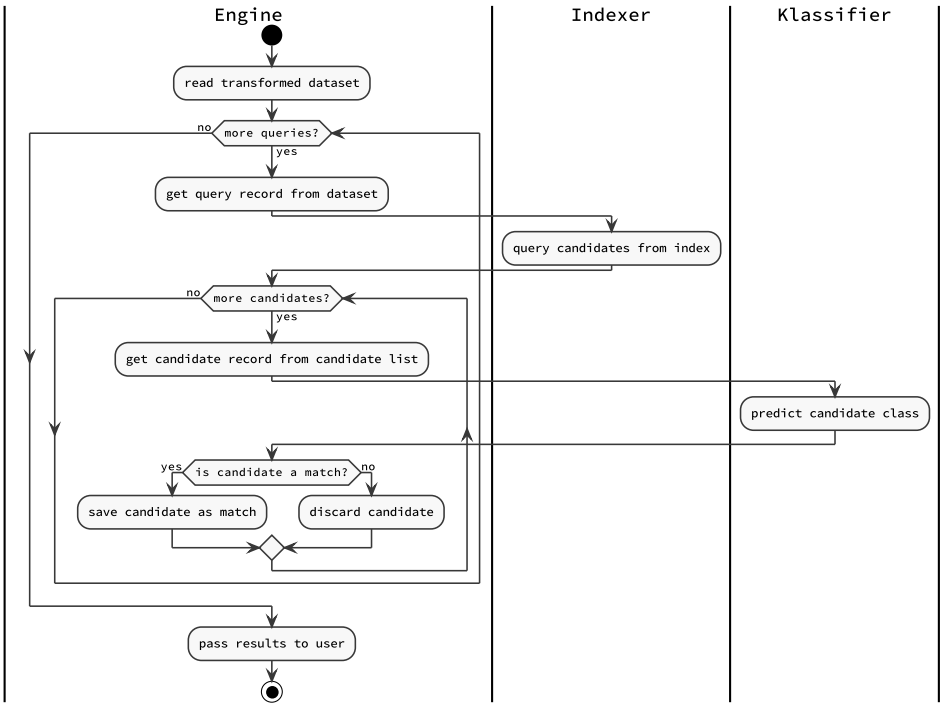

Abbildung 4.6 Aktivitätsdiagramm der Query-Phase. Zunächst werden der transformierte Datensatz vom Präprozessor gelesen. Danach werden Datensätze einzeln entnommen und dem Indexer übergeben. Dieser liefert eine Kandidatenliste. Jeder Kandidat wird vom Klassifikator in Match bzw. Non-Match klassifiziert. Matches werden von der Engine gespeichert und Non-Matches verworfen. Am Schluss wird das Ergebnis aller Anfragen dem Benutzer übergeben.

Auswertung

Für die Entwicklung von Komponenten besitzt die Engine die Möglichkeit Metriken zu messen und diese auszuwerten. Diese liefern ein wichtiges Indiz darüber, wie gut eine Komponente funktioniert. Des Weiteren ist es dadurch möglich das Zusammenspiel der Komponenten untereinander zu bewerten, indem beispielsweise Alternative Komponenten eingesetzt werden oder verschiedene freie Parameter gewählt werden. Von den Metriken, welche in Abschnitt 2.6 beschrieben wurden, kann die Engine für das Blocking die Pairs Completeness, Pairs Quality und Reduction Ratio aufzeichnen, sowie für den Klassifikator Recall, Precision, F-measure und Average Precision messen. Des Weiteren werden die Daten zum Zeichnen eines F-measure Graphen und einer Precision-Recall Kurve bereitgestellt. Darüber hinaus kann die Engine messen, wie lange einzelne Operationen einer Komponente benötigen. Beispielsweise wird gemessen, wie lange es dauert einen Datensatz in den Index einzufügen bzw. die Kandidatenliste zu einem Anfragedatensatz zu erhalten. Dadurch kann die Performanz, beispielsweise in Anfragen pro Sekunde auf einer Testhardware angegeben werden. Alle Metriken werden während der Query-Phase erhoben und können nach jeder Anfrage abgefragt werden.

4.2 Implementierung

In diesem Abschnitt wird zunächst die Programmierumgebung betrachet. Anschließend werden Optimierungen für die Implementierung der Engine und der Komponenten betrachtet, die eine Umsetzung der Algorithmen bei begrenzten Ressourcen, insbesondere Arbeitsspeicher und Rechenzeit, möglich machen.

4.2.1 Programmierumgebung

Als Programmiersprachen für die Implementierung wurden Python und C eingesetzt. Wobei C lediglich zur Implementierung der Ähnlichkeitsberechnung eingesetzt wurde, alle anderen Teile wurden mit Python umgesetzt. Python hat den Vorteil, dass es sehr einfach und schnell möglich ist, einen Prototypen eines Algorithmuses zu entwickeln und zu testen. Zudem gibt es eine Vielzahl von Qualitativ hochwertigen Paketen, die komfortable Standardfunktionalitäten bereitstellen, beispielsweise das Einlesen und das Schreiben von großen CSV-Dateien oder das Plotten von Graphen. Des Weiteren wird Python im Maschine Learning Bereich oft genutzt, was dazu führt, dass es eine Vielzahl von effizienten, ausgereiften und umfangreichen Frameworks gibt, um verschiedenste Lernaufgaben zu behandeln. Vor allem der Fusion-Lerner und der Klassifikator profitieren hiervon.

Der große Nachteil von Python ist das Global Interpreter Lock (GIL). Dieses verhindert, dass Python-Code in mehreren Threads gleichzeitig ausgeführt werden kann. Die Multithreading Bibliothek von Python ist daher lediglich geeignet, um Programme mit hoher E/A-Last zu beschleunigen, da Schreib- bzw. Lesezugriffe das GIL freigeben. Der Grund warum in Python ein GIL eingesetzt wird ist, dass dadurch die Single-Thread Ausführung optimiert wird. Multithreading, im Sinne von Gleichzeitigausführung, d.h. ein Prozess mit mehreren Threads, die auf verschiedenen Prozessorkernen, zur selben Zeit ausgeführt werden, wird dadurch allerdings komplett unterbunden. Um denoch Python zu paralelisieren gibt es zwei beliebte Möglichkeiten. Die erste Möglichkeit ist, statt Multithreading, Multiprocessing einzusetzen. Das hat allderdings den Nachteil, dass Daten zwischen Prozessen ausgetauscht werden müssen. Das lohnt sich jedoch nur für rechenintensive Aufgaben, wo der Overhead des Datenaustausches keine Rolle spielt. Die zweite Möglichkeit ist das Multithreading in einer anderen Programmiersprache umzusetzen, beispielsweise in C. Dies ist möglich, da das GIL lediglich die Mehrfachausführung von Python-Code verhindert. Allerdings erweist sich dies oft als relativ schwierig, da selbst einfache Datenklassen, beispielsweise `set` oder `dict`, keine Entsprechung in C haben und daher manuell, in beide Richtungen Python zu C und C zu Python, z.T. aufwendig konvertiert werden müssen.

Aufgrund der genannten Nachteile von Python wird die Engine und sämtliche Komponenten lediglich in einem Thread ausgeführt. Ideen dies zu optimieren konnten im Zeitrahmen der Thesis nicht umgesetzt werden. Dabei kann vor allen in der Fit-Phase durch Multi-Threading und Parallel Programming Laufzeit eingespart werden. Die längste Laufzeit haben dabei der DNF Blocks Lerner und der Fusion-Lerner.

4.2.2 Label Generator

Für den Label Generator wurden beide Ausprägungen (mit und ohne Ground Truth) umgesetzt. Zunächst wird die Variante ohne Ground Truth beschrieben. Anschließend die Variante mit Ground Truth, welche eine Modifikation der ersten Ausprägung ist.

Der Label Generator wurde gegenüber dem Algorithmus 4 von Kejriwal & Mirankern [17] und dessen Anpassung mit Ground Truth Matches in Algorithmus 5 in zwei Punkten modifiziert. Zunächst werden die Datensätze in den Blöcken alphabetisch sortiert. Damit ist es möglich ist deterministische Ergebnisse zu bekommen und daraufbasierend geeignete Testfälle zu schreiben. Des Weiteren werden, wie beim klassischen Sorted Neighborhood Verfahren, dadurch ähnliche Datensätze näher zusammengebracht, was die Wahrscheinlichkeit erhöht aussagekräftige Paare zu selektieren. Die zweite Anpassung ist sowohl eine Laufzeit-, als auch ein Arbeitsspeicheroptimierung. Ähnlich zum Record Identifier Index der Similarity-Aware Inverted-Index Verfahren, kann es durch das Blocking auf Basis der Token ebenfalls dazu kommen, dass riesige Blöcke erzeugt werden. Selbst wenn ähnliche Attribute durch Sortierung näher zueinander sortiert wurden, ist in diesen Blöcken ein großes Fenster nötig, um aussagekräftig Paare zu finden. Dies wiederum führt zu einer Explosion der Kandidatenmenge und damit des Arbeitsspeichers und der Laufzeit. Zur Optimierung wird ein Blockfilter eingeführt, sodass lediglich Kandidaten in Blöcken generiert werden, deren Anzahl an Datensätzen kleiner einer Schwelle z ist.

4.2.3 Blocking Schema Lerner

Die Algorithmen des DNF Blocks Lerners haben bei der Implementierung das Problem, dass nur eine bestimmte Menge an Arbeitsspeicher zur Verfügung steht. Der kritische Teil des Algorithmus ist für jeden Block die Erzeugung der Paarkombinationen. Angenommen die beiden Datensatzidentifikatoren eines Paares $(p1.id, p2.id)$ sind Integerwerte und der Datensatz hat nicht mehr als 2^30 Einträge, dann benötigt ein Integerwert 28 Bytes. Um möglichst effizient auf die Paare zuzugreifen, ist die Menge von Paarkombinationen als `set` implementiert. Damit ein `set` s ein Zugriffkomplexität von $O(1)$ ermöglichen kann, wird für jedes Element in der Menge ein Hashwert berechnet. Auf einem 64-bit System beträgt die Größe dieses Hashwertes h 8 Bytes. Somit benötigt ein

Eintrag $(h_j, p1_j.id, p2_j.id) \in s$ 64 Bytes. Bei Attributen mit wenigen möglichen Werten können Blöcke entstehen, die sehr viele Datensätze enthalten. Beispielsweise hat ein Block mit 10.000 Einträgen 49.995.000 Paare und benötigt 2.9 GB an Arbeitsspeicher. Somit kann bereits ein riesiger Block, den zur Verfügung stehenden Arbeitsspeicher sprengen, was dadurch zum Abbruch des Programmes führt. Aus diesem Grund wurde der Algorithmus dahingehend erweitert, dass die Erzeugung der Paare bei Ausdrücken, die zu viele Paare erzeugen würden, unterbunden wird und dies Ausdrücke mit dem niedrigsten Wert der Bewertungsskala bewertet werden.

Zur genaueren Analyse des Problems wird die Verteilung der Blöcke, anhand ihrer Größe (Anzahl von Datensätzen), betrachtet. Die Verteilungen in "Gute"-Blöcke, (benötigt weniger Arbeitsspeicher als zur Verfügung steht) und "Schlechte"-Blöcke (benötigt mehr Arbeitsspeicher als zur Verfügung steht), zu kategorisieren, wurde eine Schwelle t eingeführt. Anhand dieser Schwelle wird ein Block B bei $|B| < t$ als guter Block und bei $|B| > t$ als schlechter Block bewertet. Daraus kann für jede Verteilung berechnet werden, wie viel Prozent gute bzw. schlechte Blöcke es gibt. Dadurch ist es möglich bei Audrücken mit einer höheren schlechten Blockrate von b, beispielsweise $b = 0.1$, die Erzeugung der Blockpaare zu verhindern und die weitere Verarbeitung abzubrechen. Da aber bereits ein einziger schlechter Block, mit genügend Einträgen, den Arbeitsspeicher überfüllen kann, wird mit der Schwelle b lediglich eine Vorauswahl, besonders schlechter Ausdrücke, getroffen. Für den Fall, dass es nur wenige schlechte Blöcke gibt, bestehen deren Blockschlüssel meistens aus Stopwörtern, beispielsweise bei Strassennamen `Strasse`, `Weg`, oder `Platz`. Dieses Problem kann folglich durch eine bessere Vorverarbeitung der Daten gelöst werden. Dazu muss die Engine, die auf diese Weise gefundenen Stopwörter lernen und den Lernvorgang mit der erweiterten Vorverarbeitung der Daten wiederholen. Dieser Prozess sorgt allerdings dafür, dass das Lernen der Konfiguration deutlich länger dauert. Eine einfacherere Möglichkeit ist es, für jeden Ausdruck eine Liste mit verbotenen Blockschlüsseln anzulegen und die Blockschlüssel schlechter Blöcke dort hinzuzufügen. In der Build- und Query-Phase dürfen diese Blöckschlüssel vom Indexer demnach nicht genutzt werden.

Trotz dieser Optimierungen hat der DNF Blocks Generator immer noch hohe Arbeitsspeicheranforderungen, welche verhindern das Multithreading oder Multiprocessing auf einem Rechner zur Laufzeitoptimierung eingesetzt werden können. Denkbar ist aber die Verteilung auf ein Cluster von Rechnern, beispielsweise per Hadoop, wodurch deutlich mehr Prädikate in kürzerer Zeit überprüft werden können.

5 Evaluation

Diese Kapitel präsentiert die Ergebnisse der Evaluation. Zunächst wird dazu der experimentelle Aufbau erläutert. Im Anschluss werden die Komponenten des Systems bestimmt, für welche Alternativen zur Verfügung stehen. Danach findet die Festlegung der noch offenen freien Parameter statt. Für alle zu wählenden freien Parameter wird eine Abwägung zwischen Qualität und Effizienz durchgeführt. Als Nächstes wird der Einfluss der Ähnlichkeitsvektoren und Ähnlichkeitsfunktionen auf die Performanz des Systems geprüft. Anschließend wird das selbstkonfigurierende System gegen eine manuelle Baseline evaluiert und zum Schluss wird betrachtet welche Ergenisse mit einer vollständig synthetisierte Ground Truth Paare erzielt werden können.

5.1 Experimenteller Aufbau

Der experimentelle Aufbau beschreibt alle Komponenten und Schritte die benötigt werden, um die Evaluation durchzuführen und auszuwerten. Zunächst wird beschrieben, wie die Metriken berechnet wurden. Anschließend werden die Datensätze und deren Aufbereitung für die Evaluierung beschrieben. Danach wird beschrieben in welchen Schritten welche Metriken berechnet wurden und abschließend wird die genutzt Hardware bekannt gemacht.

5.1.1 Berechnung der Metriken für dynamische Entity Resolution

Während im statischen Entity Resolution die Metriken (vgl. Abschnitt 2.6) am Ende des Verfahrens einmalig berechnet werden können, ist dies im dynamischen Falle nicht möglich, da ein Datenstrom kein definiertes Ende hat. Das bedeutet, die Metriken müssen inkrementell mit jeder Anfrage q erhoben werden. Für Anfragen ohne tatsächliche Matches wird lediglich das Reduction Ratio erhoben, da alle anderen Metriken keine Aussagekraft haben, beispielsweise wäre der Recall undefiniert und die Precision immer 0. Zur Berechnung der Effizienzmaße Pairs Completeness, Pairs Quality und Reduction Ratio werden die tatsächlichen Matches n_M, die tatsächlichen Non-Matches n_N, die Matches in der Kandidatenmenge s_M und Non-Matches in der Kandidatenmenge s_N benötigt. Die Kandidatenmenge wird mit C bezeichnet, die tatsächlichen Matches mit P und die Menge der Datensätze des Indexers mit IX. s_M ist die Anzahl der Matches zur Anfrage q in C, s_N ist die Anzahl der Non-Matches zu q in C, n_M ist die

Gesamtanzahl der Matches zu q in den Matches P und n_N ist die Gesamtanzahl an Non-Matches zu q in IX. Für n_N muss der Anfragedatensatz von der Gesamtmenge abgezogen werden, da dieser zu Begin jeder Anfrage vom Indexer in den Datenbestand aufgenommen wird, bzw. wenn er dort schon vorhanden ist keine Rolle für die Enity Resolution spielt, da er herausgefiltert wird. Mit jeder Anfrage werden s_M, s_N und n_M mit den vorherigen Werten aufsummiert, sodass die Effizienzmaße Bezug auf alle bisher gestellten Anfragen nehmen. Die Anzahl n_N nimmt Bezug auf eine wachsende Menge, sodass beim Aufsummieren die frühen Anfragen abgewertet werden. n_M wird zur Berechnung des Reduction Ratio benötigt, sodass dieses für jede Anfrage berechnet, gespeichert und auf Abruf gemittelt wird.

Die Qualitätsmaße Recall, Precision, F-measure und Average Precision werden über die True Positives (TP), False Positives (FP) und False Negatives (FN) bestimmt. Deren Berechnung ist identisch zu den Werten der Effizienzmaße mit der Abweichung, dass diese auf der Ergebnismenge R gemessen werden. Die True Negatives werden nicht berechnet, da diese in den Metriken nicht benötigt werden. Auch hier werden die Werte für jede Anfrage summiert, sodass die aus der Summe berechneten Metriken alle bisherigen Anfragen berücksichtigen.

5.1.2 Ähnlichkeitsmaße

Der Similarity Lerner wählt aus einer Menge von Ähnlichkeitsfunktionen die besten aus. Für die Evaluierung wurden dem Similarity Lerner die folgenden 7 Metriken übergeben, wobei die letzen vier aus Abschnitt 2.4 bekannt sind:

- Bag-Distanz [37]
- Compression-Distanz [38]
- Hamming-Distanz [39]
- Jaro-Distanz
- Jaro-Winkler-Distanz
- Jaccard-Koeffizent
- Levenshtein-Distanz

Dazu wird die Implementierung der Algorithmen aus der *Harry* Bibliothek [40] von Rieck & Wressnegger genutzt. Dabei handelt es sich um eine Open-Source C-Bibliothek. Bei der Verwendung dieser Bibliothek ist die Problematik aufgetreten, dass die Variablen ausschließlich auf dem Stack gehalten werden. Da eine Ähnlichkeitsfunktion vor Benutzung konfiguriert werden muss, ist es in einem Prozess folglich nur möglich ein Ähnlichkeitsmaß gleichzeitig zu verwenden und häufiges Wechseln der Funktion ist dadurch mit hohem Mehraufwand verbunden. Deshalb wurden im Rahmen dieser Arbeit die Berechnung und Normalisierung der Ähnlichkeiten umgeschrieben, sodass alle Va-

Datensatz	Einträge	Duplikatspaare	Attribute
Abt-Buy	2.171	1.096	4
Amazon-GoogleProducts	4.587	1.299	4
Cora	1.879	64.577	5
DBLP-ACM	4.908	2.223	4
DBLP-Scholar	66.877	5.346	4
NCVoter	8.261.839	155.470	17
Restaurant	864	112	4

Tabelle 5.1 Überblick der verwendeten Datensätze

riablen auf den Heap allokiert werden. Als Folge dessen ist es nach einmaliger Konfiguration möglich, beliebig viele Ähnlichkeitsfunktionen gleichzeitig zu nutzen. Zudem wurde eine Pythonschnittstelle entwickelt, damit diese Funktionen durch den Similarity Lerner aufgerufen werden können. Die modifizierte Bibliothek ist unter dem Namen *libsimilarity*[1] auf Github, als Open-Source Software, verfügbar. Für den Similarity Lerner wurden alle genutzten Ähnlichkeitsmaße derart konfiguriert, dass beim Normalisieren der Distanzen, auf den Wertebereich zwischen 0 und 1, die Dreieckungleichung eingehalten wird.

5.1.3 Datensätze

Im Folgenden werden die Datensätze beschrieben, die in der Evaluation genutzt werden. Tabelle 5.1 bietet einen Überblick über alle Datensätze. Dabei ist der NCVoter Datensatz mit Abstand der größte, weshalb diesem besondere Aufmerksamkeit in der Evalution geschenkt wird. Die Duplikatspaare beinhalten sowohl Matches zwischen zwei Datensätzen, als auch Matches zwischen Cliquen (mehr als 2 Datensätze), wobei die Cliquen vollständig als Paare aufgelöst sind. Beispielsweise gibt es für eine Clique (1,2,3) die Paare (1,2), (1,3) und (2,3). Die Duplikatspaare wurden im Falle von des Cora und des Restaurant Datensatzes von Hand identifiziert und für die anderen Datensätze semi-automatisch über verschiedene Verfahren bestimmt.

CORA[2]

Der CORA Datensatz beinhaltet 1879 bibliographische Einträge über wissenschaftliche Veröffentlichungen aus dem Maschine Learning Bereich. Die Einträge bestehen aus Autor, Jahr, Titel, Technologie, Institution, Buchtitel, Bearbeiter, Verlag, Seiten, Adresse, Monat, Notiz, Journal, Ausgabe Typ und Datum. Insgesamt beinhaltet dieser Datensatz 64.577 Duplikate. Dieser Datensatz ist besonders schwierig zu Deduplizieren, da teilwei-

[1] `https://github.com/sappo/libsimilarity`
[2] https://hpi.de/naumann/projects/repeatability/datasets/cora-dataset.html

se nur Initialen der Autoren vorhanden sind bzw. Attribute zusammengefügt oder getauscht wurden.

Abt-Buy & Amazon-GoogleProducts[3]

Diese beiden Datensätze beinhalten Produkte aus dem Onlinehandel verschiedener Plattformen mit Name, Beschreibung, Hersteller und Preis. Der Abt-Buy Datensatz beinhaltet 2171 Einträge mit 1096 Duplikaten. Im Amazon-GoogleProducts Datensatz sind es 4587 Einträge mit 1299 Duplikaten.

DBLP-ACM & DBLP-Scholar[4]

Diese beiden Datensätze beinhalten bibliografische Einträge mit Titel, Autor(en), Konferenz, und Jahr. Der DBLP-ACM Datensatz beinhaltet 4908 Einträge und 2223 Duplikate. Im DBLP-Scholar Datensatz sind sind 66877 Einträge mit 5346 Duplikaten. Dabei ist zu beachten, dass der DBLP-ACM Datensatz einfach zu klassifizieren ist, da ein Großteil der Daten durch eine Instanz gepflegt werden.

Restaurant[5]

Der Restaurant Datensatz ist ein kleiner mit lediglich 864 Einträgen, welche aus Restaurantname, Adresse, Telefonnummer und der Küchenart bestehen. Es gibt insgesamt 112 Restaurantduplikate, welche doppelt vorkommen.

NCVoter

Der NC Voter Registration (NCVoter) Datensatz beinhaltet ca. 8 mio Datensätze aus dem Wählerverzeichnis des Bundestates North Carolina in den USA. Eine genaue Analyse des Datensatzes wurde von Christen [41] durchgeführt. Der Datensatz beinhaltet ca. 145.000 Duplikate zwischen zwei Einträgen, sowie 3.500 zwischen drei und mehr Einträgen. Die Zuordnung der Duplikate wurde dabei über die Wählerregistriernummer getätigt. Weitere Attribute sind Namenspräfix, Vorname, Zweiter, Vorname, Nachname, Namenssuffix, Alter, Geschlecht, Rassenziffer, Ethnizitätsziffer, Strasse + Hausnummer, Stadt, Bundesland, Postleitzahl, Telefonnummer, Geburtsort und Registrierdatum.

[3] http://dbs.uni-leipzig.de/en/research/projects/object_matching/fever/benchmark_datasets_for_entity_resolution
[4] http://dbs.uni-leipzig.de/en/research/projects/object_matching/fever/benchmark_datasets_for_entity_resolution
[5] http://hpi.de/naumann/projects/data-quality-and-cleansing/dude-duplicate-detection.html

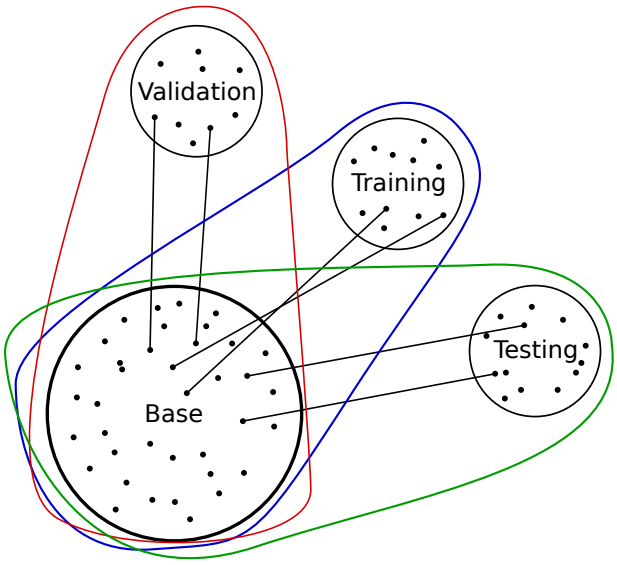

Abbildung 5.1 Aufteilung der Datensätze in Validierungsmenge, Trainingsmenge und Testmenge. Tupel in den Mengen sind durch Punkte markiert und Duplikate durch eine Line zwischen zwei Tupeln. Die farbigen Linen zeigen, wie die jeweilige Untermenge gebildet wird.

Febrl

Die Febrl-Datensätze wurden synthetisch durch den Febrlgenerator [42] erzeugt. Die Attributsdaten dafür liefert ein australisches Telefonbuch. Die generierten Einträge haben folgende Attribute: Kultur, Geschlecht, Alter, Geburtsdatum, Titel, Vorname, Nachname, Bundesland, Vorort, Postleitzahl, Hausnummer, Straße und Telefonnummer.

- Febrl-4k-1k: 5.000 Einträge mit 1.000 Duplikaten zwischen zwei Datensätzen
- Febrl-9k-1k: 10.000 Einträge mit 1.000 Duplikaten zwischen zwei Datensätzen
- Febrl-90k-10k: 100.000 Einträge mit 10.000 Duplikaten zwischen zwei Datensätzen

5.1.4 Aufbereitung der Datensätze

Für die Durchführung der Evaluierung wurde der NCVoter Datensatz aus Abschnitt 5.1.3 in jeweils vier disjunkte Teile gesplittet. Diese Aufteilung ist in Abbildung 5.1 dargestellt. Die Hälfte der Datensätze eines Datensatzes befindet sich in der Base und die andere Hälfte ist zu gleichen Teilen in Validierung, Training und Testing aufgeteilt. Datensätze in den Mengen sind durch schwarze Punkte markiert. Matches sind durch Linien verbunden. In der Build-Phase wird der initiale Index stets aus den Datensätzen der Base gebaut. Der Anfragestrom, in der Query-Phase, wird durch Datensätze aus Validierung, Training oder Testing zusammengestellt. Durch die Verteilung der Matches ist sichergestellt, dass dadurch für jedes Match eine Query durchgeführt wird, in welcher das

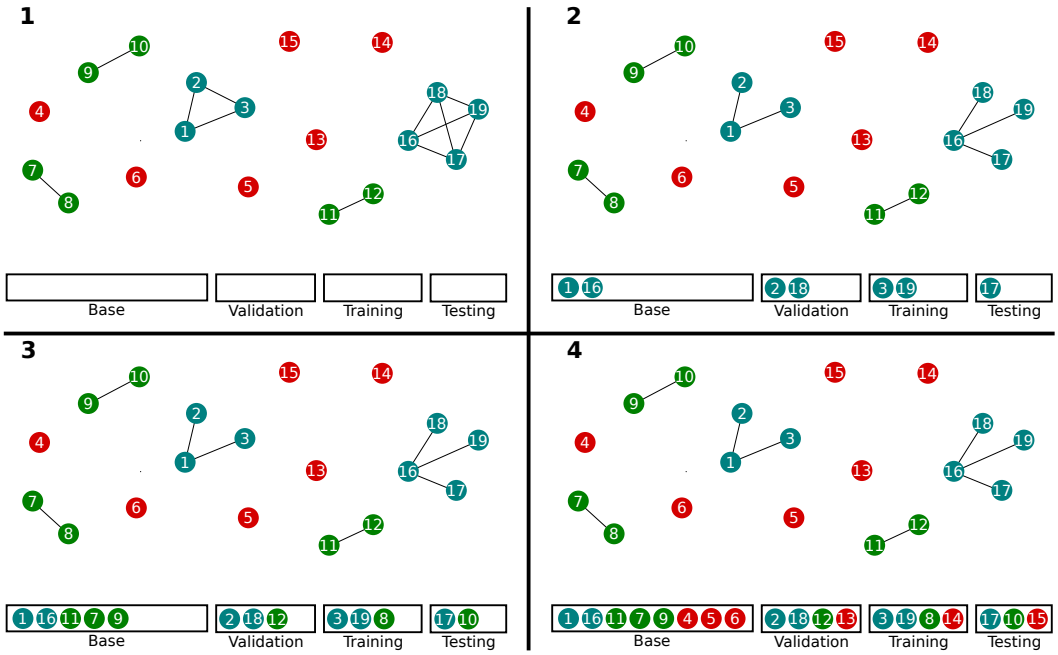

Abbildung 5.2 Vorgehen zur Aufteilung eines Datensatzes in vier Teilmengen. Datensätze werden in drei Kategorien zugeordnet: Non-Matches (rot), Matches (grün), Matchcliquen (blaugrün). Diese werden seperat in Teil 2, 3 und 4 aufgeteilt.

jeweilige andere in der Base gefunden werden kann. In der Fit-Phase werden die Duplikate zusammen benötigt, weshalb jeweils Validierung, Training und Testing mit der Base zusammengefasst werden, wie durch die rote, grüne bzw. blaue Umrandung dargestellt ist.

Das Vorgehen der Aufteilung eines Datensatzes ist in Abbildung 5.2 dargestellt. Dazu werden die Datensätze in drei Kategorien eingeteilt (Teil 1): Non-Matches (rot), Matches zwischen zwei Datensätzen (grün) und Cliquen von Matches (blaugrün). Zuerst werden die Cliquen auf die Mengen verteilt (Teil 2). Dafür wird jeweils ein Datensatz bestimmt, welcher der Base zugewiesen wird, hier `1` und `16`. Anschließend werden die restlichen Datensätze der Cliquen per Round-Robin auf Validation, Training und Testing verteilt. Für die erste Clique bedeutet das `1` in die Base, `2` in Validation und `3` in Training. Bei der zweiten Clique kommt `16` in die Base, `17` in Testing, da das Round-Robin der vorherigen Clique vorgesetzt wird, `18` in Validation und `19` in Training. Danach werden die übrigen Paare ebenfalls über Round-Robin aufgeteilt (Teil 3). Jeweils ein Datensatz der Paare wird der Base zugewiesen (`7`, `9`, `11`) und der andere wird auf Validation, Training und Testing verteilt, wobei der Round-Robin Mechanismus unabhängig von dem der Cliquen ist. Zum Schluss werden die Non-Matches aufgeteilt (Teil 4). Dazu werden jeweils drei Datensätze der Base zugewiesen und anschließend drei per Round-Robin auf Validation, Training und Testing verteilt. Dieser Round-Robin Mechanismus ist ebenfalls unabhängig von den beiden anderen. Durch die drei unabhängigen Round-Robin Aufteilungen, kann es dazu kommen, dass der Testing Datensatz bis zu drei Datensätze

weniger hat als Validation oder Testing. Bei tausenden bzw. Millionen von Datensätzen ist dies jedoch nicht ausschlaggebend.

Die übrigen Datensätze wurden nach demselben Verfahren in lediglich zwei disjunkte Mengen geteilt, sodass weiterhin 50 % der Datensätze in der Base sind und die anderen 50 % in der Anfragemenge.

5.1.5 Durchführung

Die Durchfürung der Evaluation erfolgt pro Durchlauf in drei Schritten. Im ersten Schritt konfiguriert sich das System selbst und die Konfiguration wird abgespeichert. Anschließend wird die Build- und Query-Phase zum ersten Mal durchgeführt, um die Metriken für Qualtität und Effizienz zu bestimmen. Im letzten Schritt wird die Build- und Query-Phase ein zweites Mal durchgeführt, um Laufzeiten zu messen, die durch die Erhebung der Metriken in Schritt 2 verfälscht wurden. Insbesondere wird die Zeit zum Einfügen in den Index und zum Anfragen der Duplikate für jeden Datensatz gemessen. Dabei wird für die Abschnitte 5.2-5.7 lediglich der NCVoterDatensatz benutzt. Des Weiteren werden für die Abschnitte 5.2-5.5 die Validierungsdaten des NCVoter Datensatzes genutzt. Die übrigen Datensätze werden in Abschnitt 5.8 evaluiert.

Die Evaluation wurde auf 19 komponentengleichen Linuxrechnern, mit einer Intel(R) Core(TM) i7-6700 CPU mit 3.40GHz und 8 Prozessorkernen, sowie 32 GB Arbeitsspeicher, durchgeführt. Diese Rechner sind Bestandteil eines Rechnerpools an der Hochschule RheinMain, der für die Nutzung von Studierenden bereit steht. Während der Tests wurde die Hardware nicht von anderen Studierenden genutzt.

5.2 Auswahl der Komponenten

Die Komponenten des selbstkonfigurierenden Systems wurden in der Analyse in Kapitel 3 und im Design in Kapitel 4 vorgestellt. Dabei ist der Label Generator (Abschnitt 3.1), der Blocking Schema Lerner (Abschnitt 3.2), und der Similarity Lerner (Abschnitt 3.5) fix. Alternativen gibt es für jeweils für Parser, Präprozessor, Fusion-Lerner, Klassifikator und Indexer und werden hier untersucht.

Die Datensätze aus Abschnitt 5.1.3 liegen alle im CSV-Format vor, weshalb auch die gesplitteten Datensätze ebenfalls ins CSV-Format geschrieben wurden, um alle Datensätze einheitlich von einem CSV-Parser zu verarbeiten. Im Rahmen dieser Arbeit konnte der CSV-Parser nicht dahingehend anpasst werden, zuverlässig die Attributstypen zu bestimmen. Aus diesem Grund werden alle Attribute als Strings behandelt, was zudem anderem die Wahl geeigneter Blockingprädikate für den Blocking Schema Lerner vereinfacht.

Weiterhin sind alle Datensätze aus Abschnitt 5.1.3 in englischer Sprache, weshalb der Präprozessor bekannt englische Stopwörter herausfiltert und anschließend alle Attribute in kleinschreibweise konvertiert. Weitere Attributs- bzw. Datensatzspezifische Anpassungen werden nicht durchgeführt.

Um die Hyperparameter der Klassifikatoren zu erlernen muss der Fusion-Lerner insbesondere Wissen, wie deren API Schnittstelle ist, damit er die Modelle trainieren und auswerten kann. Aufgrunddessen und weil die Implementierung von verschiedenen Lernverfahren und Klassifikatoren nicht Schwerpunkt der Thesis ist, wurde für die Umsetzung die Python Maschine Learning Bibliothek Scikit-learn [43] eingesetzt. Diese bietet ein breites Spektrum an Funktionen:

- Klassifikation, bestimmen zu welcher Klasse ein Objekt gehört.
- Regression, einen fortlaufenden Wert eines Objektes vorhersagen.
- Clustering, automatisches gruppieren von Objekten.
- Dimensionsreduktion, reduzieren der Anzahl zu betrachtender zufälliger Variablen.
- Modellauswahl, vergleichen, validieren und auswählen von Parametern und Modellen.
- Vorverarbeitung, Eingabetransformation und Normalisierung.
- Evaluation, berechnen der Effizienz und Qualität von Modellen.

Für den Fusion-Lerner sind dabei das Module zur Modellauswahl, zur Evaluation und zur Klassifikation interessant. Dieser ist zudem die einzige Komponente, die ihre Aufgabe parallelisieren kann, da dies in Scikit-learn transparent implementiert ist. Zur Auswahl stehen eine Grid Search und eine zufällige Suche mit begrenzter Tiefe. Tests mit dem größten Datensatz (NCVoter) haben ergeben, dass die Zeit, die eine Grid Search benötigt (> 1 Stunde), für die Evaluation vertretbar ist. Deshalb wird die Klasse `GridSearchCV` verwendet. Für die Kreuzvalidierung wird der Stratified K-Fold (implementiert in `StratifiedKFold`) verwendet, da dieser das Verhältnis der Ground Truth beibehält.

Die als Klassifikator nutzbaren Komponenten müssen zur Grid Search kompatibel sein. Das Scikit-learn Klassifikationsmodul beinhaltet dazu SVMs, DecisionTrees, neuronale Netze und mehr. Diese Implementierungen können ohne Anpassungen mit der Scikit-learn `GridSearchCV` verwendet werden. Die vorgestellten Klassifikatoren in Abschnitt 2.5.2 verwenden hauptsächlich Decision Trees und Support Vector Machines. Deshalb werden diese beiden für die Evaluation eingesetzt. Die entsprechenden Scikit-learn Klassen sind `DecisionTreeClassifier`, `SVC` für SVM mit RBF-Kernel und `LinearSVC` für SVM mit Linearkernel.

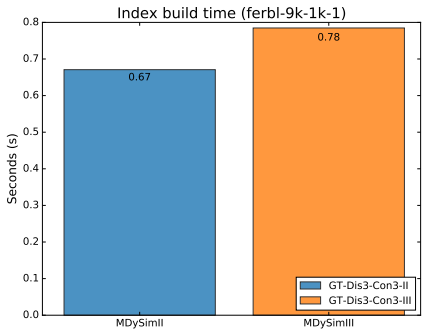

Abbildung 5.3 MDySimII vs MDySimIII - Bauzeit

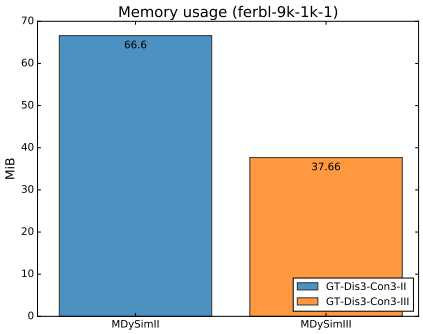

Abbildung 5.4 MDySimII vs MDySimIII - Speicherverbrauch

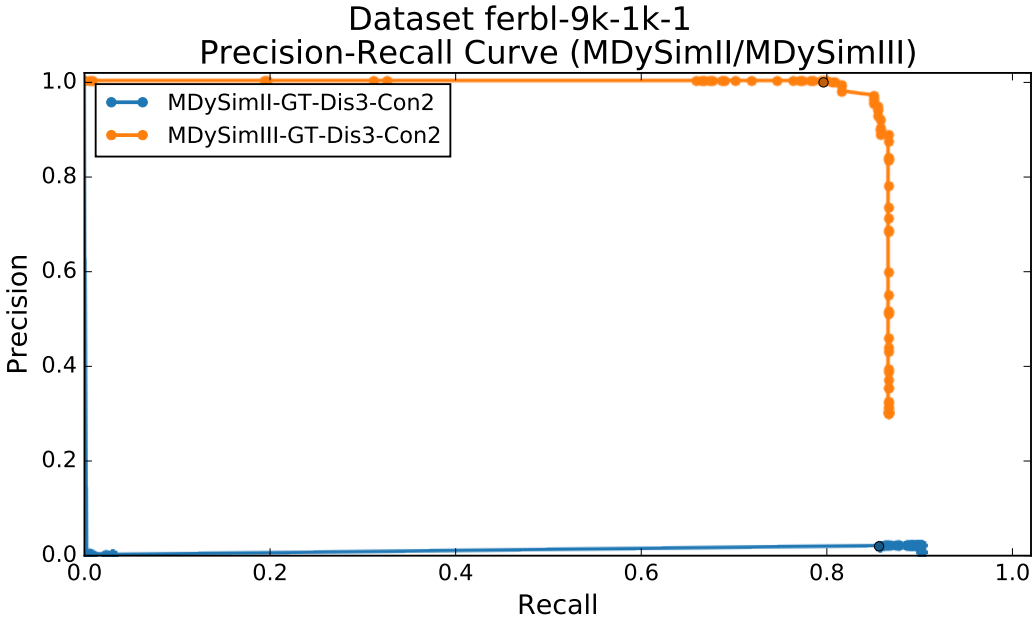

Abbildung 5.5 MDySimII vs MDySimIII - Precision-Recall Kurve

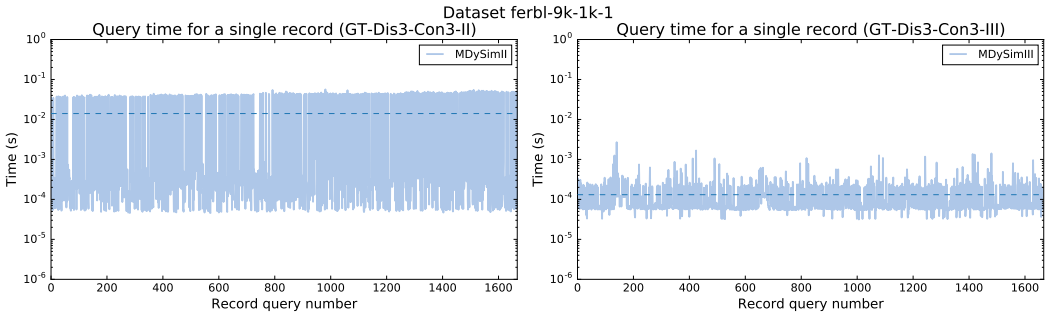

Abbildung 5.6 MDySimII vs MDySimIII - Anfragezeiten

Der Indexer kann durch den MDySimII oder den MDySimIII aus Abschnitt 3.4 besetzt werden. Beide Indexer wurden auf dem ferbl-9k-1k-1 Datensatz mit Ground Truth gegeneinander getestet, weil das MDySimII Verfahren auf dem NCVoter-Datensatz nicht in angemessener Zeit durchgeführt werden konnte. Für den MDySimII wurde ein Blocking Schema mit einem zweistelligen Ausdruck gelernt und für den MDySimIII wurde ein Blocking Schema mit einem zweistelligen und einem einstellingen Ausdruck. In Abbildung 5.3 sind die Bauzeiten der beiden Indexer verglichen. Aufgrund der komplexeren Struktur und des längeren Blocking Schemas schneidet der MDySimIII hier erwartungsgemäß leicht schlechter ab. Abbildung 5.4 zeigt den Speicherbedarf beider Indexer. Überraschend ist, dass der MDySimII fast das doppelte an Speicher benötigt, obwohl dieser die Vorteile in der Struktur und dem einfacheren Blocking Schema hat. Ein Hinweis dafür kann in der Precision-Recall Kurve in Abbildung 5.5 abgelesen werden. Zwar erreicht der MDySimII einen Recall von über 80 %, doch die Precision ist nahezu 0 %. Im Gegensatz dazu errreicht der MDySimIII knapp 60 % Recall, dafür ist die Precision nahe 100 %. Daraus folgt, dass der MDySimII im Durchschnitt deutlich größere Blöck erzeugt als der MDySimIII. Dementsprechend wächst auch der Similarity Index, da dieser die Ähnlichkeiten aller Attribute eines Blockes untereinander verknüpft. Die schlechte Precision des MDySimII macht sich direkt in den Anfragezeiten in Abbildung 5.6 bemerkbar. Die durchschnittliche Anfragezeit für den MDySimII liegt bei 10^{-2} und ist damit um 10^{-2} dramatisch schlechter als die des MDySimIII mit 10^{-4}. Der Datensatz ist mit 10.000 Einträgen relativ klein, dennoch macht sich bereits hier ein deutlicher Unterschied bemerkbar, sowohl in der Qualität als auch der Effizienz. Aufgrund der schlechten Precision skaliert der MDySimII nicht und erfüllt daher bei großen Datensätzen nicht die Anforderungen an die niedrigen Latenzen. Deswegen wird in der weiteren Evaluation der MDySimIII als Indexer genutzt.

5.3 Auswahl der Freien Parameter

Auf der Validierungsmenge wurden robuste, freie Parameter für die Evaluierung gewählt. Robust bedeutet, dass diese nicht optimal für jeden Datensatz sind, sondern gute Ergebnisse für alle Datensätze liefern und gleichzeitig verhindern, dass die Entity Resolution katastrophal versagt. Um die freien Parameter zu bestimmen, wurden diese anhand des NCVoter Datensatzes ausprobiert und ausgewertet. Für diesen Datensatz wurden dazu die folgenden acht Attribute genutzt: Geburtsdatum, Vorname, Nachname, Bundesstaat, Ort, PLZ, Straße und Telefonnummer. Zunächst werden die Parameter des Blocking Schema Lerners bestimmt, da das Blocking Schema zur Bewertung des Label Generators und des Fusion-Lerners benötigt wird.

5.3.1 Blocking Schema Lerner

Für den Blocking Schema Lerner aus Abschnitt 3.2 müssen folgende freie Parameter bestimmt werden:

- Blockschlüsselgenerator
- Blockingprädikate
- Maximale Konjunktion und Disjunktion
- Blockfilter (Größe/Ratio)

Blockschlüsselgenerator

Von den drei unterschiedlichen Möglichkeiten zur Erzeugung zusammengesetzer Blockschlüssel aus Abschnitt 3.3 wurde im Rahmen dieser Thesis nur der vorgestellte Algorithmus 8 implementiert. Weshalb auch dieser für die Evaluation genutzt wird.

Geeignete Prädikate

Die Blockingprädikate sind Hauptbestandteil eines Blocking Schema und haben deswegen den größten Einfluss auf die Qualität und Effizienz des Gesamtsystems. Um geeignete Prädikate auszuwählen, wurden folgende 6 Prädikate betrachtet.

- Identität eines Attributes (ID)
- Token eines Attributes, welche durch Leerzeichen getrennt sind (Tok)
- Prefixe eines Attributes der Längen 2-4 (Pre)
- Suffixe eines Attributes der Längen 2-4 (Suf)
- Bigram eines Attributes sind n-Gramme der Länge 2 (Bi)
- Trigram eines Attributes sind n-Gramme der Länge 3 (Tri)

Aus Performanzgründen wurden immer nur zwei Prädikate zusammen getestet. Jeder Durchlauf wurde folglich mit 16 spezifischen Blockingprädikaten durchgeführt, eines pro Attribut pro Prädikat. Da die Blockschlüsselerzeugung über ID offensichtlich am effizientesten ist, wurden alle Kombinationen der anderen Prädikate zusammen mit ID getestet. Zusätzlich wurden (Prefix, Sufix) und (Bigram, Trigram) getestet. Da einige dieser Kombinationen einen deutlichen Einfluss auf die Effizienz haben, wurde aus Zeitgründen der Febrl-4k-1k Datensatz genutzt.

In Abbildung 5.7 ist der Arbeitsspeicherverbrauch des gebauten Indexes dargestellt. Alle generierten Blocking Schemata bestehen aus zwei zweistelligen Ausdrücken. Das speicherhungriste Blocking Schema wurde durch (ID, Trigram) erzeugt und besteht aus-

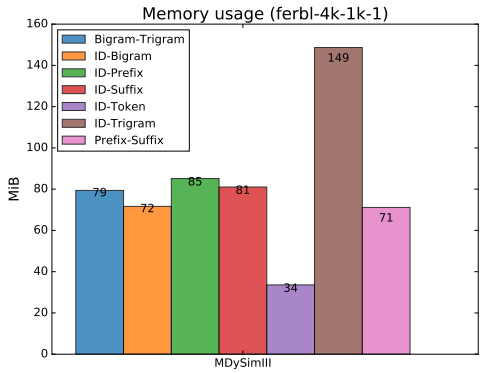

Abbildung 5.7 Maximaler Arbeitsspeicherverbrauch der unterschiedlichen Prädikatspaare in der Query-Phase.

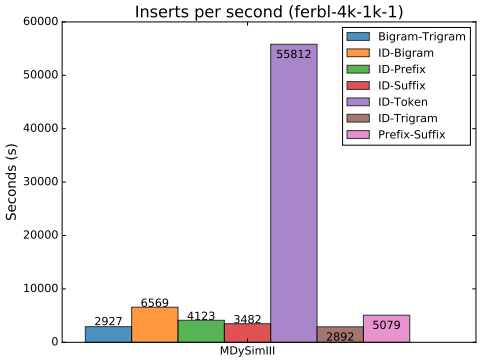

Abbildung 5.8 Einfügeoperation pro Sekunde für die unterschiedlichen Prädikatspaare in der Build-Phase.

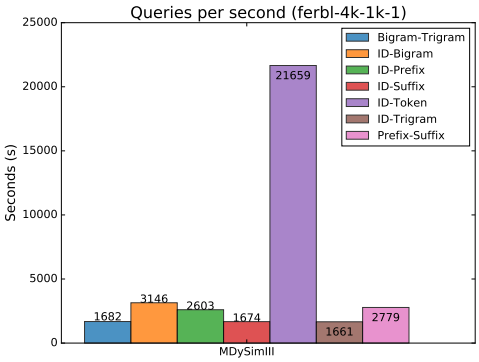

Abbildung 5.9 Anfragen pro Sekunde für die unterschiedlichen Prädikatspaare in der Query-Phase.

schließlich aus spezifischen Blockingprädikaten der Trigramme, alle anderen Blocking Schemata nutzen beide Prädikate. Das speichersparenste Schema wird durch (ID, Token) erzeugt. In den Abbildungen 5.8, 5.9 sind die Einfügeoperationen und Anfragen pro Sekunde dargestellt. Dabei zeigt sich, dass (ID, Token) mit deutlichem Abstand in beiden Abbildungen dominiert. Dies ist zum einen auf die geringerer Anzahl der Blockschlüssel zurückzuführen, die pro Attribute erzeugt werden und zum anderen auf die deutlich bessere Pairs Quality (PQ), welche in Tabelle 5.2 zu sehen ist. Hierbei erreicht (ID, Token) einen Wert von 0.96, das nächstbeste Paar (Bi-Tri) kommt lediglich auf 0.33. Die hohe Pairs Quality kommt allerdings auf Kosten der Pairs Completeness (PC), welche von allen Kombinationen mit 0.83 die schlechteste ist. Andere Paare erreichen hier bis zu 0.99, beispielsweise für (Prefix, Suffix). Die Prädikate ID und Token sind zwar qualitativ leicht unterlegen, zeigen jedoch bei der Effizienz eine deutliche Überlegenheit. Alle anderen Prädikate überzeugen zwar qualitätiv, skalieren jedoch nicht und können die Anforderungen an die niedrigen Latenzen dadurch nicht erfüllen. Für die weitere Evaluation werden deshalb nur noch die Prädikate ID und Token benutzt.

	Bi-Tri	ID-Bi	ID-Pre	ID-Suf	ID-Tok	ID-Tri	Pre-Suf
PC	0.97	0.90	0.99	0.97	0.83	0.95	0.99
PQ	0.33	0.09	0.33	0.09	0.96	0.28	0.18

Tabelle 5.2 Pairs Completeness und Pairs Quality der Prädikatkombinationen

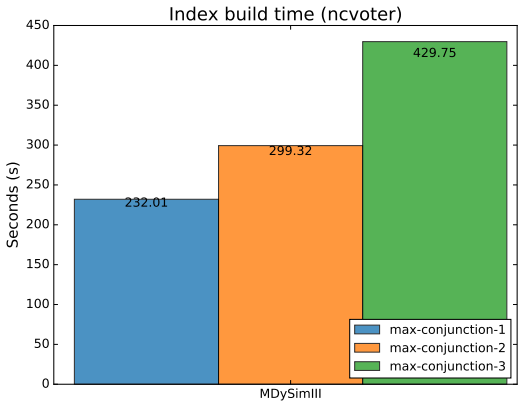

Abbildung 5.10 Bauzeiten des Indexers bei unterschiedlicher maximaler Länge der Konjunktion der Ausdrücke des Blocking Schema.

Maximale Konjunktion/Disjunktion

Die maximale Konjunktion max_k von spezifischen Blockingprädikaten von Ausdrücken und die maximale Disjunktion max_d von Ausdrücken können entscheidend sein, um ein gutes Blocking Schema zu bilden. Ein konjunktiver Ausdruck wird bewertet, indem die Blöcke durch Indexer gebaut werden, was relativ zeitintensiv ist. Auf der anderen Seite werden für die Disjunktion der Ausdrücke nur boolsche Vektoren verodert und miteinander verglichen, was deutlich effizienter ist. Im Allgemeinen gilt, je weniger Konjunktionen erlaubt sind, desto höher ist die Wahrscheinlichkeit, dass ein Ausdruck Non-Matches nicht korrekt ausschließen kann und je weniger Disjunktionen, desto höher ist die Wahrscheinlichkeit, dass nicht alle Matches erfasst werden. In [17] geben Kejriwal & Miranker an, in der Evaluation ihres DNF Blocking Schema Lerners keine Verbesserung für $max_k > 2$ gemessen zu haben. Im Gegensatz dazu wird max_d von ihnen nicht beschränkt, zudem wird über die maximal gemessenen Disjunktion keinerlei Aussage gemacht. Für die Evaluation des Algorithmus aus Abschnitt 3.2 wurde max_d für alle Durchläufe auf 5 gesetzt, da das Verfahren hierfür effizient genug ist und es keinen anderen Grund gibt diese zu restriktieren. Für max_k wurden die Werte 1, 2 und 3 getestet. Bei den ausgwählten 8 Attributen des NCVoter Datensatzes mit den zwei spezifischen Blockingprädikaten beträgt die Anzahl der gebildeten Ausdrücke für $max_k = 1$ gleich 16, für $max_k = 2$ gleich 136, das sind 8.5 Mal mehr Ausdrücke und für $max_k = 3$ gleich 696 Ausdrücke, was nochmals 5 Mal soviele sind. Dabei ist zu beachten, dass max_k jeweils die Ausdrücke von $max_k - 1$ beinhaltet. Die Anzahl der zu prüfenden Ausdrücke,

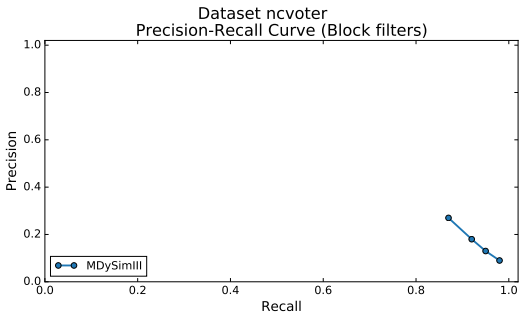

Abbildung 5.11 Precision-Recall Kurve des zweiten guten Blocking Schema bei sinkendem t.

hat deutlichen Einfluss auf die gesamte Lernzeit. Bei $max_k = 1$ dauert das Lernen nur 22 Minuten, bei $max_k = 2$ dauert es 5.5 Mal solange mit 2 Stunden und bei $max_k = 3$ dauert es nochmal 12.5 Mal solange, mit einem vollen Tag und einer Stunde. Während von $max_k = 2$ auf $max_k = 3$ die Anzahl der Ausdrücke um das fünffache wächst, ist dies bei der Lernzeit über das 12fache. Dies lässt sich mit den Bauzeiten des Indexers erklären. In Abbildung 5.10 sind die Bauzeiten des jeweils besten Blocking Schema im jeweiligen Durchlauf dargestellt. Während der Bauzeit wurden ca. 4 Mio. Datensätze eingefügt. Dabei besteht das beste Blocking Schema in allen drei Fällen stets aus Ausdrücken der Länge max_k. Die Blocking Schemata für $max_k = 1$ und $max_k = 2$ haben jeweils zwei Ausdrücke und das für $max_k = 3$ besteht aus drei Ausdrücken. Die Bauzeiten verraten, dass je länger die Ausdrücke werden, desto länger benötigt der Indexer zum Erzeugen der Blockschlüssel, wodurch sich die Bauzeit verlängert. Da die Blockschlüssel auch für jede Anfrage erzeugt werden, verlängern sich diese ebenfalls. Während bei $max_k = 1$ noch 50k Anfragen/s beantwortet werden, sind es bei $max_k = 2$ nur noch 22k Anfragen/s und bei $max_k = 3$ lediglich noch 13k Anfragen/s. Neben der Effizienz muss allerdings auch die Qualität überzeugen, was bei $max_k = 1$ nicht der Fall ist. Die Pairs Completeness beträgt lediglich 0.16 und die Pairs Quality 0.1. Für $max_k = 2$ liegt die Pairs Completeness bei guten 0.95, auch die Pairs Quality ist mit 0.13 leicht besser. Für $max_k = 3$ kann diese sich noch auf 0.98 verbessern, die Pairs Quality bleibt mit 0.13 jedoch gleich. Sowohl $max_k = 2$ als auch $max_k = 3$ bieten einen guten Recall, da $max_k = 2$ jedoch deutlich effizienter ist, wird dieser Wert für die Evaluation genutzt. Die maximale Disjunktion lag bei drei in Verbindung mit $max_k = 3$, weshalb für max_d der Wert 3 für die Evaluation ausreichend erscheint.

Blockfilter

In Abschnitt 4.2.3 wurden zwei Filter für den Blocking Schema Lerner eingeführt, die dafür sorgen, dass offensichtlich schlechte Ausdrücke und schlechte Blockschlüssel nicht im Detail betrachtet werden, wenn diese zur Überlastung der Arbeitsspeicherkapazitäten führen können. Der erste Filter ist die Schwelle t, ab welcher ein Block mit mehr Einträgen als schlechter Block gilt. Der zweite Filter ist die minimale gute Blockrate, ist

t \ g	0.75	0.80	0.85	0.90	0.95	1.0
25	0.87/0.27	0.16/0.47	0.16/0.47	0.16/0.47	0.16/0.47	0.06/0.57
50	0.92/0.18	0.92/0.18	0.92/0.18	0.16/0.44	0.16/0.44	0.06/0.57
100	0.95/0.13	0.95/0.13	0.95/0.13	0.95/0.13	0.16/0.41	0.14/0.25
200			0.98/0.09	0.98/0.09	0.98/0.09	0.14/0.25
500						
1000						

Tabelle 5.3 Tabelle mit Recall=Pairs Completeness und Precision=Pairs Quality für unterschiedliche maximale Blockgrößen und minimale gute Blockrate.

die Anzahl der Einträge in guten Blöcken prozentual kleiner als g wird der komplette Ausdruck verworfen. Erfüllt ein Ausdruck die gute Blockrate, aber erzeugt Blockschlüssel größer t, dann werden diese verboten und nur die Blöcke kleiner t benutzt. Bei den meisten schlechten Blockschlüsseln handelt es sich um Stoppwörter, die in der Vorverarbeitung nicht korrekt aussortiert wurden. In der Evaluation wurde t für die Werte 25, 50, 100, 200, 500 und 1000 jeweils auf g 0.75, 0.8, 0.85, 0.9, 0.95 und 1 angewandt. Tabelle 5.3 zeigt Recall und Presion für alle Kombinationen von g und t. Für t gleich 500 und 1000 konnten keine Ergebnisse ausgewertet werden, da diese zu Abbrüchen aufgrund zu hoher Speicheranforderungen geführt haben. Weitere Abbrüche gab es für t gleich 200 in Verbindung mit g gleich 0.75 und 0.85. Zudem ist eine minimale gute Blockrate von 1.0 offensichtlich ungeeignet, weil zu viele gute Ausdrücke dadurch ausgeschlossen werden. Die übrigen Paarungen beschränken sich auf zwei Blocking Schemata. Eines mit schlechter Pairs Completeness und mittelmäßiger Pairs Quality und eines mit guter Pairs Completeness und schlechter Pairs Quality. Über einen Klassifikator kann eine schlechte Pairs Quality zu einer guten Precision verbessert werden, der Recall hingegen ist jedoch durch die Pairs Completeness beschränkt (Recall $\leq$ Pairs Completeness). Deswegen ist das zweite Blocking Schema mit höherer Pairs Completeness zu bevorzugen. In Abbildung 5.11 ist die Precision-Recall Kurve (hier: Precision=Pairs Quality, Recall=Pairs Completeness) für das zweite Blocking Schema mit steigender maximaler Blockgröße abgebildet. Je größer t, desto größer ist entsprechend auch der Recall, da weniger Blockschlüssel verboten werden. Allerdings fällt gleichzeitig die Precision. Eine robuste maximale Blockgröße, die für alle gewählten g funktioniert wird mit 100 ausgewählt. Für die minimale gute Blockrate wird 0.85 gewählt, da dort jeweils nach unten und oben noch Puffer ist, in welchem das präferierte Blocking Schema ebenfalls ausgewählt wurde. Für deutlich größere Datensätze als den NCVoter wird die Blockgröße von 100 vermutlich zu Verschlechterungen der Effizienz führen. Da allerdings in der Evaluation kein größerer Datensatz genutzt wird, ist die gewählte Blockgröße vertretbar.

lt (w=2)	P	N	PC	PQ
0.1	551k	8k	0.95	0.11
0.2	551k	655k	0.95	0.11
0.3	551k	1,377k	0.95	0.11
0.4	433k	1,377k	0.15	0.10
0.5	433k	1,377k	0.16	0.10

lt (w=5)	P	N	PC	PQ
0.1	551k	32k	0.15	0.01
0.2	551k	1,377k	0.15	0.01
0.3	551k	1,377k	0.15	0.01
0.4	551k	1,377k	0.16	0.11
0.5	551k	1,377k	0.16	0.11

lt (w=10)	P	N	PC	PQ
0.1	551k	66k	0.15	0.01
0.2	551k	1,377k	0.15	0.01
0.3	551k	1,377k	0.15	0.01
0.4	551k	1,377k	0.16	0.11
0.5	551k	1,377k	0.16	0.13

lt (w=20)	P	N	PC	PQ
0.1	551k	119k	0.15	0.01
0.2	551k	1,377k	0.15	0.01
0.3	551k	1,377k	0.15	0.01
0.4	551k	1,377k	0.16	0.11
0.5	551k	1,377k	0.16	0.12

Tabelle 5.4 Auswertung der Konfiguration mit selbständig synthetisierter Ground Truth für verschiedene Fenstergrößen. Die im Kontrollexperiment ermittelte Pair Completeness beträgt im Vergleich 0.95 und Pairs Quality 0.13.

5.3.2 Label Generator

Die freien Parameter des Labelgenerators sind die Fenstergröße, die untere und obere Schwelle, sowie die maximalen Matches und Non-Matches. Dabei werden die Schwellen nur benötigt, falls keine Ground Truth existiert und der Label Generator diese selbstständig erzeugt. Die Auswertung der eigenständig generierten Ground Truth erfolgt durch die bekannten Matches und wird in Relation zu einem Kontrollexperitment, das mit Ground Truth Matches durchgeführt wurde, gesetzt.

Fenstergröße

Zur Bestimmung einer geeigneten Fenstergröße wurde der Label Generator ohne Matches betrachtet (vgl. Abschnitt 3.1). Diese Variante reagiert im Vergleich zur Variante mit Matches deutlich empfindlicher auf die unterschiedlichen Parameter, wodurch der Effekt der unterschiedlichen Fenstergrößen w einfacher ausgewertet werden kann. Dazu wurde die untere Schwelle lt und die obere Schwelle ut in 0.1 Schritten bis 0.5 erhöht und jeweils mit den Fenstergrößen 2, 5, 10 und 20 ausprobiert. Die maximalen Matches wurden mit 10 % der Gesamtmenge und die maximalen Non-Matches mit 25 % der Gesamtmenge bestimmt. Bei dem genutzten NCVoter Datensatz sind somit die maximalen Matches bei 551k und die maximalen Non-Matches bei 1.337k. In der Tabelle 5.4 sind die Ergebnisse für die unterschiedlichen Fenstergrößen dargestellt. Die obere Schwelle ut hatte dabei keine entscheidende Auswirkung, sodass lediglich die untere Schwelle lt betrachtet wird. Für jede Schwelle wurden Matches (P), Non-Matches

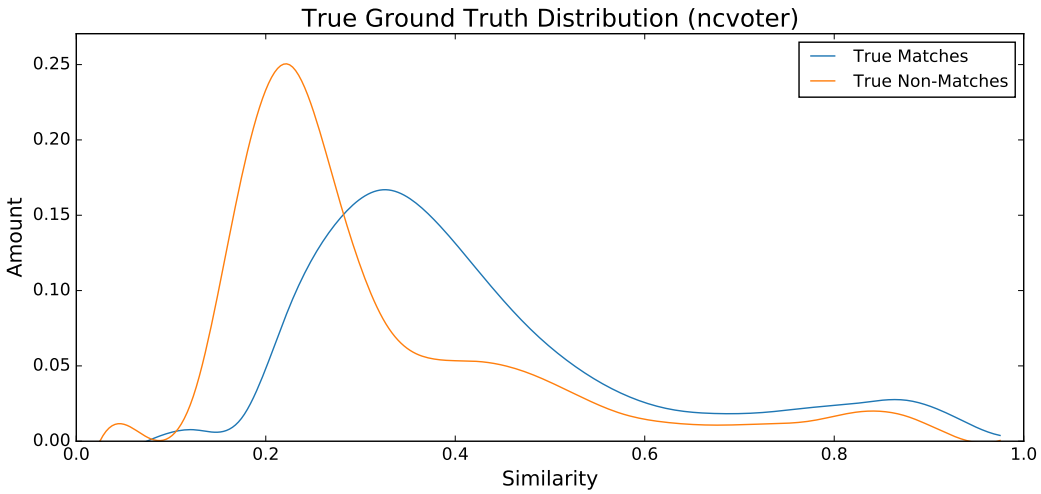

Abbildung 5.12 Ähnlichkeitsverteilung der TF/IDF Ähnlichkeiten der Ground Truth des NCVoter Datensatzes, welche anhand der tatsächlichen Matches erzeugt wurde. Die Datensätze Matches bzw. Non-Matches wurden in 5 % Schritten nach Ähnlichkeit zusammengefasst.

(N), Pairs Completeness (PC) und Pairs Quality (PQ) analysiert. Die Pairs Quality liefert keinen entscheidenden Hinweis auf ein geeignetes Fenster, da diese sich lediglich zwischen 1 % und 11 % hin und her bewegt. Dies bedeutet zwar einen Performanzunterschied, welcher jedoch nicht ausschlaggebend signifikant ist. Beim Blick auf die Pairs Completeness zeigt sich, dass diese sich zwischen 13 % und 16 % für alle Fenstergrößen bewegt, mit Außnahme von $w = 2$. Dort ist die Pairs Completeness für $lt \leq 0.3$ mit 95 % deutlich besser. Das Kontrollexperiment mit $w = 2$ erreicht ebenfalls eine Pairs Completeness von 95 % und die Pairs Quality ist mit 13 % lediglich 2 % besser. Für $w = 2$ werden am wenigsten Paare gebildet, allerdings sind die Paare die gebildet werden, die mit der höchsten Ähnlichkeit zueinander, da nur Blocknachbarn betrachtet werden. Aufgrund dessen werden viele deutlich verschiedene Paare ausgeschlossen, wie sich bei $lt = 0.1$ bemerkbar macht, da hier lediglich 8k Non-Matches generiert wurden. Bei $lt = 0.2$ gibt es mit 655k dann allerdings schon eine große Auswahl an Non-Matches und mit $lt = 0.3$ wurden bereits mehr Paare generiert als das Maximum. Aufgrund des guten Abschneidens gegenüber dem Kontrollexperiment und keiner wirklichen Konkurrenz durch andere Fenstergrößen wird diese für die weitere Evaluation mit 2 bestimmt.

Untere und obere Schwelle

Die untere Schwelle lt legt fest, bis zu welchem Ähnlichkeitswert Paare als Non-Matches betrachtet werden und die obere Schwelle ut legt fest, ab welchem Ähnlichkeitswert Paare als Matches betrachtet werden, wobei stets gilt $lt \leq ut$. In einem Experiment wurden lt und ut in 0.1 Schritten betrachtet und so alle Konfigurationen bis 1.0 auf dem NCVoter-Datensatz getestet. Für das Fenster wurde der bereits bestimmte Wert von 2 gesetzt. Zur

Auswertung wurden Pairs Completeness, Pairs Quality und die Ground Truth analysiert. Anhand der Pairs Completeness und Pairs Quality kann betrachtet werden, wie gut ein Blocking Verfahren auf der generierten Ground Truth funktioniert. Durch die gefilterte Ground Truth hingegen kann herausgefunden werden, wie viele Ground Truth Paare für den Fusion-Lerner zur Verfügung stehen. Die Tabellen 5.5, 5.6, 5.7 betrachten nacheinander die Pairs Completeness, die Matches und die Non-Matches. Die Pairs Quality ist uninteressant, da deren Werte relativ konstant bei 0.1 liegen, mit einer Varianz von 0.03. In Tabelle 5.5 ist gut zu sehen, dass die Pairs Quality zwischen einer ut von 0.1 und 0.4 immer eine gute Pairs Quality von 95 % erzeugt. Der Blick auf das erlernte Blocking Schema zeigt, dass dieses immer dasselbe und gleich zu dem aus Abschnitt 5.3.2 ist. Dies trifft auch noch teilweise für $ut = 0.5$ zu, allerdings nur für $lt \leq 0.3$. Für alle $ut > 0.5$ variieren die Blocking Schemata, wobei unabhängig von lt keines über 17 % Pairs Completeness kommt. In Abbildung 5.12 ist die Ähnlichkeitsverteilung der Ground Truth, des Kontrollexperiments dargestellt, die aus den tatsächlichen Matches erzeugt wurde. Der Großteil der Matches hat eine Ähnlichkeit zwischen 0.2 und 0.5, wohingegen der Großteil der Non-Matches sich überlappend zwischen 0.1 und 0.3 befindet. Für $ut > 0.5$ fällt der Recall dramatisch ab, weil ein Großteil der Matches nicht mehr erfasst wird und für $ut = 0.5$ sind die Recallwerte für $lt \leq 0.3$ noch gut. Wird lt allerdings weiter erhöht fällt der Recall, da sich nun zu viele tatsächliche Matches in den Non-Matches befinden.

Deshalb werden in den Tabellen 5.6, 5.7 lediglich ut-Werte kleiner 0.6 betrachtet. Die maximalen Matches, die der Label Generator erzeugen darf, liegen bei 10 % der Gesamtmenge und betragen 551k. Bei den Non-Matches ist das Limit 25 % und damit 1,3 mio. Für die künstliche Anreicherung der gefilterten Non-Matches stehen jedoch alle erzeugten Non-Matches zur Verfügung, dementsprechend je höher lt desto mehr Non-Matches und umgekehrt für ut. In Tabelle 5.6 ist zu sehen, dass für $ut \leq 0.4$ die Ausgangsmenge der Matches auf das Maximum beschränkt wurde. Da jeweils die Matches mit der höchsten Ähnlichkeit genutzt werden, sind diese Mengen identisch, weshalb auch die gefilterten Mengen mit jeweils 300k Datensätzen aufgrund desselben Blocking Schema identisch sind. Für $ut = 0.5$ ist die Ausgangsmenge kleiner als das Maximum. Die gefilterte Menge beträgt in diesem Fall 288k, was mehr als genügend Matches sind um einen Klassifikator zu trainieren. Zu beachten ist, dass diese Menge 6-Mal soviele Matches beinhaltet, wie die Menge tatsächlichen Matches.

In Tabelle 5.7 wird die Anzahl der Non-Matches dargestellt. Für $lt = 0.1$ sind insgesamt nur 8k Paare erzeugt worden, weil das TF/IDF Blocking die meisten der sehr unähnlichen Paare ausschließt. Nach dem Filtern durch das Blocking Schema sind keine Non-Matches mehr vorhanden, da das Blocking Schema verhindert, dass diese offensichtlichen Non-Matches zusammen gruppiert werden. Für $lt = 0.2$ gibt es ein ähnliches Bild. Zwar ist die Anzahl der Ausgangsmenge mit 655k deutlich höher, dennoch werden lediglich 66 Paare gefunden, die einen gemeinsamen Blockschlüssel haben, was zum Trainieren eines Klassifikators nicht ausreichend ist. Interessanter wird es erst ab $lt = 0.3$. Hier

PC	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1.0
0.1	0.95	0.95	0.95	0.95	0.95	0.15	0.15	0.15	0.15	0.13
0.2		0.95	0.95	0.95	0.95	0.15	0.15	0.15	0.15	0.13
0.3			0.95	0.95	0.95	0.15	0.15	0.16	0.13	0.13
0.4				0.95	0.15	0.16	0.16	0.16	0.14	0.14
0.5					0.16	0.16	0.13	0.14	0.14	0.06
0.6						0.13	0.14	0.14	0.16	0.06
0.7							0.14	0.10	0.16	0.06
0.8								0.10	0.16	0.06
0.9									0.14	0.06
1.0										0.06

Tabelle 5.5 Pairs Completeness bei verschiedenen Schwellen

Matches	0.1	0.2	0.3	0.4	0.5
0.1	551k/300k	551k/300k	551k/300k	551k/300k	443k/288k
0.2		551k/300k	551k/300k	551k/300k	443k/288k
0.3			551k/300k	551k/300k	443k/288k
0.4				551k/300k	443k/287k
0.5					443k/273k

Tabelle 5.6 Anzahl der Matches und gefilterten Matches bei verschiedenen Schwellen

Non-Matches	0.1	0.2	0.3	0.4	0.5
0.1	8k/0	8k/0	8k/0	8k/0	8k/0
0.2		655k/66	655k/66	655k/66	655k/66
0.3			1377k/2430	1377k/2430	1377k/2430
0.4				1377k/13916	1377k/11k
0.5					1377k/8k

Tabelle 5.7 Anzahl der Non-Matches und gefilterten Non-Matches bei verschiedenen Schwellen

wird das erste Mal das Maximum der Ausgangsmenge mit 1377k erreicht. Die gefilterten Non-Matches betragen 2430, was im Vergleich zu den Matches immer noch sehr wenig ist, aber durchaus für das Klassifikatortraining genügt. Mit $lt = 0.4$ erhöht sich diese Anzahl nochmals um das 5-fache. Ein Blick auf Abbildung 5.12 zeigt aber, dass sich dadurch der Großteil der Matches in den Non-Matches befindet. Die Linien für Matches und Non-Matches schneidet sich im Bereich 0.1 - 0.5 bei 0.28. Unterhalb gehen viele Non-Matches verloren und oberhalb viele Matches. Deshalb wird sowohl lt als auch ut auf 0.3 festgelegt, da ab hier auch genügend Paare zum Trainieren eines Klassifikators zur Verfügung stehen.

Sowohl mit, also auch ohne Ground Truth Match werden dem Label Generator mitgeteilt wie viele Matches max_p bzw. Non-Matches max_n in der Ground Truth enthalten sein dürfen. Im Fall ohne Ground Truth Matches werden jeweils die max_p Matches, bzw. max_n Non-Matches ausgewählt, welche die höchste Ähnlichkeit haben. Im Fall mit Ground Truth Matches werden sowohl Matches als auch Non-Matches nach ihrer Ähnlichkeitsverteilung ausgewählt. Mithilfe dieser Ground Truth sucht der DNF Blocking Schema Lerner nach dem besten Blocking Schema. Anschließend wird die Ground Truth anhand des Blocking Schema gefiltert, sodass diese nur Paare enthält, die einen gemeinsamen Blockschlüssel haben. In diesem Schritt werden sehr viele Non-Matches herausgefiltert, da das der primäre Zweck des Blocking Schema ist. Damit für den Fusion-Lerner genügend Non-Matches zur Verfügung stehen, werden die Non-Matches mit allen generierten Non-Matches des Label Generators angereichert. Hierbei spielt max_n keine Rolle. Die Auswirkungen dieser Parameter wurden auf dem NCVoter Datensatz mit Ground Truth und ohne Ground Truth mit Fenstergröße $w = 2$ und Schwellen $lt = ut = 0.3$ evaluiert. Ausgangssituation ist (0.1, 0.25) mit max_p 10 % und max_n 25 % der Gesamtmenge, welche bereits zur Ermittlung der Fenstergröße und der Schwellen genutzt wurden. Bei allen getesteten Paarungen ist das Limit der Non-Matches höher als das der Matches, um das Verhältnis im Datensatz zu repräsentieren. Getestet wurden zunächst größere Werte mit (0.5, 2.5), (1, 5), (5, 25), (10, 50). Dabei wuchsen die Matches leicht auf 650k und die Non-Matches bis stark auf 19 Mio. Auf das Blocking Schema hat die vergrößerte Ground Truth in keinem Fall einen Einfluss. Die Betrachtung der Matches im größten Fall (10, 50) ergibt, dass diese sich lediglich um ca. 100k verändert hat. Die neu hinzugekommen Matches wurden durch die Filterung auf ein paar Hundert reduziert. Das bedeutet, dass die erweiterten Matches die Ausdrücke des Blocking Schema, im Bezug auf Pairs Completeness, abwerten und zwar je mehr, desto größer die Anzahl der Matches. Allerdings haben sich die Non-Matches mit 19 mio. dramatisch erhöht, durch die Filterung bleiben lediglich die bekannten 2k übrig, sodass die erhöhte Menge sich positiv auf das Reduction Ratio und die Pairs Quality auswirkt. Damit wird der Negativeffekt auf die Pairs Completeness aufgehoben und das Blocking Schema bleibt dasselbe. Neben größeren Werten wurden auch kleinere getestet (0.0001, 0.00025), (0.001, 0.0025), (0.01, 0.025). Für (0.01, 0.25) werden in etwa so viele Matches ausgewählt wie tatsächlich enthalten sind. Recall und Precision sind allerdings mit 15 % und 0.1 % sehr schlecht. Dasselbe Ergebnis zeigt sich bei den noch kleineren Paaren. Daraus folgt, dass das Ausgangspaar (0.1, 0.25) bereits gut gewählt war und auch robuste Werte liefert.

5.3.3 Fusion-Lerner

Für den Fusion-Lerner müssen noch vier freie Parameter bestimmt werden:

- Anzahl der k Teilmengen für die Kreuzvalidierung
- Anzahl der maximalen Paare beim Subsampling und deren Verhältnis zu Matches und Non-Matches
- Qualitätsmaß zur Bewertung der trainierten Modelle

Bei der Kreuzvalidierung durch das Stratified K-Fold Verfahren wird die Anzahl für K mit 3 bestimmt, da mit anderen Werten kein nennenswerter Unterschied gemessen wurde und dieser der Standardwert in Scikit-learn ist.

Während der Entwicklung hat sich eine Grid Search auf 5.000 Paaren als genügend effizient herausgestellt, dabei wurde eine Verhältnis von einem Match auf drei Non-Matches benutzt. Im Rahmen dieser Arbeit war es zeitlich nicht mehr möglich andere Werte zu evaluieren, weshalb diese für die Evaluation übernommen wurden. Das Sampling der Paare wird dabei anhand der Ähnlichkeitsverteilung durchgeführt.

Ein Qualitätsmaß muss sowohl Recall als auch Precision berücksichtigen, deshalb wurden hierzu das F-measure und die Average Precision betrachtet. In mehreren Durchläufen wurden die beiden Maße jeweils für einen Klassifikator genutzt. Dabei wurde zunächst festgestellt, dass Recall und Precision immer gleich sind, unabhängig davon welcher Klassifikator und welche Parameter durch die Grid Search bestimmt wurden. Die Ergebnisse wurden gegen Baseline verglichen, bei welcher kein Klassifikator genutzt wurde und die Kandidatenmenge C der Ergebnismenge R entspricht. Hierbei wurde festgestellt, dass mit dem Klassifikator die Precision dieselbe und der Recall um 13 % niedriger ist. Zweck des Klassifikators ist es jedoch, möglichst ohne Verlust des Recalls die Precision so nah wie möglich an 1 zu bringen. Aufgrund dieses Ergebnisses kann keine endgültige Entscheidung über das Qualitätsmaß getroffen werden. Das Problem wird deshalb in Abschnitt 5.4 genau analysiert. Für diese Untersuchung wird jedoch ein Qualitätsmaß benötigt, weshalb zunächst das F-measure ausgewählt wird, welches bereits für das Blocking Verfahren gute Dienste leistet.

5.4 Einfluss der Ähnlichkeitsvektoren

Der Grund für das Versagen der Klassifikatoren bei der Auswahl der freien Parameter des Klassifikators liegt vermutlich an der reduzierten Menge von Ähnlichkeitswerten, die der MDySimIII für Kandidatenpaare zurückgibt, da nur Ähnlichkeiten für Attribute angegeben werden, die einen gemeinsamen Block haben. Beim Blick auf die Kandidatenmenge wurde festgestellt, dass dies für fast alle Paare lediglich ein Attribut ist. Damit der Ähnlichkeitsvektor in mehr Stellen besetzt ist wurde der MDySimIII in zwei Varianten modifiziert. Die erste Variante berechnet die fehlenden Ähnlichkeitswerte der Attribute, die Teil des Blocking Schema sind. Die zweite Variante berechnet alle fehlenden Ähnlich-

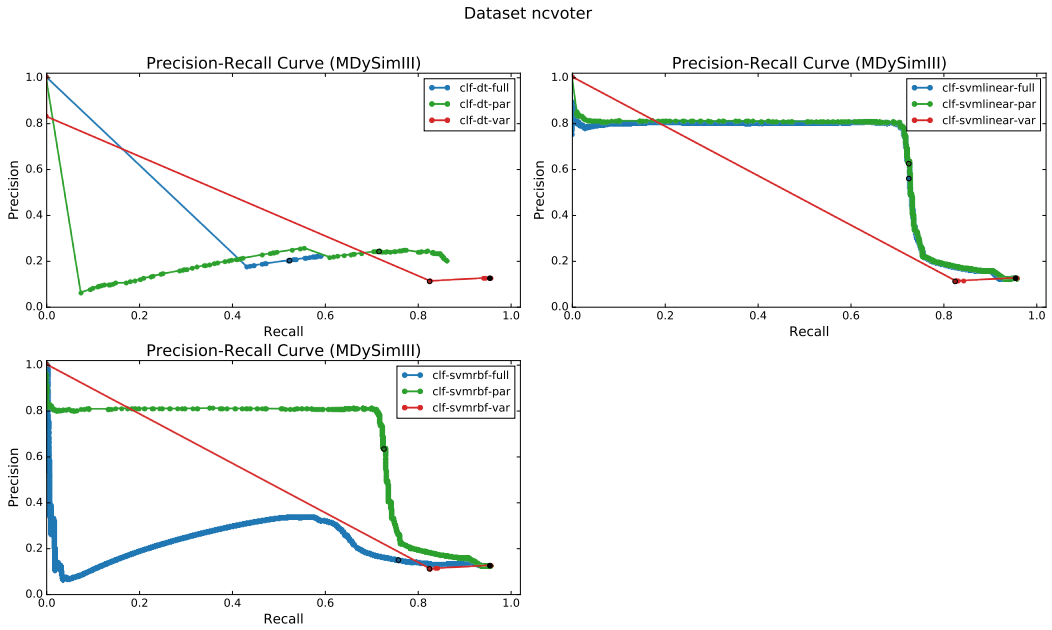

Abbildung 5.13 Precision-Recall Kurven für Varvektor (var), Teilvektor (par) und Vollvektor (full) Ähnlichkeiten, gruppiert nach Klassifikator (clf): Decision Tree (dt), SVM mit RBF-Kernel (svmrbf) und SVM mit Linearkernel (svmlinear).

keitswerte, sodass der Vektor anschließend vollbesetzt ist. Das ursprüngliche Verfahren mit variable besetzem Vektor wird im Folgenden als Varvektor (par) bezeichnet, die erste neue Variante als Teilvektor (par) und die zweite neue Variante als Vollvektor (full). Zum Vergleich wurde jeweils ein Kontrollexperiment (KE) ohne Klassifikator, aber mit entsprechender Ähnlichkeitsberechnung durchgeführt. Die Ähnlichkeitsberechnung beeinflusst hierbei lediglich die Laufzeit, da ohne Klassifikator die Ähnlichkeiten nicht weiter verwendet werden und die Kandidatenmenge C der Ergebnismenge R entspricht. Die folgenden sechs Konfigurationen werden hierzu evaluiert:

- Var-KE, MDySimIII ohne Klassifikator mit vorausberechneten Ähnlichkeiten
- Teil-KE, MDySimIII ohne Klassifikator mit Blocking Schema Attributsähnlichkeitsberechnung
- Voll-KE, MDySimIII ohne Klassifikator mit voller Ähnlichkeitsberechnung
- Varvektor, MDySimIII mit Klassifikator mit vorausberechneten Ähnlichkeiten
- Teilvektor, MDySimIII mit Klassifikator mit Blocking Schema Attributsähnlichkeitsberechnung
- Vollvektor, MDySimIII mit Klassifikator mit voller Ähnlichkeitsberechnung

Alle sechs Konfigurationen nutzen dazu dieselbe Ground Truth mit vorklassifizierten Matches, dasselbe Blocking Schema, sowie die gleichen vom Similarity Lerner bestimmten Ähnlichkeitsmetriken. Des Weiteren wurden Varvektor, Teilvektor und Vollvektor je-

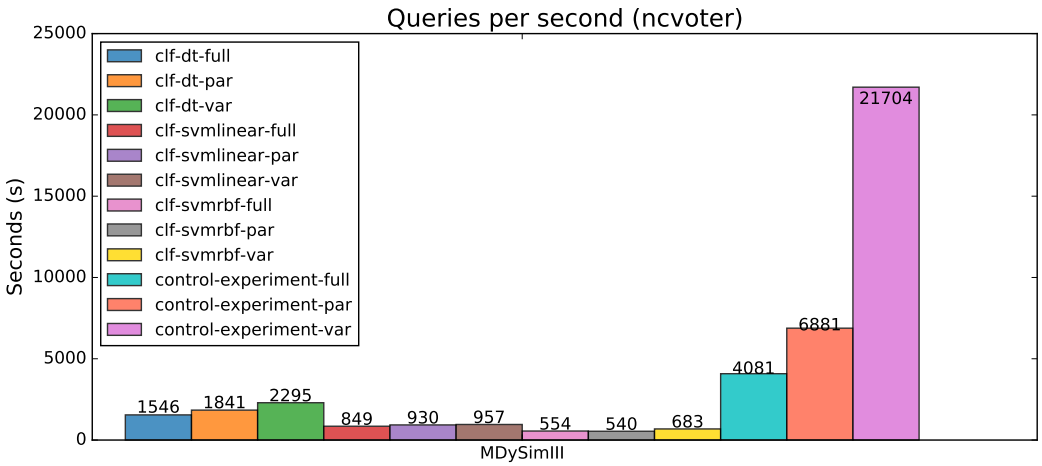

Abbildung 5.14 Anfragen pro Sekunde für Varvektoren (var), Teilvektoren (par) und Vollvektoren (full) bei Klassifikation durch verschiedene Klassfifkatoren.

weils mit Decision Tree (dt), SVM mit RBF-Kernel (svmrbf) und SVM mit Linearkernel (svmlinear) getestet, um die Ergebnisse sinnvoll miteinander vergleichen zu können.

Abbildung 5.13 gruppiert die Precision-Recall Kurven von Varvektor, Teilvektor und Vollvektor nach Klassifikator. In jeder Kurve ist durch einen Punkt das Recall-Precision Paar hervorgehoben, das anhand der im entsprechenden Klassifikator benutzen Wahrscheinlichkeitsschwelle erreicht wurde. Das Recall-Precision Paar der Kontrollexperimente ist jeweils durch einen schwarzen Punkt am Ende der Kurven gekennzeichnet. Auf den drei Varvektorkurven sind insgesamt nur vier Punkte zu erkennen. Jeder Punkt entspricht einer Wahrscheinlichkeitsschwelle, das bedeutet, dass die auf den Ähnlichkeiten berechneten Wahrscheinlichkeiten der Zugehörigkeit zu den Matches sich auf vier Wahrscheinlichkeiten beschränkt. Der Grund dafür liegt im erlernten Blocking Schema: [(`Vorname`, `ID`) $\wedge$ (`Nachname`, `ID`)] $\vee$ [(`Zweitname`, `ID`) $\wedge$ (`Strasse`, `ID`)]. Dieses besteht ausschließlich aus ID Prädikaten. Folglich werden nur Datensätze mit gleichen Attributen gruppiert, deren berechnete Ähnlichkeit offensichtlich 1.0 ist. Dementsprechend sind die Ähnlichkeitsvektoren der Kandidaten für Vorname und Nachname mit 1.0 besetzt oder für Zweitname und Strasse oder beides. Der vierte Wert ist eine 1.0 nur für die Strasse. Dieser kommt zustande, wenn beide Datensätze eine Übereinstimmung in der Strasse haben und keine Zweitnamen besitzen. In diesem Fall besteht der Blockschlüssel nur aus dem Strassenamen und die Ähnlichkeit bei mindestens einem leeren Attribut ist per Definition immer 0. Dementsprechend bekommt der Klassifikator nur ungenügend Informationen um eindeutig die Non-Matches herauszufiltern, was die schlechten Werte aus Abschnitt 5.3.3 erklärt. Die drei Kurven der Varvektoren zeigen zudem, dass es nicht möglich ist einen besseren Wert anzunehmen als der des Kontrollexperimentes. Im Gegensatz dazu bietet sowohl Teilvektor als auch Vollvektor dem Klassifikator deutlich mehr Unterscheidungsmerkmale. Dies lässt sich in den Precision-Recall Kurven daran erkennen, dass deren Kurven mit deutlich mehr Punkten besetzt sind. Überraschend

ist, dass der Teilvektor trotz weniger Ähnlichkeiten für den Decision Tree und die SVM mit RBF-Kernel besser abschneidet als der Vollvektor und für die SVM mit Linearkernel eine fast identische Kurve hat. Die beiden SVMs mit Teilvektor sind dazu dem Decision Tree deutlich überlegen, welcher über 20 % an Recall verliert und die Precision lediglich um 11 % steigern kann. Bei den SVMs mit Teilvektor liegt der Recall verlust zwar auch bei über 20 %, allerdings ist die Precision mit ca. 60 % eine deutliche Verbesserung. Zudem lässt sich in den Kurven ablesen, dass die Precision des Klassifikators durch Kalibrierung der Wahrscheinlichkeitsschwelle auf bis zu 80 % bei fast gleichem Recall optimiert werden kann.

In Abbildung 5.14 werden die Konsequenzen auf die Effizienz in Anfragen pro Sekunde der verschieden Konfiguration dargestellt. Mit deutlichem Abstand am meisten Anfragen pro Sekunde können durch Var-KE erzielt werden. Bei Par-KE sind es bereits ca. 70 % weniger und bei Full-KE ganze 80 % weniger Anfragen pro Sekunde. Aber nicht ausschließlich die Ähnlichkeitsberechnung benötigt viel Zeit, wie bei den nicht Kontrollexperimenten zu sehen ist, bremst die Klassifikation ebenfalls erheblich, sogar stärker als die Ähnlichkeitsberechnung. Der Klassfikationsaufwand ist unabhängig von der Anzahl der Stellen, an welchen der Ähnkeitsvektor besetzt ist. Die beste Performanz bieten entsprechend die Varvektoren, welche jedoch eine extrem schlechte Qualität haben. Die Teilvektoren sind über die Hälfte langsamer als die Varvektoren, aber ca. 70 % schneller als die Fullvektoren. Des Weiteren sind die Qualitätswerte mindestens genauso gut, z.T. sogar besser als die der Fullvektoren.

Beim Auswerten der Daten gab es eine Unstimmigkeit. Und zwar wurde als Ähnlichkeitsmaß für alle Attribute immer Bag-Distanz gewählt. Insgesamt wurden in dem Similarity Lerner die vier Metriken Bag-Distanz, Levenshtein-Distanz, Damerau-Distanz und Jaro-Distanz zur Auswahl übergeben. Ein genauer Blick auf die vom Similarity Lerner berechnete Average Precision zeigt, dass diese pro Attribut für jedes Ähnlichkeitsmaß den gleichen Wert hat. Dementsprechend wurde immer die zuerst getestete Metrik ausgewählt. Dieses Verhalten des Similarity Lerners wird in Abschnitt 5.6 genauer analysiert.

5.5 Einfluss der Ähnlichkeitsmetriken

Die Auswertung der Ähnlichkeitsmetriken in Abschnitt 5.4 zeigt, dass der Similarity Lerner für jedes Attribute für die meisten Metriken denselben Average Precision Wert berechnet. Daraus folgt, dass entweder die Ähnlichkeitsmetriken keinen oder nur geringen Einfluss auf das Urteilsvermögen des Klassifikators haben, oder dass die Average Precision kein geeignetes Maß ist, um eine Ähnlichkeit auszuwählen. Infolgedessen wird der Einfluss der Ähnlichkeitsmetriken geprüft, indem pro Durchlauf jeweils eine Metrik für alle Attribute manuell bestimmt wird. Dabei werden die Durchläufe für jeden der

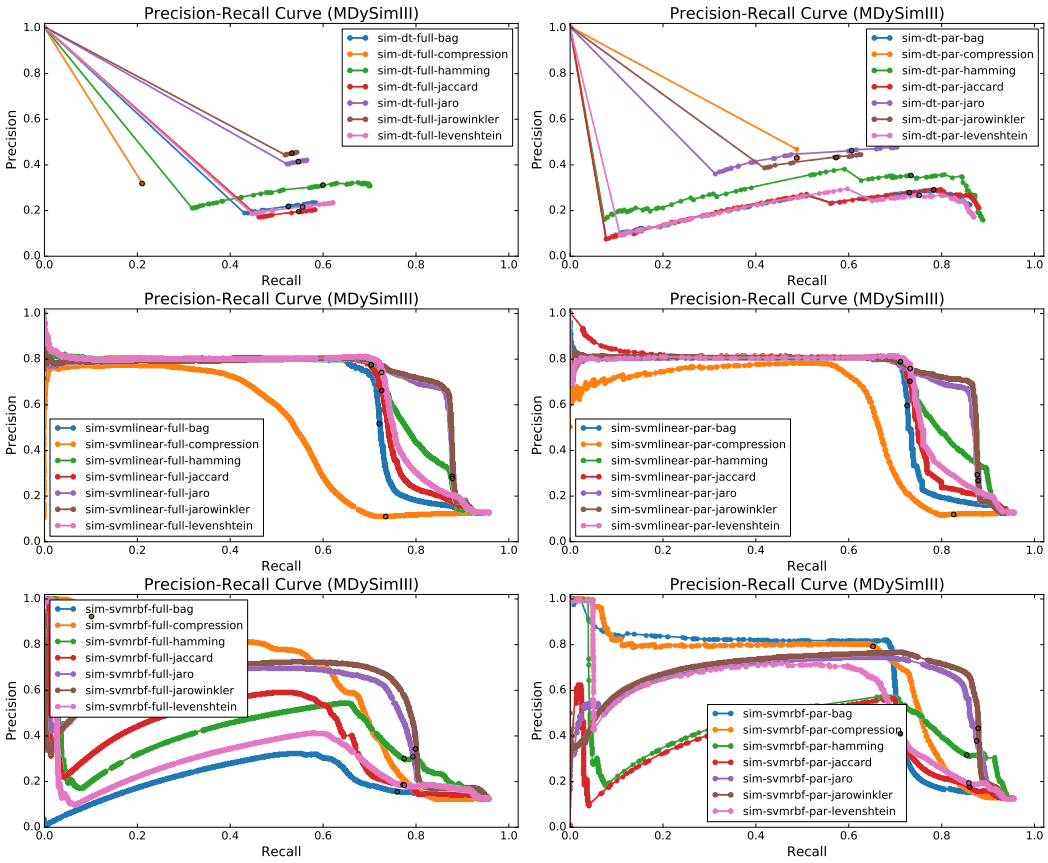

Abbildung 5.15 Precision-Recall Kurven von 7 Ähnlichkeitsmetriken gruppiert nach Klassifikator (Zeilen) und Vektortyp (Spalten). Durch einen Punkt ist die aktuelle Wahrscheinlichkeitsschwelle des Klassifikators hervorgeboben.

drei Klassifikatoren wiederholt. In Abschnitt 5.4 wurde bzgl. Qualität und Effizienz eine Tendenz zu den Teilvektoren gegenüber den Vollvektoren festgestellt. Zum Zweck diese Tendenz zu bestärken bzw. zu entkräften wird jeder der Klassifikatoren einmal mit dem Teilvektor und einmal mit dem Vollvektor geprüft. Für die Analyse wurde die Damerau-Distanz weggelassen, da sich deren berechnete Ähnlichkeiten bei genauer Betrachtung zu fast 100 % mit der Levenshein-Distanz deckt. Zusätzlich wurden ausführlichkeitshalber 4 weitere Ähnlichkeitsmetriken hinzugenommen:

- Bag-Distanz
- Compression-Distanz
- Hamming-Distanz
- Jaro-Distanz
- Jaro-Winkler-Distanz
- Jaccard-Koeffizent
- Levenshtein-Distanz

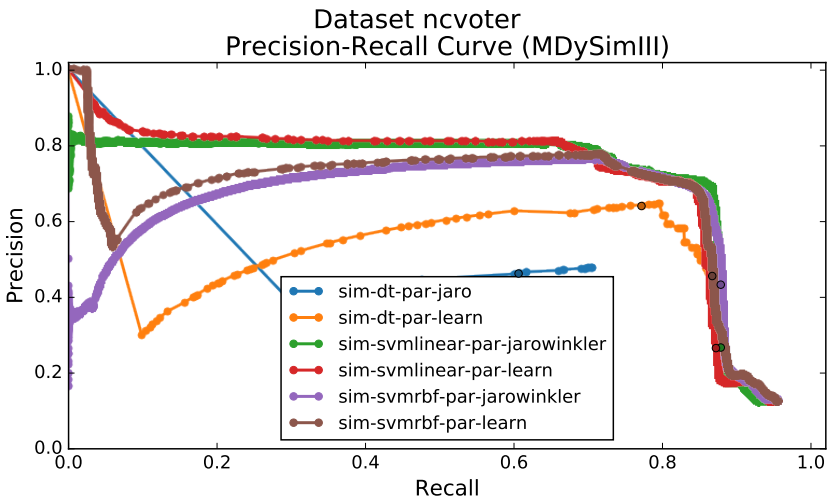

Abbildung 5.16 Precision-Recall Kurven der Teilvektoren (par) durch die gelernten Ähnlichkeitsmaße und Teilvektoren mit jeweils der besten einheitlich Ähnlichkeit.

Bei 7 Ähnlichkeitsmetriken, drei 3 Klassifikatoren und 2 Vektortypen wurden insgesamt 42 Durchläufe ausgeführt und ausgewertet. Des Weiteren wurde für alle Durchläufe dieselbe Ground Truth mit vorklassifizierten Matches und dasselbe Blocking Schema verwendet. Das Blocking Schema ist erneut [(`Vorname`, `ID`) $\wedge$ (`Nachname`, `ID`)] $\vee$ [(`Zweitname`, `ID`) $\wedge$ (`Strasse`, `ID`)]. Die für den Similarity Learner und Klassifikator interessante gefilterte Ground Truth besteht aus 48.256 Matches, 61.764 Non-Matches.

Das Ergebnis ist in den Precision-Recall Kurven in Abbildung 5.15 zu sehen. Dabei wurden die Kurven für die Ähnlichkeitsmetriken jeweils nach Klassifikator und Vektortyp gruppiert. In der linken Spalte sind die Vollvektoren, in der rechten die Teilvektoren und in den Zeilen von oben nach unten der Decision Tree, die SVM mit Linearkernel und die SVM mit RBF-Kernel. Die Vermutung, dass der Einfluss der Ähnlichkeitsmetriken nicht vorhanden bzw. gering ist, kann auf einen Blick wiederlegt werden. Beispielsweise ist im Plot für die SVM mit Linearkernel und Teilvektor ein deutlicher Unterschied zwischen den Kurven der Compression-Distanz und der Jaro-Winkler-Distanz zu erkennen, womit erwiesen ist, dass diese Metriken entscheidenden Einfluss auf das Urteilsvermögen des Klassifikators haben. Des Weiteren bestätigt der Vergleich der Vollvektorkurven mit den Teilvektorkurven die Entscheidung zu letzterem, da dieser dem Klassifikator bessere Werte liefert und somit die Kurven mindestens gleich, meist aber besser sind. Insgesamt schneidet die SVM mit Linearkernel am besten ab. Allerdings sind die vom Klassifikator genutzten Wahrscheinlichkeitsschwellen (hervorgehobener Punkt auf der Kurve) nicht optimal. Beispielsweise können die Jaro-Distanz und die Jaro-Winkler-Distanz von einer Kalibrierung der Schwelle massiv profitieren, weil dadurch die Precision bei fast gleichem Recall um ca. 40 % verbessert werden kann.

Metrik	Vorname	Nachname	Zweitname	Strasse
Bag	0.7111	0.6982	0.7121	0.4295
Compression	0.4286	0.4620	0.4174	0.3850
Hamming	0.7111	0.6999	0.7121	0.4288
Jaccard	0.7198	0.6995	0.7130	0.4269
Jaro	0.7111	0.6989	0.7121	0.4295
Jaro-Winkler	0.7112	0.7005	0.7121	0.4289
Levenshtein	0.7111	0.6989	0.7121	0.4295

Tabelle 5.8 Tabelle mit Average Precisions pro Attribut für 7 Ähnlichkeitsmetriken.

Die zweite Vermutung, dass die Average Precision keine geeignete Metrik ist um Ähnlichkeiten auszuwählen wurde überprüft, indem die Durchläufe mit Teilvektor und allen 7 Ähnlichkeitsmetriken wiederholt wurden. Die Ground Truth und das Blocking Schema sind gleich zu den 42 Durchläufen zuvor. Die ausgewählten Ähnlichkeitsmetriken sind: Vorname → Jaccard-Distanz, Nachname → Jaro-Winkler-Distanz, Zweitname → Jaccard-Distanz und Strasse → Bag-Distanz. In Tabelle 5.8 sind die errechneten Average Precision Werte pro Attribut aufgelistet. Die Compression-Distanz fällt hierbei aus dem Rahmen und hat deutlich schlechtere Werte als der Rest. Die Werte für Bag-Distanz, Jaro-Distanz und Levenshtein-Distanz sind wie zuvor identisch. Für die Hamming-Distanz, Jaccard-Distanz und Jaro-Winkler-Distanz gibt es dazu leicht aufweichende Werte, was zur Auswahl unterschiedlicher Metriken geführt hat. In Abbildung 5.16 sind die drei Kurven der gelernten Ähnlichkeitsmaße mit jeweils der besten Kurve des entsprechenden Klassifikator aus Abbildung 5.15 gegenübergestellt. Aufgrund der Auswahl sind die Kurven diesmal deutlich besser. Für die SVMs ist die Kurve beinahe so gut wie die beste Kurve und für den Decision Tree ist die Kurve sogar deutlich besser. Aufgrunddessen ist die Average Precision ein geeignetes Maß, sofern genügend Metriken zur Auswahl stehen.

5.6 Human Baseline

In diesem Abschnitt wird das selbstkonfigurierende System mit einer Ausgangskonfiguration (Baseline) verglichen. Diese Baseline wurde von einem Kommilitonen erstellt und besteht aus einem Blocking Schema BS und Ähnlichkeitsfunktionen S für die jeweiligen Attribute. Dabei wurden dem Kommilitonen zuvor die Aufgabenstellung und Konsequenzen für Qualität und Effizienz erläutert. Zudem wurden praktische Hinweise für ein gutes Blocking Schema [30, S. 98] und gute Ähnlichkeitsfunktionen [30, S. 126], z. B. Attribute zu verknüpfen, welche unabhängig voneinander sind, geteilt. Der Zweck dieses Vergleich ist zu betrachten, ob sich die Selbstkonfiguration gegenüber einer menschlichen, manuellen Konfiguration behaupten kann. In beiden Fällen wurden dieselben in diesem Kapitel ermittelten freien Parameter benutzt. Die Auswahl der Ähnlichkeitsfunk-

Konfiguration	Vorname	Zweitname	Nachname	Strasse	PLZ	Tel.Nr.
Baseline	Jaccard	Hamming	Jaccard	Hamming	Jaccard	Jaccard
Selbstkonf.	Jaccard	Jaro-Winkler	Jaccard	Jaro	-	-

Tabelle 5.9 Gelernte und bestimmte Ähnlichkeitsmaße für die Baseline und die Selbstkonfiguration.

tionen bestand für Mensch und Maschine aus den in Abschnitt 5.1.2 vorgestellten sieben Funktionen. Die Evaluation der Konfigurationen wurde zunächst auf den Trainingsdaten durchgeführt, wodurch die Robustheit der gewählten freien Parameter erstmals auf die Probe gestellt wurde. Der Eventstrom in der Query-Phase wurde einmal durch die Trainingsanfragen und ein zweites mal durch die Testanfragen erzeugt. Beide beinhalten jeweils 1.3 Mio. Datensätze. Im folgenden sind die Blocking Schemata des selbstkonfigurierende Systems und der Baseline dargestellt.

Baseline:

$$\begin{aligned}((\text{Nachname}, \text{ID}) \wedge (\text{Strasse}, \text{Token}) \wedge (\text{PLZ}, \text{ID})) \quad &\vee \\ ((\text{Vorname}, \text{ID}) \wedge (\text{Zweitname}, \text{ID})) \quad &\vee \\ (\text{Telefonnummer}, \text{ID})\end{aligned}$$

Selbstkonfiguration:

$$\begin{aligned}((\text{Vorname}, \text{ID}) \wedge (\text{Nachname}, \text{ID})) \quad &\vee \\ ((\text{Zweitname}, \text{ID}) \wedge (\text{Strasse}, \text{ID}))\end{aligned}$$

Die Tabelle 5.9 Tabelle zeigt die pro Attribut ausgewählten Ähnlichkeitsfunktionen. Für alle vier Durchläufe wurde vom Fusion-Lerner eine SVM mit RBF-Kernel und Strafparameter C der Fehlerfunktion gleich 1000 trainiert.

In Abbildung 5.17 sind die Precision-Recall Kurven der beiden Baselines für Anfragen auf den Trainingsdaten (train) und Testdaten (test), sowie die beiden Kurven der Selbstkonfiguration für die Konfiguration auf den Trainingsdaten und Anfragen auf den Trainingsdaten (train-train), bzw. Testdaten (train-test) dargestellt. Dabei ist zunächst zu sehen, dass die Test- und Trainingskurven jeweils sehr nahe beianander sind. Dem kann entnommen werden, dass bei der Konfiguration keine Überanpassung stattgefunden hat und die Entity Resolution entsprechend auf beiden annährend gleich gut funktioniert. Die Kurven sind jeweils durch die Pairs Completeness und Pairs Quality limitiert, d.h. der Recall kann nicht größer werden als die Pairs Completeness und die Precision nicht geringer als die Pairs Quality in Tabelle 5.10. Obwohl die Kurven der Selbstkonfiguration Recall und Precision besser maximieren können, ist der Klassifikator mit einer Stan-

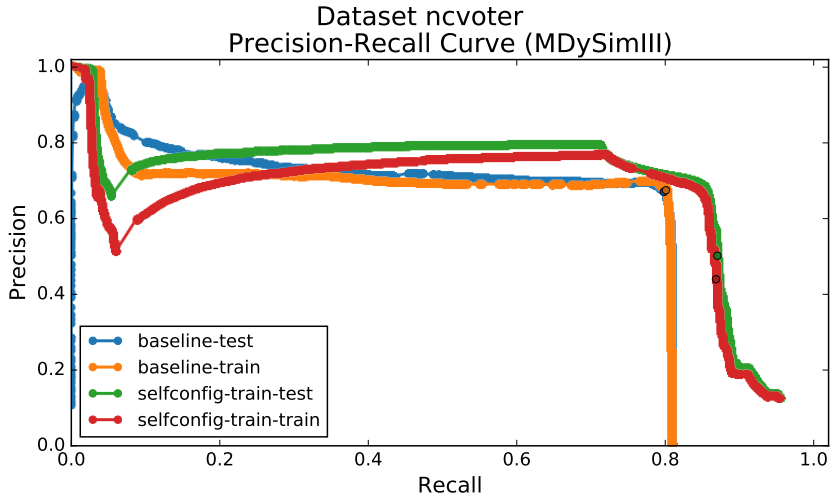

Abbildung 5.17 Precision-Recall Kurven der Baseline- und Selbstkonfigurationsdurchläufe, für Trainings- und Testdaten.

Konfiguration	PC	PQ	Recall	Precision	F-measure	Avg. Precision
Baseline (train)	0.808	0.004	0.801	0.675	0.733	0.581
Baseline (test)	0.808	0.004	0.789	0.671	0.729	0.600
Selbstkonf. (train)	0.956	0.127	0.868	0.440	0.584	0.648
Selbstkonf. (test)	0.956	0.127	0.870	0.502	0.637	0.695

Tabelle 5.10 Vergleich der Qualitätsmetriken für die Baseline und die Selbstkonfiguration.

dardwahrscheinlichkeitsschwelle, in der Precision dem Baseline Klassifikator deutlich unterlegen, wodurch die Baseline bessere F-measurewerte erzielt. An den Kurven der Selbstkonfiguration ist zu erkennen, dass ca. 85 % der Matches mit einer Precision von 65 % bis 70 % Prozent erkannt werden. Darüberhinaus fällt die Precision in kleinen Recallschritten stark ab, was darauf schließen lässt, dass der Klassifikator für diese Paare enorme Schwierigkeiten hat Matches von Non-Matches zu trennen. Die Kurve der Baseline dagegen ist einfacher. Hier können fast alle Matches mit einer Precision von ca. 60 % erkannt werden. Für die letzen Paar Prozent fällt die Kurve bis auf unter 1 % Precision. Daraus lässt sich ableiten, dass das Blocking Schema nur wenige herausfordernde Paare selektiert und die Standardwahrscheinlichkeitsschwelle deshalb fast Recall und Precision maximiert.

Bezüglich der Qualität der Ergenisse schneidet die Baseline recht gut ab und ohne Kalibrierung der Klassfifkatoren ist diese dem selbstkonfigurierendem System im F-measure über 10 % voraus. Als nächstes wird die Effizienz bzgl. Speicherverbrauch (Abbildung 5.18), Einfügeoperationen pro Sekunde (Abbildung 5.19) und Anfragen pro Sekunde (Abbildung 5.20) betrachtet. Die Einfügeoperationen pro Sekunde sind abhängig von der Länge des Blocking Schema und der Art der Prädikate. Die Baseline

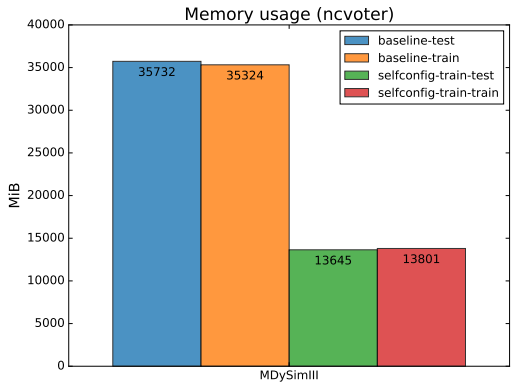

Abbildung 5.18 Vergleich des maximal benötigten Arbeitsspeichers zwischen Baseline und Selbstkonfiguration

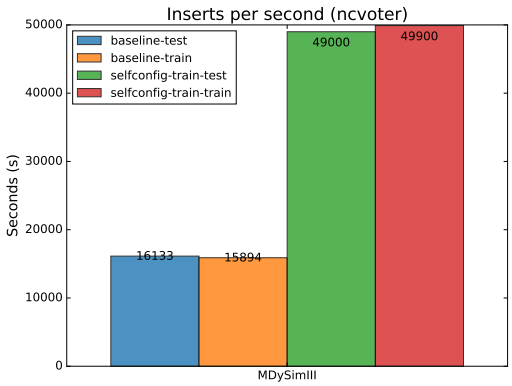

Abbildung 5.19 Vergleich der Einfügeoperationen pro Sekunde für Baseline und Selbstkonfiguration

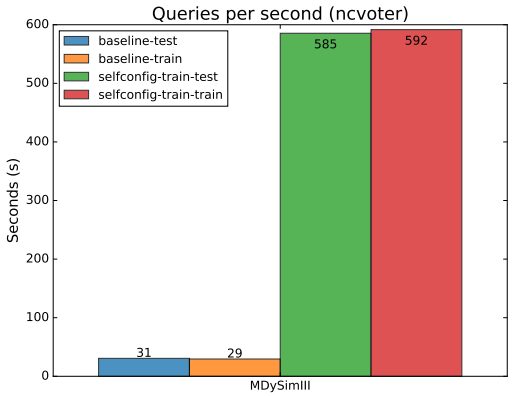

Abbildung 5.20 Vergleich der Anfragen pro Sekunde für Baseline und Selbstkonfiguration

ist fast 3-Mal so langsam, weil diese zum einen zwei spezifische Blockingprädikate mehr hat, wodurch mehr Blockschlüssel erzeugt werden und zum anderen ein Prädikat Token erzeugt, wohingegen die Selbstkonfiguration nur ID Prädikate besitzt. Der Speicherverbrauch der Baseline ist gegenüber der Selbstkonfiguration über das Doppelte. Dass die Baseline ein wenig mehr Arbeitsspeicher verbraucht war absehbar, da durch das längere Blocking Schema mehr Blockschlüssel erzeugt und damit mehr Blöcke gebaut werden. Der Grund, warum die Baseline mehr als doppelt soviel Arbeitsspeicher benötigt, liegt in der Pairs Quality verborgen. Diese ist nämlich um das 31-fache schlechter als die der Selbstkonfiguration. Damit gibt es nicht nur mehr Blöcke, sondern die Blöcke sind im Durchschnitt auch deutlich größer. Aufgrunddessen ist auch der Similarity Index deutlich größer, welcher quadratisch zur Anzahl der Elemente eines Blockes wächst. Größere Blöcke bedeuten auch mehr Kandidaten bei einer Anfrage, mehr Ähnlichkeitsberechnungen und mehr Klassifikationen. Deshalb sind die Anfragen pro Sekunde der Baseline fast 20-Mal weniger. Dieser Wert ist jedoch etwas pessimistisch, da ca. 3 GB Arbeitsspeicher mehr benötigt wurden als im Testsystem verfügbar sind und dadurch die Statistik durch Swapping verfälscht wurde. Nichtsdestotrotz schneidet die Baseline

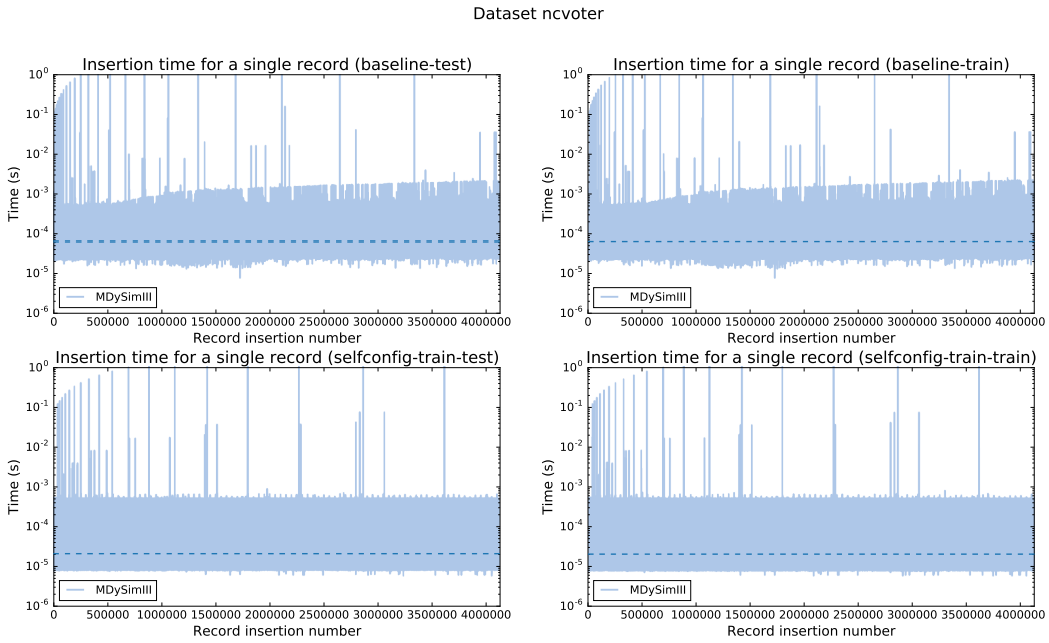

Abbildung 5.21 Vergleich der Einfügezeiten für alle 1.3 Mio. Anfragen zwischen Baseline und Selbstkonfiguration.

bezüglich der Effizienz ziemlich schlecht ab und eine Verwendung dieser Konfiguration in einem Event Stream Processing System ist dadurch fraglich.

Über die beiden vorliegenden Konfigurationen wird im Folgenden die Performanz des MDySimIII Verfahrens betrachtet. Dazu sind in den Abbildungen 5.21, 5.22 die Einfügezeiten in den Index für alle 4.2 Mio. Einträge des Base Datensatzes und die Anfragezeiten für alle 1.3 Mio. Einträge des Trainings- bzw. Testdatensatzes zu sehen. Für die Baseline ist dabei zu sehen, dass die Einfügezeiten mit der Anzahl der Datensätze im Index leicht wachsen. Der Grund dafür sind die großen Blöcke. Je größer die Blöcke werden, desto mehr Ähnlichkeitsberechnungen müssen beim Einfügen durchgeführt werden. Dahingegen bleiben die Einfügezeiten für die Selbstkonfiguration unverändert. Dies ist zum einen der Fall, weil die Blöcke wesentlich geringer sind, zudem besteht das Blocking Schema nur aus ID Prädikaten, sodass beim Einfügen keine Ähnlichkeitsberechnungen stattfinden, weil nur Datensätze mit gleichen Attributen zusammen gruppiert werden und diese offensichtlich eine Ähnlichkeit von 1.0 haben. Datensätze mit gleichen Attributen zusammen gruppiert werden und diese offensichtlich eine Ähnlichkeit von 1.0 haben.

Die Anfragezeiten für die Baseline wachsen ebenfalls leicht an. Das liegt zum einen an der wachsenden Einfügezeiten, da ein neuer Datensatz während einer Anfrage eingefügt wird und zum anderen an der dadurch wachsenden Anzahl von Kandidaten. Ein weiterer Grund ist, dass für die letzten 300k Anfragen das Betriebssystem begonnen hat den Arbeitsspeicher zu swappen. Dahingegen bleiben die Anfragezeiten bei der Selbstkonfi-

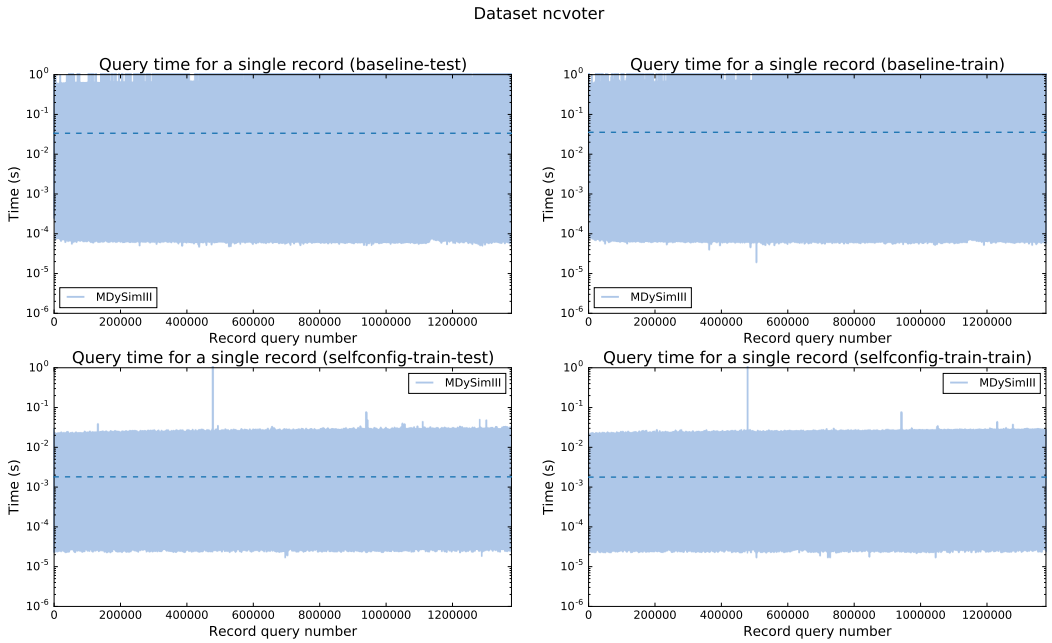

Abbildung 5.22 Vergleich der Anfragezeiten für alle 1.3 Mio. Anfragen zwischen Baseline und Selbstkonfiguration.

guration erneut gleich, weil die Einfügezeiten nicht wachsen und aufgrund der kleinen Blöcke die Menge der Kandidaten auch nicht größer wird.

5.7 Grund Truth vs No Ground Truth

Dieser Abschnitt vergleicht die Selbstkonfiguration mit einer synthetisierten Ground Truth, welcher die Matches bekannt sind (gt) und einer Selbstkonfiguration, welcher die Matches nicht bekannt sind (nogt). Der Aufbau und die Parameter sind dabei identisch zum vorherigen Abschnitt 5.6. Mit bekannten Matches ist das Blocking Schema identisch zum vorhierigen Abschnitt, ohne Matches unterscheidet sich das Blocking Schema in einer Position. Statt dem Ausdruck im zweiten Ausdruck wurde der Zweitname gegen den Bundestaat, bei gleichem Prädikat, getauscht.

Selbstkonf. mit Matches:

$$((\text{Vorname}, \text{ID}) \wedge (\text{Nachname}, \text{ID})) \quad \vee$$
$$((\text{Zweitname}, \text{ID}) \wedge (\text{Strasse}, \text{ID}))$$

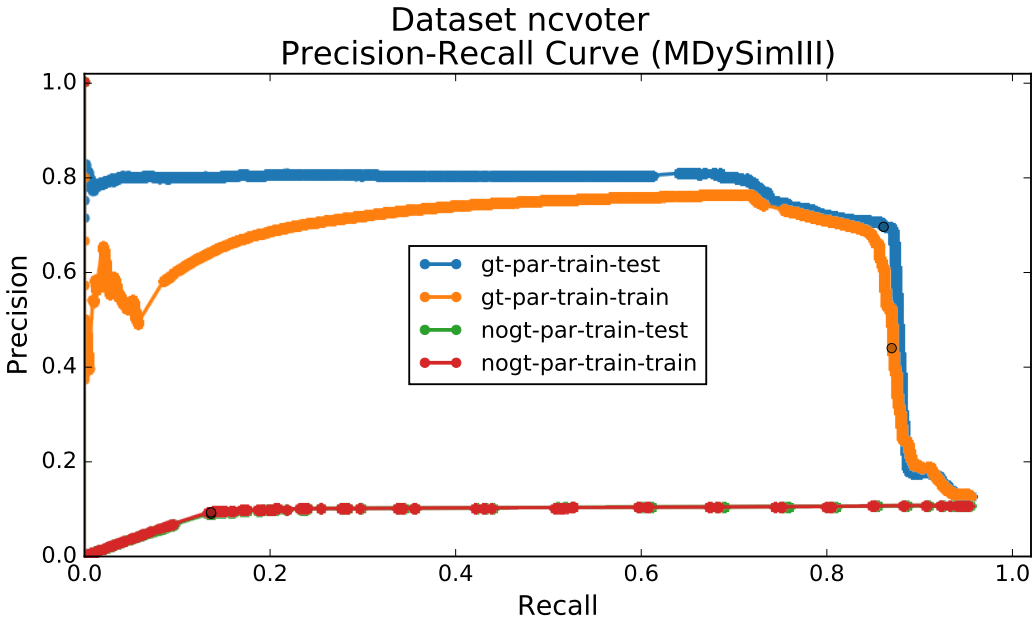

Abbildung 5.23 Precision-Recall Kurven für die Selbstkonfigurationen mit (gt) und ohne (nogt) bekannte Matches für Trainings- und Testdaten.

Selbstkonf. ohne Matches:

$$((\text{Vorname}, \text{ID}) \wedge (\text{Nachname}, \text{ID})) \quad \vee$$
$$((\text{Bundesstaat}, \text{ID}) \wedge (\text{Strasse}, \text{ID}))$$

Die sehr ähnlichen Blocking Schemata resultieren in sehr ähnlichen Ergebnissen für Pairs Completeness, welche mit 95.6 % identisch ist und Pairs Quality, die ohne Matches mit 10.8 % lediglich 2 % schlechter ist, als mit Matches (Tabelle 5.11). Diese Werte sind hinsichtlich der deutlich schlechteren Ground Truth erstaunlich (vgl. Abbildung 5.12). Allerdings endet hier der Erfolg der automatisch erzeugten Ground Truth. Nachdem diese durch das Blocking Schema gefiltert wurde, stehen 300k Matches nur 3k Non-Matches gegenüber, die aufgrund der Ähnlichkeitsschwelle von 0.3 nicht sonderlich herausfordernd für einen Klassifikator sind. Diese gefilterte Ground Truth wird im Verhältnis 1:3, zu Gunsten der Non-Matches, für den Klassifikator gesubsampelt. Bekannt ist, dass die tatsächlichen Matches lediglich 50k betragen, deshalb ist die Wahrscheinlichkeit, dass die gesubsampelte Ground Truth eine gute Repräsentation des Datensatzes ist, relativ gering. Dementsprechend sind die Resultate für Recall und Precision extrem schlecht. In der Precision-Recall Kurve ist gut zu sehen, dass auch durch eine Kalibrierung der Wahrscheinlichkeitsschwelle des Klassifikators in keinem Punkt eine Verbesserung der Precision möglich ist. Die Kurven mit Matches sind ähnlich zu den Selbstkonfiguration aus Abschnitt 5.6. Während die train-train Kurve nahzu identisch ist, unterscheidet sich die train-test Kurve leicht. Der Grund dafür ist zum einen, dass für den Fusion-Lerner

Konfiguration	PC	PQ	Recall	Precision	F-measure	Avg. Precision
Selbstkonf. GT (train)	0.956	0.127	0.870	0.440	0.585	0.631
Selbstkonf. GT (test)	0.956	0.127	0.861	0.696	0.770	0.709
Selbstkonf. NoGT (train)	0.956	0.108	0.136	0.093	0.111	0.095
Selbstkonf. NoGT (test)	0.956	0.108	0.136	0.090	0.109	0.095

Tabelle 5.11 Vergleich der Qualitätsmetriken für die Selbstkonfiguration mit (GT) und ohne (NoGT) bekannte Matches.

die gesubsamplete Ground Truth eine andere ist, da die Paare zwar nach Ähnlichkeit, aber trotzdem per Zufall gezogen werden und zum anderen, dass die Varianz der Bewertungen, des Fusion-Lerner für die verschiedenen SVM Parameter, unter einem Prozent ist. Daher kann selbst eine geringfügig verschiedene Ground Truth zu komplett anderen Parametern führen. Für die train-test Daten wurde dadurch eine SVM mit RBF-Kernel und einem C von 10 trainiert. Die zugehörige Kurve zeigt, dass dieser C Parameter besser funktioniert und auch die Standardwahrscheinlichkeitsschwelle befindet sich jetzt fast im Maximum. Aus diesen Erkenntnissen lässt sich schließen, dass das Verfahren zum Subsampling für die Ground Truth ohne bekannte Matches ungeeignet ist und dass das F-measure als Maß für den Fusion-Lerner zur Parameterbestimmung nicht gut funktioniert.

Die Effizienz ist in beiden Konfigurationen fast identisch, was allerdings nicht verwunderlich ist, da das Blocking Schema ähnlich ist, die Prädikatsfunktionen dieselben sind und auch die Pairs Quality keine große Abweichung zeigt.

5.8 Datensatzvergleich

In diesem Abschnitt wird evaluiert, wie gut die Selbstkonfiguration für andere Datensätze als den NCVoter funktioniert. Dabei findet die Build-Phase und die Query-Phase jeweils auf der Hälfte der Daten statt. In Abbildung 5.24 sind die Precision-Recall Kurven für die Datensätze dargestellt. Eindeutiger Gewinner ist der Restaurantdatensatz, welcher alle Duplikate findet und bei einer Precision von 98 % fast keinen Fehler macht. Auf den Ferbl-Datensatzen funktioniert die Selbstkonfiguration ebenfalls gut, dennoch können ca. 20 % der Duplikate nicht gefunden werden, da diese auf Rechtschreibfehlern basieren, die durch die Prädikate ID und Token nicht erfasst werden. Die Publikationsdatensätze (DBLP-ACM und DBLP-Scholar) erreichen auch gute Werte, wobei der DBLP-ACM Datensatz wie erwartet besser abschneidet. Für den als besonders schwer geltenden CORA Publikationsdatensatz, können mit 75 % Recall zwar noch relativ viele Duplikate gefunden werden, allerdings ist es dem Klassifikator nicht möglich die Ergebnismenge sinnvoll einzugrenzen. Der Grund dafür ist, dass bei Duplikaten die Attribute

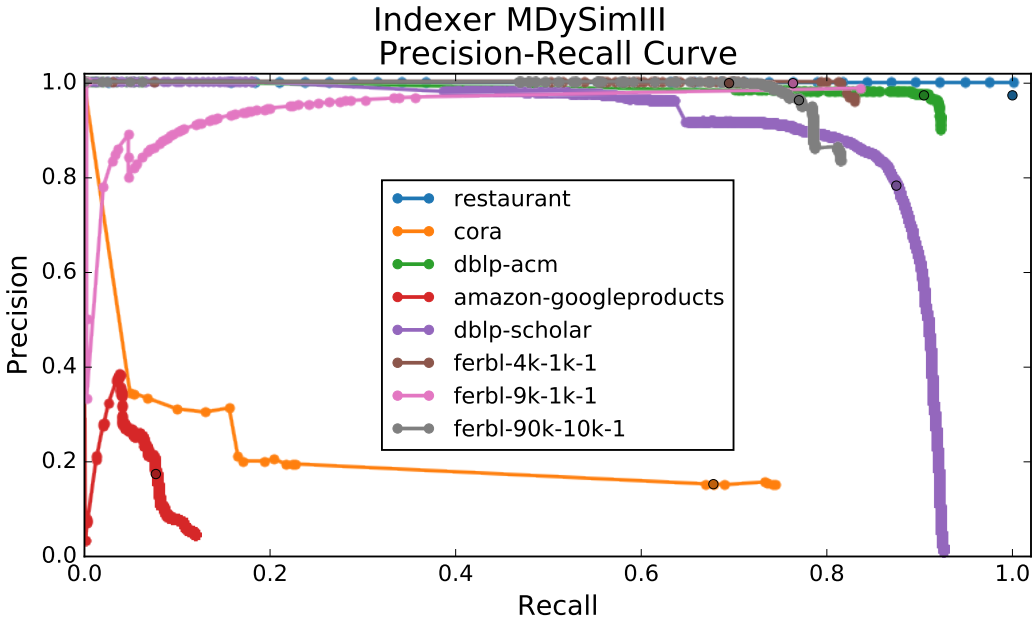

Abbildung 5.24 Precision-Recall Kurven der Selbstkonfiguration auf allen Datensätzen im Vergleich

häufig nicht übereinstimmen, beispielsweise ist der Titel für den einen Datensatz im Attribut `Titel` und für den anderen Datensatz im Attribut `Buchtitel`, sodass deren Titelähnlichkeit 0 beträgt. Dadurch gibt es weniger Merkmale für den Klassifikator und folglich treten beim Klassifizieren dieselben Schwierigkeiten auf, wie bei den Varvektoren. Komplett versagt hat die Selbstkonfiguration auf dem Amazon-GoogleProdukts Datensatz. Das liegt vor allem daran, dass die Attribute aus vielen Token bestehen. Durch die Token werden jedoch viele Blöcke erzeugt, was die Pairs Qualtity extrem senkt. Das gewählte Blocking Schema mit einer Pairs Completeness von 11 % und einer Pairs Quality von 22 % ist dadurch im F-Measure besser als die übrigen Kandidaten, die zwar einen besseren Recall haben, allerdings ist die Pairs Quality in allen Fällen unter 0.01 %.

6 Zusammenfassung und Ausblick

Dieses Kapitel gibt eine Zusammenfassung der Hauptbeiträge dieser Arbeit und einen Ausblick in welche Richtungen diese weiterentwickelt werden können.

6.1 Zusammenfassung

Die vorliegende Arbeit untersucht das Problem der Selbstkonfiguration eines Entity Resolution Workflows für Event Stream Processing Systeme. In Kapitel 2 wurden dazu die drei Anforderungen **Niedrige Latenzen**, **Datenmodifikation zur Laufzeit** und **Hohe Trefferrate** festgelegt. Diese Anforderungen können jedoch nur durch einen Kompromiss zwischen Qualität und Effizienz erfüllt werden. Hauptproblem bei der Verwendung eines Entity Resolution Systems ist die Anpassung der freien Parameter auf die Domäne der Daten. Die drei entscheidenden Parameterkonfigurationen dabei sind das Blocking Schema, die Ähnlichkeitsmetriken und die Klassifikation, für welche in Kapitel 3 Verfahren analysiert und entwickelt wurden, die eine automatische Bestimmung der Parameter ermöglichen. Die Grundlage dieser Verfahren bilden gelabelte Daten, die vorklassifizierte Duplikatspaare und nicht Duplikatspaare beinhalten. Für viele Datensätze sind jedoch keine gelabelten Daten verfügbar und das manuelle Erzeugen ist sowohl zeit- als auch kostenintensiv. Aufgrunddessen wurde ein Verfahren entwickelt, dass automatisch gelabelte Daten synthetisiert, damit es möglich ist, dass das System sich unüberwacht auf einer neuen Domäne selbst adaptiert. Des Weiteren wurde ein Blocking Verfahren weiterentwickelt, dass mit geeigneten Parametern, alle drei Anforderungen erfüllt. In Kapitel 4 wurden die analysierten und entwickelten Verfahren zu einem Gesamtsystem zusammengefügt, dass sich dem Anwender gegenüber transparent selbstkonfiguriert und anhand der Konfiguration und der Bestandsdaten, Anfragen aus einem Eventstrom beantwortet. Die Evaluation in Kapitel 5 untersuchte zunächst die noch offenen (nicht selbstkonfigurierbaren) freien Parameter und wählte robuste Werte für diese aus. Die Qualität und Effizienz des Systems wurde danach gegenüber einer manuell bestimmten Baseline überprüft. Dabei zeigte sich, dass es mit einigen praktischen Hinweisen in angemessener Zeit möglich ist, eine Konfiguration zu bestimmen, die qualitativ gute Ergebnisse ermittelt. Bezüglich der Anforderungen an die Effizienz versagt die Baseline jedoch komplett. Diese ist maßgeblich vom Blocking Schema abhängig, allerdings werden für dieses nur Qualitätsparameter konfiguriert, die implizit Auswirkungen auf die Effizienz haben. Daher ist es ohne bzw. mit nur geringer Kenntnis über die Funktionsweise des

Blocking-Verfahren, nur schwer möglich abzuschätzen, wie sich eine Konfiguration auf die Effizienz auswirkt. Die Selbstkonfiguration kann dies deutlich besser beurteilen und erreicht deshalb ca. 20-Mal bessere Effizienzwerte, zudem können gegenüber der Baseline theoretisch 15 % mehr Duplikate gefunden werden. Während das selbstkonfigurierte Blocking Schema in der Evaluation überzeugte, erwies sich die Konfiguration des Klassifikationsmodells als wenig stabil, sodass unterschiedliche Konfigurationen auf denselben Ausgangsdaten, z. T. bis zu 40 % in der Precision variieren. Abschließend wurde die Konfiguration aus den automatisch synthetisierten gelabelten Daten evaluiert. Gegenüber den tatsächlichen gelabelten Daten, sind diese qualitativ fragwürdig, dennoch wird ein Blocking Schema gelernt, dass in Qualität und Effizienz fast ebenbürtig ist. Zum Trainieren des Klassifikationsmodells sind die gelabelten Daten jedoch unbrauchbar, weshalb keine sinnvolle Klassifikation der Kandidatenmenge durchführt werden kann.

6.2 Ausblick

Die in dieser Arbeit vorgestellten und entwickelten Verfahren und Algorithmen haben eine Reihe von Beiträgen und Verbesserungen der aktuellsten Entity Resolution Methoden vorgenommen. Angepasst wurde das Blocking-Schema Lernverfahren, für die Anforderungen an ESP-Systeme und das DySimII Blocking-Verfahren, für das neue Blocking-Schema. Des Weiteren wurde im DySimII-Verfahren eine essentielle Schwäche korrigiert. Entwickelt wurde ein Lernverfahren zur Auswahl von Ähnlichkeitsfunktionen und ein Verfahren zur Bestimmung der Parameter eines Klassifikators. Zudem wurde an der Synthetisierung von Ground Truth Paaren gearbeitet. Alle Beiträge bieten jedoch noch einige Möglichkeiten der Weiterentwicklung und Probleme, die es zu lösen gilt, welche im Folgenden betrachtet werden.

Selbstkonfiguration bzgl. einer Mindestqualität oder -effizienz: Die aktuelle Selbstkonfiguration versucht stets einen Kompromiss zwischen Qualität und Effizienz einzugehen, ohne dabei einer Mindestanforderung an Qualität oder Effizienz genüge zu tun. In der Evaluation sind aus Effizienzgründen, beispielsweise ein Großteil der Blockingprädikate ausgeschlossen worden, womit das System einen Durchsatz von mindestens 500 Anfragen pro Sekunde erreicht hat. Für ein System, dass maximal 10 Anfragen pro Sekunde beantworten muss, wäre ein Teil dieser Prädikate durchaus noch infrage gekommen, wodurch die Qualität des Ergebnisses hätte verbessert werden können. Anhand solcher Vorgaben kann versucht werden, weitere freie Parameter, automatisch konfigurierbar zu machen.

Optimierung der Disjunktion des Blocking-Schema: Beim Lernen des Blocking Schema, werden nach der Bewertung eines Ausdrucks, dessen Datensatzpaare auf die Ground Truth abgebildet. Damit bei der Disjunktion der Ausdrücke diese vergleichbar sind und

die Blöcke nicht erneut gebaut werden müssen, um eine Disjunktion zu bewerten. Dabei werden vor allen Dingen die Non-Matches unterrepräsentiert, da nur eine kleine Auswahl derer Teil der Ground Truth ist. Über das Bauen der eigentlichen Blöcke kann daher deutlich genauer das F-Measure der Disjunktion ermittelt werden. Allerdings dauert dieser Vorgang auch deutlich länger, weil Blockschlüssel und Blöcke aus mehreren Attributen erzeugt werden. Eine Idee dieses Problem zu lösen ist ein Branch-and-Bound-Verfahren zu nutzen, dass anhand einer Heuristik, beispielsweise dem maximal erreichbaren F-Measure, nur sinnvolle Disjunktionen testet. Eine optimistische Heuristik ist beispielsweise, dass sich die Recallwerte der einzelnen Ausdrücke addieren und die Precision sich dabei mittelt.

Parallelisierung des Lernens: In der Selbstkonfigurationsphase dauert das Lernen des Blocking Schema am längsten, da für jeden zu prüfenden Ausdruck der Indexer alle Blöcke bauen muss. Dabei können die Ausdrücke unabhängig voneinander bewertet werden, was eine reibungslose Parallelisierung ermöglicht. Dieser Prozess benötigt jedoch z.T. sehr viel Arbeitsspeicher, weshalb Multithreading bzw. Multiprocessing keine Optionen sind. Denkbar ist aber die Ausführung auf einem Cluster von Rechnern. Die hierbei genutzen Daten sind statisch, deshalb kann dazu ein Batchverfahren wie MapReduce eingesetzt werden.

Parallelisieren der Anfragen: Zur Zeit werden die Anfragen auf einem CPU Kern von einem Thread bearbeitet. Durch die Parallelisierung kann hier die Effizienz deutlich gesteigert werden, vor allem da die Ähnlichkeitsberechnung, aufgrund der Teilvektoren, nicht vollständig vorausberechnet wird. Eine Idee ist, die Blöcke, die durch das Blocking-Verfahrens gebildet werden, auf mehrere Threads, Prozesse oder Rechner zu verteilen. Dadurch können die Ähnlichkeiten verschiedener Attribute gleichzeitig berechnet bzw. abgerufen werden. Damit allerdings kein Knoten zum Flaschenhals wird, benötigt es einen Algorithmus, der die Blöcke nach Bearbeitungszeit und Anfragehäufigkeit verteilt und eventuell nach Last anpasst. Einen Ansatz hierfür beschreibt Kolb in [3]. Dabei ist der Algorithmus für MapReduce-Verfahren optimiert, liefert jedoch einen guten Einstieg in die Problematik. Ein interessanter Effekt des Verteilens der Blöcke auf mehrere Rechner ist, dass damit auch Datensätze verarbeitet werden können, die deutlich mehr Arbeitsspeicher benötigen als technisch auf einem einzelnen Rechner möglich sind.

Verbessern des Bewertungsmaß des Similarity Lerners: Der Similarity Lerner bewertet Ähnlichkeitsmetriken anhand der Average Precision. Hat dieser genügend Daten zum Lernen und einen großen Pool von Ähnlichkeitsmetriken zur Auswahl, werden gute Metriken gelernt, mit welcher ein Klassifikator qualitativ gute Entscheidungen treffen kann. Allerdings sind die Average Precision Werte der Metriken pro Attribut oft im Bereich 10^{-1}, was vor allem bei kleineren Datensätzen dazu führt, dass die Auswahl der Ähnlichkeitsmetriken suboptimal ist. Hier gilt es herauszufinden, wie die Qualtität der Ähnlichkeitsmetriken besser differenziert werden kann, um robustere Entscheidungen zu treffen.

Lernen von Hyperparametern der Ähnlichkeitsmaße: Der Similarity Lerner wählt aktuell aus einer Menge von vorkonfigurierten Ähnlichkeitsmaßen aus. Dabei ist aus Abschnitt 2.4 bekannt, dass Ähnlichkeitsmaße z.T. mehrere Parameter haben, die je nach Datendomäne entscheidenden Einfluss auf die Qualität haben können. Ähnlich zum der Fusion-Lerner, könnte der Similarity Lerner ebenfalls ein Parametergrid für eine Ähnlichkeitsfunktion abtestet. Dazu muss allerdings sichergestellt sein, dass für jeden Parameterwert stets die Dreiecksungleichung erfüllt ist.

Verbessern des Bewertungsmaß des Fusion-Lerners: Aktuell wählt der Fusion-Lerner die besten Parameter für einen Klassifikationsmodell aus, indem dieses über das F-Measure bewertet wird. Die Evaluation hat jedoch gezeigt, dass für viele der unterschiedlichen Parametereinstellungen sehr ähnliche Werte berechnet werden, die sich je nach gesubsampelter Ground Truth zu Gunsten der einen oder anderen Parameter ändern. Dementsprechend variiert die Qualität der Klassifikation teilweise stark. Auch hier gilt es noch herauszufinden, wie die Parameter besser bewertet werden können, sodass die Auswahl auch auf unterschiedlich gesubsampelten Ground Truths robustere Ergebnisse erzielt.

Kalibrierung der Wahrscheinlichkeitsschwellen: Die aktuell gewählten Wahrscheinlichkeitsschwellen der Klassifikatoren entsprechen den Standardeinstellungen der Scikit-learn Bibliothek. Die Evaluation hat gezeigt, dass durch eine Kalibrierung in fast allen Fällen die Precision bei minimalem Recallverlust dramatisch verbessert werden kann.

Aktives Lernen der Ground Truth: Das Verfahren von Kejriwal & Miranker aus [17] kann zwar ein schwache Ground Truth generieren, diese trennt jedoch Matches von Non-Matches durch eine harte Ähnlichkeitsschwelle, was nicht den realen Daten entspricht. Ein anderer Ansatz möglichst effizient eine Ground Truth zu Erzeugen, sind aktive Lernmethoden (vgl. Abschnitt 2.5.3), welche selbständig eine kleine Menge herausfordernder, manuell zu klassifizierende, Datenpaare bestimmen und so sequentielle eine Ground Truth synthetisieren. Eine interessante Weiterentwicklung ist es, diese beiden Ansätze zu kombinieren und die automatisch erzeugte schwache Ground Truth sequentielle, durch Auswahl herausfordernder Paare, zu verbessern und die harte Ähnlichkeitsschwelle zu lockern.

Literaturverzeichnis

[1] Fellegi, Ivan P. ; Sunter, Alan B.: A Theory for Record Linkage. In: *Journal of the American Statistical Association* Bd. 64 (1969), Nr. 328, S. 1183–1210

[2] Köpcke, Hanna ; Rahm, Erhard: Frameworks for Entity Matching: A Comparison. In: *Data & Knowledge Engineering* Bd. 69 (2010), Nr. 2, S. 197–210

[3] Kolb, Lars: *Effiziente MapReduce-Parallelisierung von Entity Resolution-Workflows*, University of Leipzig, Dissertation, 2014

[4] Kolb, Lars ; Rahm, Erhard: Parallel Entity Resolution with Dedoop. In: *Datenbank-Spektrum* Bd. 13 (2013), Nr. 1, S. 23–32

[5] Malhotra, Pankaj ; Agarwal, Puneet ; Shroff, Gautam: Graph-Parallel Entity Resolution Using LSH & IMM. In: *EDBT/ICDT Workshops*, 2014, S. 41–49

[6] Whang, S. E. ; Marmaros, D. ; Garcia-Molina, H.: Pay-As-You-Go Entity Resolution. In: *IEEE Transactions on Knowledge and Data Engineering* Bd. 25 (2013), Nr. 5, S. 1111–1124

[7] Ramadan, Banda ; Christen, Peter ; Liang, Huizhi ; Gayler, Ross W.: Dynamic Sorted Neighborhood Indexing for Real-Time Entity Resolution. In: *J. Data and Information Quality* Bd. 6 (2015), Nr. 4, S. 15:1–15:29

[8] Christen, Peter ; Gayler, Ross: Towards Scalable Real-Time Entity Resolution Using a S Imilarity-Aware Inverted Index Approach. In: *Proceedings of the 7th Australasian Data Mining Conference - Volume 87*, *AusDM '08*. Darlinghurst, Australia, Australia : Australian Computer Society, Inc., 2008 — ISBN 978-1-920682-68-2, S. 51–60

[9] Köpcke, Hanna ; Thor, Andreas ; Rahm, Erhard: Evaluation of Entity Resolution Approaches on Real-World Match Problems. In: *Proceedings of the VLDB Endowment* Bd. 3 (2010), Nr. 1-2, S. 484–493

[10] Draisbach, Uwe ; Naumann, Felix: A Comparison and Generalization of Blocking and Windowing Algorithms for Duplicate Detection. In: *Proceedings of the International Workshop on Quality in Databases (QDB)*, 2009, S. 51–56

[11] Christen, P.: A Survey of Indexing Techniques for Scalable Record Linkage and Deduplication. In: *IEEE Transactions on Knowledge and Data Engineering* Bd. 24 (2012), Nr. 9, S. 1537–1555

[12] Aizawa, A. ; Oyama, K.: A Fast Linkage Detection Scheme for Multi-Source Information Integration. In: *International Workshop on Challenges in Web Information Retrieval and Integration*, 2005, S. 30–39

[13] Hernández, Mauricio A. ; Stolfo, Salvatore J.: The Merge/Purge Problem for Large Databases. In: *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data, SIGMOD '95*. New York, NY, USA : ACM, 1995 — ISBN 978-0-89791-731-5, S. 127–138

[14] Cohen, William W.; Richman, Jacob: Learning to Match and Cluster Large High-Dimensional Data Sets for Data Integration. In: *Proceedings of the Eighth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, *KDD '02*. New York, NY, USA : ACM, 2002 — ISBN 978-1-58113-567-1, S. 475–480

[15] Ramadan, Banda; Christen, Peter; Liang, Huizhi; Gayler, Ross W.; Hawking, David: Dynamic Similarity-Aware Inverted Indexing for Real-Time Entity Resolution. In: *Pacific-Asia Conference on Knowledge Discovery and Data Mining* : Springer, 2013, S. 47–58

[16] Li, Shouheng; Liang, H.; Ramadan, Banda; others: *Two Stage Similarityaware Indexing for Large Scale Realtime Entity Resolution* : AusDM, 2013 – Maßstab

[17] Kejriwal, M.; Miranker, D. P.: An Unsupervised Algorithm for Learning Blocking Schemes. In: *2013 IEEE 13th International Conference on Data Mining*, 2013, S. 340–349

[18] Gu, Quanquan; Li, Zhenhui; Han, Jiawei: Generalized Fisher Score for Feature Selection. In: *arXiv preprint arXiv:1202.3725* (2012)

[19] Elmagarmid, A. K.; Ipeirotis, P. G.; Verykios, V. S.: Duplicate Record Detection: A Survey. In: *IEEE Transactions on Knowledge and Data Engineering* Bd. 19 (2007), Nr. 1, S. 1–16

[20] Levenshtein, Vladimir I.: Binary Codes Capable of Correcting Deletions, Insertions, and Reversals. In: *Soviet Physics Doklady*. Bd. 10, 1966, S. 707–710

[21] Damerau, Fred J.: A Technique for Computer Detection and Correction of Spelling Errors. In: *Communications of the ACM* Bd. 7 (1964), Nr. 3, S. 171–176

[22] Needleman, Saul B.; Wunsch, Christian D.: A General Method Applicable to the Search for Similarities in the Amino Acid Sequence of Two Proteins. In: *Journal of Molecular Biology* Bd. 48 (1970), Nr. 3, S. 443–453

[23] Waterman, Michael S.; Smith, Temple F.; Beyer, William A.: Some Biological Sequence Metrics. In: *Advances in Mathematics* Bd. 20 (1976), Nr. 3, S. 367–387

[24] Smith, T. F.; Waterman, M. S.: Identification of Common Molecular Subsequences. In: *Journal of Molecular Biology* Bd. 147 (1981), Nr. 1, S. 195–197

[25] Monge, Alvaro E.; Elkan, Charles; others: The Field Matching Problem: Algorithms and Applications. In: *KDD*, 1996, S. 267–270

[26] Cohen, William W.: WHIRL: A Word-Based Information Representation Language. In: *Artificial Intelligence* Bd. 118 (2000), Nr. 12, S. 163–196

[27] Gravano, Luis; Ipeirotis, Panagiotis G.; Koudas, Nick; Srivastava, Divesh: Text Joins in an RDBMS for Web Data Integration. In: *Proceedings of the 12th International Conference on World Wide Web*, *WWW '03*. New York, NY, USA : ACM, 2003 — ISBN 978-1-58113-680-7, S. 90–101

[28] Sonnenburg, Sören; Rätsch, Gunnar; Rieck, Konrad: Large Scale Learning with String Kernels. In: *Large Scale Kernel Machines* (2007), S. 73–103

[29] Lodhi, Huma; Saunders, Craig; Shawe-Taylor, John; Cristianini, Nello; Watkins, Chris: Text Classification Using String Kernels. In: *Journal of Machine Learning Research* Bd. 2 (2002), Nr. Feb, S. 419–444

[30] Christen, Peter: *Data Matching: Concepts and Techniques for Record Linkage, Entity Resolution, and Duplicate Detection* : Springer Science & Business Media, 2012 – Maßstab

[31] Boser, Bernhard E. ; Guyon, Isabelle M. ; Vapnik, Vladimir N.: A Training Algorithm for Optimal Margin Classifiers. In: *Proceedings of the Fifth Annual Workshop on Computational Learning Theory, COLT '92*. New York, NY, USA : ACM, 1992 — ISBN 978-0-89791-497-0, S. 144–152

[32] Bilenko, Mikhail ; Mooney, Raymond J.: Adaptive Duplicate Detection Using Learnable String Similarity Measures. In: *Proceedings of the Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD '03*. New York, NY, USA : ACM, 2003 — ISBN 978-1-58113-737-8, S. 39–48

[33] Christen, Peter: Automatic Record Linkage Using Seeded Nearest Neighbour and Support Vector Machine Classification. In: *Proceedings of the 14th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* : ACM, 2008, S. 151–159

[34] Arasu, Arvind ; Götz, Michaela ; Kaushik, Raghav: On Active Learning of Record Matching Packages. In: *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data* : ACM, 2010, S. 783–794

[35] Vogel, Tobias ; Heise, Arvid ; Draisbach, Uwe ; Lange, Dustin ; Naumann, Felix: Reach for Gold: An Annealing Standard to Evaluate Duplicate Detection Results. In: *Journal of Data and Information Quality* Bd. 5 (2014), Nr. 1-2, S. 1–25

[36] Ramadan, Banda ; Christen, Peter: Unsupervised Blocking Key Selection for Real-Time Entity Resolution. In: Cao, T. ; Lim, E.-P. ; Zhou, Z.-H. ; Ho, T.-B. ; Cheung, D. ; Motoda, H. (Hrsg.): *Advances in Knowledge Discovery and Data Mining*. Bd. 9078. Cham : Springer International Publishing, 2015 — ISBN 978-3-319-18031-1 978-3-319-18032-8, S. 574–585

[37] Bartolini, Ilaria ; Ciaccia, Paolo ; Patella, Marco: String Matching with Metric Trees Using an Approximate Distance. In: *String Processing and Information Retrieval* : Springer, Berlin, Heidelberg, 2002, S. 271–283

[38] Cilibrasi, Rudi ; Vitányi, Paul MB: Clustering by Compression. In: *IEEE Transactions on Information theory* Bd. 51 (2005), Nr. 4, S. 1523–1545

[39] Hamming, Richard W.: Error Detecting and Error Correcting Codes. In: *Bell Labs Technical Journal* Bd. 29 (1950), Nr. 2, S. 147–160

[40] Rieck, Konrad ; Wressnegger, Christian: Harry: A Tool for Measuring String Similarity. In: *J. Mach. Learn. Res.* Bd. 17 (2016), Nr. 1, S. 258–262

[41] Christen, Peter: Preparation of a Real Temporal Voter Data Set for Record Linkage and Duplicate Detection Research (2013)

[42] Christen, Peter: Febrl - an Open Source Data Cleaning, Deduplication and Record Linkage System with a Graphical User Interface. In: *Proceedings of the 14th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* : ACM, 2008, S. 1065–1068

[43] Pedregosa, F. ; Varoquaux, G. ; Gramfort, A. ; Michel, V. ; Thirion, B. ; Grisel, O. ; Blondel, M. ; Prettenhofer, P. ; Weiss, R. ; u. a.: Scikit-Learn: Machine Learning in Python. In: *Journal of Machine Learning Research* Bd. 12 (2011), S. 2825–2830

Abbildungsverzeichnis

Tabellenverzeichnis

Erklärung

Erklärung gem. ABPO, Ziff. 6.4.3

Ich versichere, dass ich die Master-Thesis selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel benutzt habe.

Wiesbaden, 21.04.2017

Kevin Sapper

Hiermit erkläre ich mein Einverständnis mit den im Folgenden aufgeführten Verbreitungsformen dieser Master-Thesis:

Verbreitungsform	**ja**	**nein**
Einstellung der Arbeit in die Bibliothek der Hochschule RheinMain	√	
Veröffentlichung des Titels der Arbeit im Internet	√	
Veröffentlichung der Arbeit im Internet	√	

Wiesbaden, 21.04.2017

Kevin Sapper