Ein Modell für die praktische Anwendung von *Link Prediction* auf Knowledge Graphen

Tobias Hübenthal
5640520

Masterarbeit
Wirtschaftsmathematik (M.Sc.)

Department Mathematik/Informatik
Mathematisch-Naturwissenschaftliche Fakultät
Universität zu Köln
Oktober 2021

Erstgutachter
Prof. Dr. Hubert Randerath

Zweitgutachter
Dr. Jens Dörpinghaus

Inhaltsverzeichnis

1 Einleitung **1**

1.1 Motivation 1

1.2 Aufbau der Arbeit 2

2 Grundlagen **4**

2.1 Graphentheorie 4

2.2 Datenbanken 6

2.2.1 Relationale Datenbanken 7

2.2.2 NoSQL- und Graphdatenbanken 8

2.3 Komplexitätstheorie und Laufzeitanalyse 10

2.4 Bildgebende Verfahren in der Medizin 11

2.4.1 Allgemeines 11

2.4.2 *DICOM* 12

2.5 *Link Prediction* 13

2.5.1 Einführung in das maschinelle Lernen 13

2.5.2 *Gradient Descent*, Newton und BFGS 15

2.5.3 *Scores* auf der Basis der Topologie des Graphen 16

2.5.4 *Link Prediction* für Pfade anhand von Knotenattributen 18

3 Praktische Umsetzung **24**

3.1 *Importer* für klinische Daten aus bildgebenden Verfahren 24

3.1.1 Aufbau der *DICOM*-Dateien 24

3.1.2 Benutzerspezifische Konfiguration 26

3.1.3 Datenschema 29

3.1.4 Datenimport in *Neo4j* 35

3.1.5 Kantenwiederholungen und zweiter *Importer* 36

3.1.6 Proof of Concept 38

3.2 *Link Prediction* mit *Neo4j* und *CRFs* 40

3.2.1 Ein-Knoten-Pfade 41

3.2.2 Mehr-Knoten-Pfade 44

4 Evaluation **51**
4.1 Laufzeitanalyse für den *Importer* 51
4.2 Auswertung der *Link Prediction* 57
4.2.1 Laufzeitanalyse 57
4.2.2 Güte der Ergebnisse 61
4.2.3 Fazit 69

5 Ausblick **71**

Abbildungsverzeichnis

2.1 Beispielhafte Darstellung zweier generischer Knoten in *Neo4j*, die durch eine Kante verbunden sind. 10
2.2 Darstellung des Prozesses der Knowledge Discovery. 14
2.3 Beispiel für ein Hidden-Markov-Modell: Z_t beschreibt den Zustand und X_t die davon abhängige Beobachtung zum Zeitpunkt t. 20
2.4 Entwicklung des *F1-Scores* in Abhängigkeit der Entwicklung von Präzision und *Recall*. 23

3.1 Struktur der *DICOM*-Daten. Der Patient stellt die oberste Stufe (grün) dar. Von dort aus verzweigt sich der Baum bis zur Ebene der eigentlichen Bilder (rot). 25
3.2 Datenschema für den Import der *DICOM*-Files. 30
3.3 Teilausschnitt des Graphen: Beispielhaftes Dreieck im Graphen zwischen den genannten Beispielknoten `Manufacturer` (rot), `General Equipment` (blau) und `General Series` (orange). 31
3.4 Ein Beispiel für die Knoten und Kanten in der Graphdatenbank nach abgeschlossenem Import: Es werden ein `General Image`-Knoten und seine direkte Nachbarschaft dargestellt. 37
3.5 Kantenwiederholungen in der ersten Version des *Importers*: Vom Patienten *(orange)* führen mehrere hundert Relationen zur Studie *(rot)* und von dort wiederum mehrere hundert Relationen zur untergeordneten Serie *(grau)*, welche dann ebenfalls mehrere hundert Bilder *(grün)* enthält, von denen jedoch durch die verwendete Query nur eines zurückgegeben wurde. 38
3.6 Resultat der zweiten Version des *Importers*: Vom Patienten *(orange)* führt eine Relation zur Studie *(rot)* und von dort wiederum eine Relation zur untergeordneten Serie *(blau)*. Um den Serien-Knoten angeordnet befinden sich die Bilder *(grün)* der Serie. Zusätzlich werden zum Patienten noch die *File*-Knoten *(beige)* und die Quelle *(dunkelgrün)* dargestellt sowie einige weitere Knoten, wie z.B. das Geschlecht *(hellblau)*. 39
3.7 Das Schema der Datenbank der Zweitdaten für den Proof of Concept. . . 39

3.8 Der Eingabepfad p besteht aus den grau unterlegten Knoten $v_i,\ i = 1,...,n$. Die weißen Knoten $u_j,\ j = 1,...,m$ dienen als Label der Knoten von p. . 40

4.1 Laufzeiten (in Minuten) des *main*-Teils des ersten *Importers* über 50 Testinstanzen für kleine, mittlere und große Daten. 51
4.2 Laufzeiten (in Minuten) des *print*-Teils des ersten *Importers* über 50 Testinstanzen für kleine, mittlere und große Daten. 52
4.3 Durchschnittliche Laufzeit (in Minuten) des *main*-Teils des ersten *Importers* über 50 Testinstanzen für kleine, mittlere und große Daten. 53
4.4 Laufzeiten (in Minuten) des *main*-Teils des zweiten *Importers* über 50 Testinstanzen für kleine, mittlere und große Daten. 54
4.5 Laufzeiten (in Minuten) des *print*-Teils des zweiten *Importers* über 50 Testinstanzen für kleine, mittlere und große Daten. 55
4.6 Durchschnittliche Laufzeit (in Minuten) des *main*-Teils des zweiten *Importers* über 50 Testinstanzen für kleine, mittlere und große Daten. . . . 56
4.7 Laufzeiten des Imports (in Sekunden) der *CSV*-Daten in die Graphdatenbank *Neo4j*. 57
4.8 Durchschnittliche Laufzeit der relevanten Teile der Querys *Q1 - Q5*. . . . 58
4.9 Durchschnittliche Laufzeit der einzelnen Teile der Querys *Q6 - Q9*. . . . 60
4.10 Präzision-*Recall*-Diagramm für die Querys *Q1 - Q5*. 62
4.11 Vergleich der Präzision, des *Recalls* und des *F1-Scores* der Querys *Q1 - Q5*. 63
4.12 Vergleich der *F1-Scores* der Querys *Q9 - Q13* ohne Duplikate. 66
4.13 Vergleich des *F1-Scores*, der Präzision und des *Recalls* der Querys *Q12* und *Q13* ohne Duplikate. 67
4.14 Präzison-*Recall*-Diagramm für die Versionen von *Q12*. 67
4.15 Präzison-*Recall*-Diagramm für die Versionen von *Q13*. 68

Tabellenverzeichnis

3.1 Beispielhafter Ausschnitt aus der *ini*-Datei für den Patientenknoten. 28
3.2 Auszug der Ausgabe der Query *Q1*. Die Bezeichnung *score* bezieht sich auf den Wert, der für den Knoten und sein Label durch den *Common Neighbors*-Algorithmus von *Neo4j* berechnet wurde. 41
3.3 Querys für Ein-Knoten-Pfade. 45
3.4 Querys für Mehr-Knoten-Pfade. 46
3.5 Querys mit Fehlern in *py2neo*. 48

4.1 Zusätzliche Querys zum Testen. 59
4.2 Detailbetrachtung des Ergebnisses für *Q1*. 61
4.3 Detailbetrachtung des Ergebnisses für *Q2*. 62
4.4 Ausschnitt der Detailbetrachtung des Ergebnisses für *Q4*. 63
4.5 Ausschnitt der Detailbetrachtung des Ergebnisses für *Q9*. 64

Abkürzungsverzeichnis

CSV Comma Separated Values

DICOM Digital Imaging and Communications in Medicine

IOD Information Object Definition

uid unique identifier

CRF Conditional Random Field

NER Named Entity Recognition

HPC High Performance Clustering

RRZK Regionales Rechenzentrum der Universität zu Köln

DZNE Deutsches Zentrum für neurodegenerative Erkrankungen

CoNLL Conference on Computational Natural Language Learning

IE Information Entity

BFGS Broyden-Fletcher-Goldfarb-Shanno

Kapitel 1

Einleitung

1.1 Motivation

1970 schrieb Edsger W. Dijkstra in seinen *Notes on Structured Programming*: „The art of programming is the art of organizing complexity, of mastering multitude and avoiding its bastard chaos as effectively as possible.“ [1]. Knowledge Graphen haben in den letzten Jahren stark dazu beigetragen, Ordnung in vermeintliches Chaos zu bringen. Insbesondere in der Abteilung der Bioinformatik des Fraunhofer Instituts SCAI und in Zusammenarbeit mit dem Deutschen Zentrum für neurodegenerative Erkrankung in Bonn (DZNE) kommen sie wiederholt zum Einsatz. Durch ihre weitreichenden Einsatzfelder, große Anpassungsfähigkeit und Skalierbarkeit eignen sie sich hervorragend zur Anordnung und Organisation großer Datenmengen. Zudem bieten sie die Möglichkeit, Daten in passenden Kontext einzubetten. [2] Im Internet werden Knowledge Graphen als Basis für viele Informationssysteme verwendet. Ein Beispiel hierfür bietet Googles Knowledge Graph aus dem Jahr 2012. Knowledge Graphen eigenen sich durch ihre entitären Bestandteile besonders zur Modellierung realer anwendungsnaher Szenarien. [3] Mit Hilfsmitteln wie z.B. der Graphdatenbank *Neo4j* stehen geeignete Mittel zur Visualisierung und Interaktion mit den Daten zur Verfügung [4]. Aus diesen Gründen bieten sie eine hervorragende Möglichkeit, anwenderfreundliche und realitätsnahe Ordnung in das Chaos großer Mengen klinischer Daten zu bringen. Durch ihre inhärente Struktur erschließen sich neue Möglichkeiten in der Forschung und Diagnostik. [2]
Eine dieser Möglichkeiten liegt im Bereich des maschinellen Lernens. Durch die stetig zunehmende Generierung von Daten kommt maschinelles Lernen zur Vorhersage von Verhaltensweisen, Erkennung von Clustern und für zahlreiche andere Zwecke zum Einsatz. Dabei sind die Anwendungsfelder mannigfaltig, z.B. Spam Filter, *Natural Language Processing*, Stimmerkennung und Überprüfung der Kreditwürdigkeit. [5] [6] [7] Maschinelles Lernen bedeutet nach [5], die Programmierung von Computern, so dass ein bestimmtes Kriterium ohne vollständiges Wissen anhand von vorhandenen Daten aus der Vergangen-

heit optimiert wird. Die großen Mengen an vorhandenen Daten bilden die Grundlage dafür, indem sie für die nahe Zukunft als korrekt angesehene Muster vorgeben, nach denen dann in neueren Daten gesucht werden kann. [7] [6] [5]
Nach [8] findet maschinelles Lernen auch im Bereich der medizinischen Daten umfangreiche Anwendung. Als Beispiel werden dort neben anderen vor allem Brustkrebsdaten angeführt. Die vorliegende Arbeit möchte die verschiedenen Bereiche weiter miteinander verbinden. Daten aus bildgebenden Verfahren der Medizin sollen in einen Knowledge Graphen überführt werden. Dann soll durch eine Abwandlung der *Named Entity Recognition* (NER) eine Möglichkeit für die Anwendung maschinellen Lernens, insbesondere *Link Prediction*, auf dem Graphen gezeigt werden. Dabei werden auch die Resultate untersucht.

1.2 Aufbau der Arbeit

Die Arbeit beginnt mit den notwendigen Grundlagen der Graphentheorie. Anschließend wird ein kurzer Überblick über Datenbanken, insbesondere über die hier verwendete Graphdatenbank *Neo4j*, sowie über Komplexitätstheorie gegeben. Der zweite Teil der Grundlagen beschäftigt sich mit einem Umriss bildgebender Verfahren und vertieft das Thema am Beispiel des *DICOM*-Formats, welches im Rahmen dieser Arbeit als Datengrundlage verwendet und genauer betrachtet wird. Im dritten Teil der Grundlagen wird *Link Prediction* betrachtet. Zunächst wird eine allgemeine Einführung zum Thema maschinelles Lernen gegeben. Danach folgt eine kurze Erläuterung einiger üblicher Verfahren für die praktische Umsetzung, die auch später Anwendung finden. Anschließend wird *Link Prediction* als ein Teil des maschinellen Lernens genauer beleuchtet. Dabei werden verschiedene mögliche Ansätze vorgestellt, die später teilweise zusammengeführt werden sollen.
Im dritten Kapitel werden zunächst die Ziele der Arbeit präsentiert. Darauf aufbauend ist der Rest des Kapitels zweigeteilt: Es stellt die methodische Umsetzung der beiden vorgestellten Probleme dar. Dabei wird im ersten Teil die Umsetzung des Datenimports aus *DICOM*-Dateien in die Graphdatenbank und im zweiten schließlich die danach erfolgte *Link Prediction* erläutert. Beide Teile beinhalten die verwendeten Python-Skripte. Der erste Teil geht dabei genauer auf die geforderte Flexibilität und Anpassbarkeit des Skripts und die daraus resultierende Erstellung des Datenschemas ein. Überdies wird der eigentliche Import der *CSV*-Dateien in die Graphdatenbank *Neo4j* beschrieben. Der zweite Teil beschäftigt sich dann exemplarisch mit verschiedenen Beispielen der *Link Prediction* unter Verwendung der *Conditional Random Fields*. Dies orientiert sich an der Dokumentation der Python-Bibliothek *sklearn-crfsuite*. Es werden beispielhafte Querys verschiedener Längen untersucht und die Ergebnisse präsentiert. Dafür kommt zum Teil der Hochleis-

tungsrechner *CHEOPS* des RRZK zum Einsatz.[1]
Im vierten Kapitel werden die Ergebnisse aus Kapitel drei bewertet. Auch dies erfolgt in zwei Teilen. Zunächst wird der *Importer* bezüglich seiner theoretischen Zeitkomplexität untersucht und diese dann mit der praktischen Laufzeit verglichen. Im zweiten Teil des Kapitels werden die Ergebnisse der *Link Prediction* bezüglich ihrer Laufzeit und der Güte des Ergebnisses evaluiert.
Das letzte Kapitel liefert einen Ausblick indem es sich abschließend mit noch offenen Problemen und einer möglichen Weiterführung des Themas befasst.
Der in dieser Arbeit verwendete Code, alle vollständigen Messwerte, alle Befehle und Resultate sind in einem *Git Repository* hinterlegt: `https://github.com/TbsHbnthl/master-s-thesis-link-prediction-on-large-scale-knowledge-graphs`.

[1] An dieser Stelle bedanke ich mich beim RRZK für die Bereitstellung und Nutzungsgenehmigung der Systeme des Hochleistungsrechners *CHEOPS* für diese Arbeit.

Kapitel 2

Grundlagen

2.1 Graphentheorie

Diese Arbeit beruht in weiten Teilen auf den Grundlagen der Graphentheorie. Daher wird in diesem Kapitel eine kurze Einführung mit den relevanten Definitionen vermittelt.

Definition 2.1.1 *Ein gerichteter Graph $D = (V,A)$ besteht aus einer nicht leeren endlichen Menge V, deren Elemente Knoten genannt werden, und einer endlichen Menge A, deren Elemente geordnete Tupel paarweise verschiedener Knoten sind.* [9]

Definition 2.1.2 *Ein Graph $D' = (V',A') \subseteq D = (V,A)$ ist ein Subgraph von G, wenn die Bedingungen $V' \subseteq V$ und $A' \subseteq A$ erfüllt sind.* [10]

Definition 2.1.3 *Ein Property Graph D_p ist ein Tupel $D_p = (V,A,\rho,\lambda,\sigma)$, für das gilt:*

1. *V ist eine endliche Knotenmenge.*
2. *A ist eine endliche Kantenmenge.*
3. *$\rho : A \to (V \times V)$ ordnet jeder Kante in A ein Knotenpaar (u,v) aus Knoten $u,v \in V$ zu.*
4. *$\lambda : (V \cup A) \to L$ ist eine Funktion, die jedem Knoten $v \in V$ und jeder Kante $a \in A$ ein Label $l \in L$ zuweist.*
5. *$\sigma : (V \cup A) \times P \to X$ ist eine Funktion, die jedem Knoten $v \in V$ und jeder Kante $a \in A$ Properties $p \in P$ zuordnet. Jedem solchen p werden dabei Werte $x(p) \in X$ zugewiesen.* [11]

Für zwei Knoten $v_1, v_2 \in V$ und eine Kante $a \in A$ mit $\rho(a) = (v_1, v_2)$ wird v_1 als Startknoten und v_2 als Zielknoten bezeichnet. [11]

In [12] wird ein Knowledge Graph, oder auch semantisches Netzwerk, als Möglichkeit, Informationen und Daten systematisch auf einem abstrakten Level mit Knowledge zu verbinden, verstanden. Nach [13] muss ein Knowledge Graph verschiedene Kriterien erfüllen:

- Es werden hauptsächlich Entitäten aus der realen Welt und deren Relationen zueinander beschrieben und in einem Graphen dargestellt.
- Klassen und Relationen werden mithilfe eines Schemas definiert.
- Beliebige Entitäten können paarweise miteinander durch Relationen verbunden werden.
- Mehrere Themenbereiche werden abgedeckt.

Eine Entität bezeichnet dabei ein eindeutig abgrenzbares und identifizierbares Objekt. Insgesamt wird für die vorliegende Arbeit folgende Definition zugrunde gelegt:

Definition 2.1.4 *Ein Knowledge Graph $D = (E,R)$ ist ein Graph, dessen Knotenmenge E aus Entitäten und dessen Kantenmenge R aus Relationen zwischen Entitäten besteht. $E = \{E_1,...,E_n\}$ ist eine Vereinigung strukturierter Ontologien E_i und $R = \{R_1,...,R_m\}$ analog eine Vereinigung intra- und interontologischer Relationen. Jede Entität $E_i \in E, i \in \{1,...,n\}$ kann mit zusätzlichen Kontextinformationen $C = \{C_1,...,C_k\}$ über eine weitere Relation verknüpft werden. Diese Kontextinformationen sind im Allgemeinen selbst wieder Entitäten.* [12]

Diese Definition kann man für die spätere Anwendung in *Neo4j* mit den in Definition 2.1.3 definierten Funktionen ρ, λ und σ verknüpfen, um den Knowledge Graphen zu erhalten, der im weiteren Verlauf der Arbeit verwendet wird.

Definition 2.1.5 *Ein Pfad ist ein nicht-leerer Graph $P = (V,E)$ mit $V = \{v_0, v_1, ..., v_k\}$, $E = \{(v_0,v_1),(v_1,v_2),...,(v_{k-1},v_k)\}$, wobei alle v_i, $i \in \{1,2,...,k\}$ paarweise verschieden sind. Die Knoten v_0 und v_k sind durch P verknüpft und heißen Endknoten von P. Die vereinfachte Darstellung von P ist $P = (v_0, v_1, ..., v_k)$. Sofern den Kanten nicht über eine Funktion $l : E \to \mathbb{R}$ eine Länge $l \in \mathbb{R}$ zugewiesen wird, gibt die Anzahl der Kanten in einem Pfad P seine Länge an.* [10]

Definition 2.1.6 *Wenn $P' = (v_0, v_1, ..., v_k)$ ein Pfad in einem (gerichteten) Graphen $D = (V,A)$ mit $v_i \in V$ $\forall i = 1,...,k$ mit $k \geq 3$ ist, nennt man $P = P' + v_0$ einen Kreis.* [10]

Definition 2.1.7 *Ein Graph $G = (V,E)$ heißt zusammenhängend, wenn er nicht-leer ist und zwei beliebige paarweise verschiedene Knoten $v_i, v_j \in V$ über einen Pfad P in G verbunden sind.* [10]

Definition 2.1.8 *Ein Baum ist ein kreisfreier und zusammenhängender (Sub-)Graph.* [10]

Definition 2.1.9 *Ein Spannbaum $G' = (V', E')$ ist ein Subgraph eines Graphen $G = (V, E)$, für den gilt, dass er ein Baum ist und dass $V' = V$.* [10]

Definition 2.1.10 *Sei $G = (V, E)$ ein Graph. Die k-Nachbarschaft $\Gamma_k(v)$ eines Knoten $v \in V$ ist die Menge aller Knoten v', die zu v eine Entfernung haben, die kleiner oder gleich k ist, wobei $k \geq 1$ gilt.* [14] *Für $k = 1$ ist $\Gamma(v)$ die (direkte) Nachbarschaft des Knoten $v \in V$. Analog lässt sich die Nachbarschaft $\Gamma(U)$ einer Teilmenge $U \subseteq V$ definieren.* [10]

Definition 2.1.11 *Sei $G = (V, E)$ ein (ungerichteter) Graph. Eine Clique $C \subseteq V$ bezeichnet eine Teilmenge der Knoten, so dass $\forall u, v \in C \; \exists \; e = (u, v) \in E$ oder $e = (v, u) \in E$. Eine Clique heißt maximal, wenn sie nicht Teilmenge einer größeren Clique ist.* [15]

Definition 2.1.12 *Sei $G = (V, E)$ ein nicht-leerer Graph. Der Grad $d(v)$ eines Knoten $v \in V$ bezeichnet die Anzahl der Kanten $e \in E$, s.d. $v \in e$.* [16]

Definition 2.1.13 *Sei $G = (V, E)$ ein Graph und I eine Indexmenge. Communitys sind Teilmengen $V_i = \{v_{i_1}, ...v_{i_k}\} \subseteq V$, $i \in I$, für die gilt, dass die Knoten $v_{i_1}, ...v_{i_k}$ untereinander eine hohe Kantendichte und gleichzeitig zu Knoten $v_{j_1}, ..., v_{j_l}$ einer anderen Community V_j eine geringe Kantendichte aufweisen.* [16]

Dies bildet die theoretische Basis für den Graphen, der später in der Arbeit verwendet wird. Da dieser in der praktischen Anwendung innerhalb einer Datenbank gespeichert wird, führt dies zum nächsten Teil der Grundlagen.

2.2 Datenbanken

Hier soll das Thema Datenbanken kurz umrissen werden, da insbesondere die Graphdatenbank *Neo4j* eine zentrale Rolle in dieser Arbeit spielt. Datenbanken dienen der Organisation von Informationen. Sie stellen strukturierte Datenmengen dar, deren Elemente gespeichert und durch eine datenbankspezifische Sprache abgefragt und bearbeitet werden können. Dabei werden vier Funktionen unterschieden:

- Modifikation der Anordnung der Daten innerhalb der Datenbank,
- Änderung, Erstellung oder Löschen der Daten (Die letzten beiden Bereiche sind als Update der eigentlichen Datenbank zu verstehen, da sie die Struktur der Datenbank beeinflussen.),
- Zugriff auf die Daten innerhalb der Datenbank oder durch externe Anwendungen,
- administrative Tätigkeiten zur Verwaltung der Datenbank. [4]

Um diese Schritte ausführen zu können, wird ein Datenbanksystem (engl. *database management system (dbms)*) verwendet. Dieses teilt sich in die Speicherungs- und Verwaltungskomponente. Erstere beinhaltet alle abgespeicherten Daten und ihre Beschreibung. Letztere umfasst die datenbankspezifische Sprache und verwaltet Zugriffsrechte der Benutzer. Es wird zwischen verschiedenen Datenbanken differenziert. Im Folgenden werden kurz die Unterschiede zwischen relationalen Datenbanken und Graphdatenbanken erläutert. [17][4]

2.2.1 Relationale Datenbanken

Relationale Datenbanken bestehen aus Tabellen bzw. Relationen. Diese erhalten einen Tabellennamen und innerhalb der Tabellen werden den Spalten Attributsnamen zugeordnet. Eine Tabellenspalte enthält als Attribut eindeutige Schlüssel (engl. *keys*), die der Identifikation dienen. Dies ist notwendig, da einzelne Datenwerte in der Tabelle mehrfach vorkommen können. Als Beispiel, das zur späteren Anwendung passt, kann eine Tabelle mit Patienten[1] betrachtet werden. Diese erhalten jeweils einen eindeutigen Schlüssel zur Identifikation. Datenwerte einer solchen Tabelle, die mehrfach vorkommen können, sind beispielsweise Alter oder Namen. Die Anzahl der Attribute, also der Spalten, ist beliebig und der Reihenfolge wird keine Bedeutung beigemessen. Gleiches gilt für die Zeilen der Tabelle, die als Tupel aus einem eindeutigen Identifikationsschlüssel und den Werten der weiteren Attribute bestehen. Diese Organisation der Datenbank erlaubt mengenorientierte Abfragen und Manipulationen. Das bedeutet, dass Ergebnisse als Tabelle zurückgegeben werden. Falls keine Übereinstimmung gefunden wurde, ist diese Tabelle leer. Der Vorteil dieser Art der Abfrage liegt darin, dass mehrere Aktionen in der Datenbank ausgeführt werden können. Dadurch wird Kleinschrittigkeit vermieden. Bei prozeduralen Datenbanksprachen hingegen müssen die Abläufe zur Suche und Bereitstellung der Informationen vom Anwender selbst implementiert werden. Dort werden satzorientierte oder navigierende Befehle verlangt, eine Tatsache, die umfangreichere Kenntnis erforderlich macht. [4][17][18] Die meist verwendete Sprache ist die *Structured Query Language* (SQL). Sie ist deskriptiv, da sie das Resultat und nicht die Rechenschritte beschreibt, und folgt einem festen Muster. Durch die strukturierte Sprache kann das Datenbanksystem Abfragen des Benutzers mit eigenen internen Methoden bearbeiten. Dies vereinfacht die Nutzung. Die Sprache untergliedert sich in vier Bereiche: Datendefinition, Datenmanipulation, Datenabfrage und Datenschutz. [17] [19]
Das relationale Datenbanksystem ist dabei durch folgende Aspekte charakterisiert:

- Modell: Alle Daten und Relationen sind in Tabellen hinterlegt.

[1] Aus Gründen der Lesbarkeit wird im Text verallgemeinernd das generische Maskulinum verwendet. Diese Formulierungen umfassen gleichermaßen Personen aller Geschlechter.

- Schema: Die Tabellen- und die Attributsdefinitionen sind im relationalen Datenbankschema hinterlegt.
- Sprache: Die verwendete Sprache ist deskriptiv für Datendefinition, -selektion und -manipulation.
- Architektur: Anwendungsprogramme und die eigentlichen Daten bleiben weitgehend voneinander getrennt.
- Mehrbenutzerbetrieb: Mehrere Benutzer können gleichzeitig mit dem Datenbanksystem arbeiten, ohne dass sie sich gegenseitig behindern oder die Datenintegrität in Gefahr gerät.
- Konsistenzgewährung: Das Datenbanksystem sorgt für korrekte Speicherung der Daten.
- Datensicherheit: Das Datenbanksystem sorgt für den Schutz der Daten vor Zerstörung oder unbefugtem Zugriff. [17]

Relationale Datenbanken sind daher weit verbreitet und finden in vielen Bereichen Anwendung. [17]

2.2.2 NoSQL- und Graphdatenbanken

Nach [17] stellt stellt das aktuelle Thema *Big Data* ein Problem für relationale Datenbanken dar. Der Begriff *Big Data* orientiert sich größtenteils an drei V's:

- *Volume*: Es handelt sich um einen sehr umfangreichen Datenbestand.
- *Variety*: Es wird eine große Bandbreite an Datentypen verarbeitet.
- *Velocity*: Es wird erwartet, dass die Daten in der gleichen Geschwindigkeit, in der sie ankommen, auch ausgewertet werden. [17]

Darüber hinaus werden dort und in [20] noch zwei weitere V's für *Value* und *Veracity*, also Wert und Aussagequalität, genannt. Für diese Form der Daten eignen sich relationale Datenbanken nach [20] und [17] ab einer gewissen Datenmenge nicht mehr. Die Bandbreite der verschiedenen Datentypen (*Variety*) verträgt sich nicht mit den im Vorhinein festgelegten Datenschemata der relationalen Datenbanken. Außerdem ergibt sich nach [20] ein Problem bei der Architektur: Datenvolumen in Höhe von 100 Terabyte benötigen in der Regel eine Aufteilung auf mehrere physische Maschinen (auch horizontales Skalieren genannt), was die Komplexität erhöht und die Stabilität gefährden kann. Die Alternative der vertikalen Skalierung dagegen ist nach [20] durch die Anschaffung neuer,

verbesserter Hardware mit deutlich höheren Kosten verbunden. Dies führt zu den NoSQL-Datenbanken, die nach [20] und [17] sehr gut horizontal skalieren. [20] [17]
NoSQL-Datenbanken (engl. für *not (only) SQL*) bezeichnen nach [17] nicht-relationale Datenbanken, die zwei Bedingungen erfüllen:

- Die Daten werden nicht wie in relationalen Datenbanken in klassischen Tabellen gespeichert.
- Die Datenbanksprache ist nicht (nur) SQL. Es kommen also nicht ausschließlich relationale Datenbanktechniken zum Einsatz.

Die Daten innerhalb einer NoSQL-Datenbank werden in einer stark verteilten Datenhaltungsarchitektur als *key-value*-Paare, in Spalten, Dokumentenspeichern oder Graphen gespeichert. [17]
Key-value-Datenbanken stellen die einfachste Form dar. Daten werden mithilfe eines Datenobjekts als Identifikationsschlüssel, dem *key*, und einem Datenobjekt als Wert gespeichert. Der Raum der Schlüssel unterstützt, abgesehen von verwendeten Sonderzeichen, keine weitere Struktur. Die Datenbank ist schemafrei, wodurch Datenobjekte in beliebiger Form hinterlegt werden können. Eine Erweiterung dieses simplen Prinzips bieten die spaltenorientierten Datenbanken. Für einen verbesserten Lesezugriff werden die Daten spaltenweise gespeichert, da für eine Zeile meist nicht alle Spalten benötigt werden, es umgekehrt aber häufig verwendete Spalten gibt. Die Datenobjekte werden dann über Zeilenschlüssel und die Objektattribute über Spaltenschlüssel adressiert. Spalten gleichen Typs können zu Spaltenfamilien zusammengefasst werden. Es wird davon ausgegangen, dass sie zusammen gelesen werden können. Die Schemata einer Tabelle beziehen sich lediglich auf Spaltenfamilien, nicht auf ganze Tabellen. [17] [4] [20]
Dokumentendatenbanken vereinen Schemafreiheit und die Möglichkeit, gespeicherte Daten zu strukturieren, was bei *key-value*-Datenbanken nur sehr eingeschränkt möglich ist. Solche strukturierten Datensätze werden Dokumente genannt. Durch die Schemafreiheit entsteht große Flexibilität bei der Datenspeicherung, wodurch sie sich gut für *Big Data* eigenen. Auf der äußersten Ebene sind sie *key-value*-Datenbanken, deren gespeicherte Werte zu den Schlüsseln den Dokumenten entsprechen. Diese enthalten auf einer zweiten Ebene eine eigene, dokumentspezifische Struktur, die üblicherweise aus rekursiv verschachtelten Attribut-Wert-Paaren besteht. Auch diese Strukturen sind schemafrei. [17][4] [20]
Die Graphdatenbank unterscheidet sich deutlich von den anderen Modellen. Die Daten werden als Knoten und Kanten gespeichert, die jeweils einem Typ oder Label angehören. Sie enthalten selbst wiederum Daten als Werte für ihre Attribute. Die Daten einer Graphdatenbank werden als Graph abgebildet. Ein Beispiel ist in Abbildung 2.1 dargestellt. Das Schema der Datenbank ist implizit, was bedeutet, dass neue Daten der Datenbank hinzugefügt werden können, ohne vorher den Typ zu kennen. Dieser wird dann von

der Datenbank selbst erstellt. Graphdatenbanken werden verwendet, wenn die Relationen der Datensätze zueinander wichtig sind. Beispiele dafür sind Soziale Netzwerke, Verlinkung von Webseiten untereinander oder auch diese Arbeit. Ein besonderer Vorteil ist die indexfreie Nachbarschaft: Es ist möglich, die direkten Nachbarn eines Knoten zu ermitteln, ohne alle Kanten berücksichtigen zu müssen, wie es in einer relationalen Datenbank der Fall wäre. Der Aufwand der Abfrage einer Beziehung ist daher unabhängig von der Datenmenge. Bei einer relationalen Datenbank wächst er mit der Anzahl der gesuchten Referenzen. [17] [4] [18] [20]

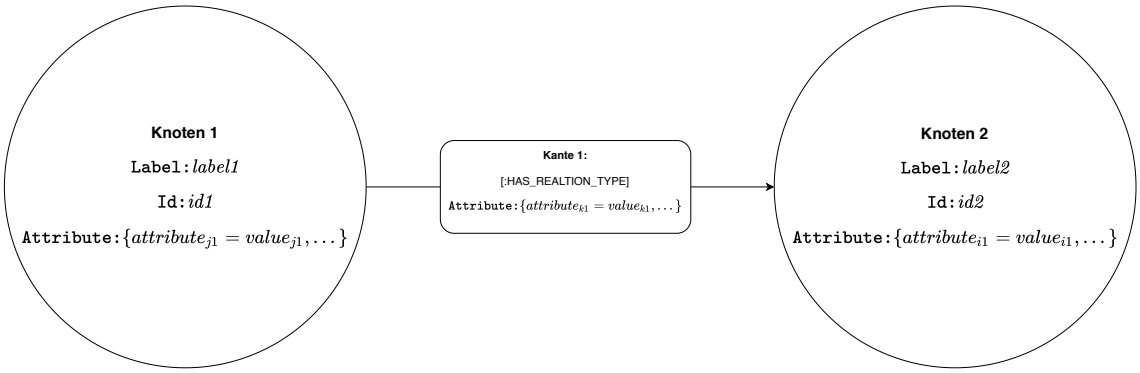

Abbildung 2.1: Beispielhafte Darstellung zweier generischer Knoten in *Neo4j*, die durch eine Kante verbunden sind.

2.3 Komplexitätstheorie und Laufzeitanalyse

Zur Ermittlung der Zeitkomplexität eines Algorithmus wird der ungünstigste Fall untersucht und dessen Laufzeit durch obere Schranken begrenzt. Es gibt noch weitere Betrachtungsmöglichkeiten, wie zum Beispiel eine Einschränkung der Laufzeit nach oben und unten oder eine stärkere Restriktion bezüglich der oberen Schranke. Diese werden hier nicht weiter betrachtet, da eine geringere Komplexität kein Problem darstellt. Dabei hängt die Laufzeit von der Eingabeinstanz $\mathcal{I}$, der Codierungslänge $\mathcal{L}$ der Eingabe, dem Computer $\mathcal{C}$, der verwendeten Programmiersprache $\mathcal{P}$ und der Implementierung $\mathcal{F}$ des Algorithmus ab. Computer unterscheiden sich sowohl zu einem bestimmten Zeitpunkt als auch über die Zeit betrachtet untereinander. Sie besitzen nur eine kurze Verwendungsdauer, bevor die Technik überholt ist. Auch Programmiersprachen werden verändert und gewinnen oder verlieren an Bedeutung. Daher wird üblicherweise von einem generischen Computer und einer ebenso generischen Programmiersprache ausgegangen. Damit hängt die Laufzeit eines Algorithmus noch von $\mathcal{I}$ und $\mathcal{L}$ ab. Statt der Angabe einer Zeiteinheit wird die Anzahl der auszuführenden Berechnungsschritte des Computers gezählt und in Abhängigkeit von der Eingabe angegeben. Für die Angabe der Laufzeit wird die $\mathcal{O}$-Notation verwendet. Es wird zwischen konstanter $\mathcal{O}(k \cdot 1)$, polynomieller $\mathcal{O}(\alpha \cdot n^{\beta})$ und exponentieller $\mathcal{O}(x^n)$ Laufzeit unterschieden. Dabei sind k, α, β und x Parameter, die nicht

von der Eingabe abhängen. Im ersten Fall ändert sich die Laufzeit nicht mit wachsender Eingabeinstanz n. Im zweiten Fall wächst die Laufzeit nicht schneller als ein Polynom der Eingabeinstanz n. Im letzten Fall hingegen steigt die Laufzeit höchstens exponentiell im Verhältnis zum Wachstum der Eingabeinstanz n. [21] [22] [23]

Definition 2.3.1 *Sei* $f : \mathbb{N}_0 \to \mathbb{R}_{\geq 0}$. *Dann ist* $\mathcal{O}(f) = \{g : \mathbb{N}_0 \to \mathbb{R}_{\geq 0} | \exists n_0 \in \mathbb{N}_0, c \in \mathbb{N} : g(n) \leq c \cdot f(n) \forall n \geq n_0\}$ *die Menge aller Funktionen g, die nicht schneller wachsen als die Funktion f mit einem Skalar. Vereinfacht wird statt* $g \in \mathcal{O}(f)$ *auch* $g = \mathcal{O}(f)$ *geschrieben.* [23]

Theorem 2.3.2 *Seien* $d \in \mathbb{R}_{\geq 0}$ *und* $f, g, h : \mathbb{N}_0 \to \mathbb{R}_{\geq 0}$. *Dann gilt:*

1. $d = \mathcal{O}(1)$,
2. $f = \mathcal{O}(f)$,
3. $d \cdot f = \mathcal{O}(f)$,
4. $f + g = \mathcal{O}(max\{f, g\})$,
5. $f \cdot g = \mathcal{O}(f \cdot h)$, *falls* $g = \mathcal{O}(h)$.

Der Beweis des Theorems ist in [23] zu finden.

2.4 Bildgebende Verfahren in der Medizin

2.4.1 Allgemeines

Da diese Arbeit bildgebende Verfahren in der Medizin mit Graphentheorie und maschinellem Lernen verbinden soll, wird hier ein kurzer Einblick in ersteres vermittelt. Insbesondere wird später das *DICOM*-Format herausgearbeitet.
Bildgebende Verfahren sind Diagnosemethoden der Medizin, die unter anderem in der Krebsmedizin Anwendung finden. Das erste bildgebende Verfahren war das Röntgen, das gegen Ende des 19. Jahrhunderts entwickelt wurde. Mit den technischen Systemen der Bildgebung entwickelten sich gleichzeitig entsprechende medizinische Fachrichtungen für die Aufnahme und Interpretation: die Radiologie und die Nuklearmedizin. Heute zählen auch Ultraschall, Computertomographie, Kernspintomographie und andere dazu. [24] [25]
Es wird dabei zwischen verschiedenen Formen der Bildgebung unterschieden. Den Anfang machte die morphologische Bildgebung, die für die Erkennung von Tumoren eingesetzt wird. Dann folgte die funktionelle Bildgebung, bei der herausspülbare Kontrastmittel injiziert werden, um funktionelle Prozesse zu beurteilen. Darüber hinaus gibt es noch die quantitative, die molekulare, die interventionelle und die multimodale Bildgebung. Sie

liefern zusätzliche Messgrößen sowie Bilder in Echtzeit während Eingriffen und ermöglichen sogar, aufgrund biochemischer Vorgänge Gewebeveränderungen vorherzusagen. [25]
Die Verfahren haben in den letzten Jahren verstärkt Anwendung in der Medizin gefunden und zum Fortschritt in den entsprechenden Bereichen beigetragen. In der Diagnostik kommen nach [25] heutzutage neun wichtige Verfahren zum Einsatz: Projektionsröntgen, Szintigraphie, Einzelphotonen-Emmissionstomographie, Positronen-Emissions-Tomographie, Magnetresonanztomographie, Ultraschall, Endoskopie, optische Kohärenztomographie, Operationsmikroskopie und Computertomographie. Letztere konnte in den letzten Jahren durch Erweiterungen und Weiterentwicklung ein noch breiteres Anwendungsgebiet als zuvor finden. Die Resultate dieser Verfahren werden digital gespeichert. Sie enthalten eine große Menge an Informationen zum Patienten, dem Untersuchungsverfahren und zusätzlichen Modalitäten sowie zu Diagnosen. Ein wichtiges Beispiel für derartige Dateien bietet das im weiteren Verlauf der Arbeit betrachtete *DICOM*-Format. [25] [26] Weitere Informationen zu diesen einzelnen Verfahren und zu derzeit noch in der Entwicklung befindlichen Verfahren können [25] entnommen werden. Aber nicht nur bei der Diagnostik, sondern auch bei der Therapie stellen bildgebende Verfahren einen wichtigen Bestandteil dar, denn sie ermöglichen die Dokumentation der potentiellen Erfolge einer Behandlung. Die Weiterentwicklung mithilfe von Medizintechnik, Computer- und Informationstechnologie ermöglichen nicht-invasive und zum Teil sogar dreidimensionale Einblicke in Bereiche, die dem Auge nicht ohne weiteres zugänglich sind. Darüber hinaus ermöglichen Computer und künstliche Intelligenz die stetig wachsenden Datenmengen anhand von Algorithmen automatisch zu verarbeiten. Dadurch wird der Befundungsprozess unterstützt. [27]
Insbesondere das maschinelle Lernen verändert inzwischen viele Berufsfelder und hat erheblichen Einfluss auf verschiedene Bereiche der Gesellschaft. Computer sind anhand von Algorithmen in der Lage, Muster zu erkennen, die sich dem Menschen nicht erschließen. Darüber hinaus ermöglichen Computer auch die Integration klinischer Daten in bestehende Systeme und die Vereinigung verschiedener Datenquellen. Auf Basis der aus Bilddaten durch Algorithmen extrahierten Merkmale können dann Aussagen getroffen werden. [28]

2.4.2 *DICOM*

DICOM ist die Abkürzung für *Digital Imaging and Communications in Medicine*. Darunter versteht man einen über viele Jahre erstellten universellen Standard in digitaler Bildgebung in der Medizin. Dieser bezieht sich sowohl auf die Speicherung als auch die Kommunikation diagnostischer und therapeutischer Informationen, Bilder und zugehöriger Daten. Der Fokus des Standards liegt auf Konnektivität und Kompatibilität, um Arbeitsabläufe zu vereinfachen. Dabei wird besonderer Wert auf die folgenden Kriterien

gelegt: Bildqualität, umfängliche Unterstützung verschiedener Datentypen und Parameter der Bild-Akquise, Kodierung der medizinischen Daten und Klarheit bei der Beschreibung der zugehörigen Geräte und ihrer Funktionalität. [29] [26] Dieser Standard stellt damit alle notwendigen Voraussetzungen für die normierte Darstellung und Verwendung der Daten zur Verfügung. Neben dem eigentlichen Bild werden in jeder Datei ebenfalls alle zugehörigen medizinischen Daten, die für die Radiologie relevant sind, gespeichert. Dafür stehen mehrere tausend standardisierte Attribute aus dem *DICOM data dictionary* zur Verfügung. Beispiele hierfür sind Daten zum Patienten wie der Names des Patienten, sein Geschlecht und Alter, aber auch Daten zum eigentlichen Bild wie die Farbtiefe oder der Zeitpunkt der Aufnahme des Bildes. [26]
Der eigentliche Aufbau der Daten, die einem *DICOM*-File zugrundeliegen, gleicht einem Baum. Da diese Struktur in Unterabschnitt 3.1.1 noch eine Rolle spielt, wird Näheres dazu dort erläutert.

2.5 *Link Prediction*

2.5.1 Einführung in das maschinelle Lernen

Zunächst soll ein kurzer Überblick über maschinelles Lernen und künstliche Intelligenz gegeben werden. Dafür wird als erstes betrachtet, was mit Intelligenz gemeint ist. Nach [30] ist es schwierig, eine allgemeingültige Definition zu geben. Intelligenz ist nicht greifbar und wird meist als das definiert, was mithilfe von Intelligenztests messbar ist. Der Begriff kann aber in unterschiedlichen kulturellen Kontexten verschiedene Bedeutungen besitzen. Allgemein gilt, dass Intelligenz als die Fähigkeit, aus Erfahrungen zu lernen und Probleme zu lösen, betrachtet wird. Nach [6] wird unter künstlicher Intelligenz etwas verstanden, das folgende Punkte erfüllt:

- logisches Denken
- Treffen von Entscheidungen
- Planen
- Lernen
- Kommunikation.

Der Autor beschreibt, dass hier noch weitere Aspekte wie Empfindungen und Bewusstsein genannt werden können, von denen die heutige Forschung aber noch weit entfernt ist. Davon abgesehen stellt der erstgenannte Punkt nach [6] die größte Schwierigkeit der künstlichen Intelligenz dar.
Das maschinelle Lernen deckt vor allem den vierten Punkt, aber die Punkte zwei, drei

und fünf ab. Für den Bereich des maschinellen Lernens wird unter Intelligenz eher intelligentes Verhalten verstanden und das Lernen wird als Möglichkeit zur Optimierung dieses Verhaltens aufgefasst. Unter maschinellem Lernen wird also im Allgemeinen die Verbesserung artifiziellen Verhaltens verstanden, die auf Rückmeldungen der Umwelt basiert. Es werden drei Bereiche unterschieden: das überwachte, das unüberwachte und das bestärkende Lernen. Alle drei werden im weiteren Verlauf noch kurz betrachtet. [6] [31]

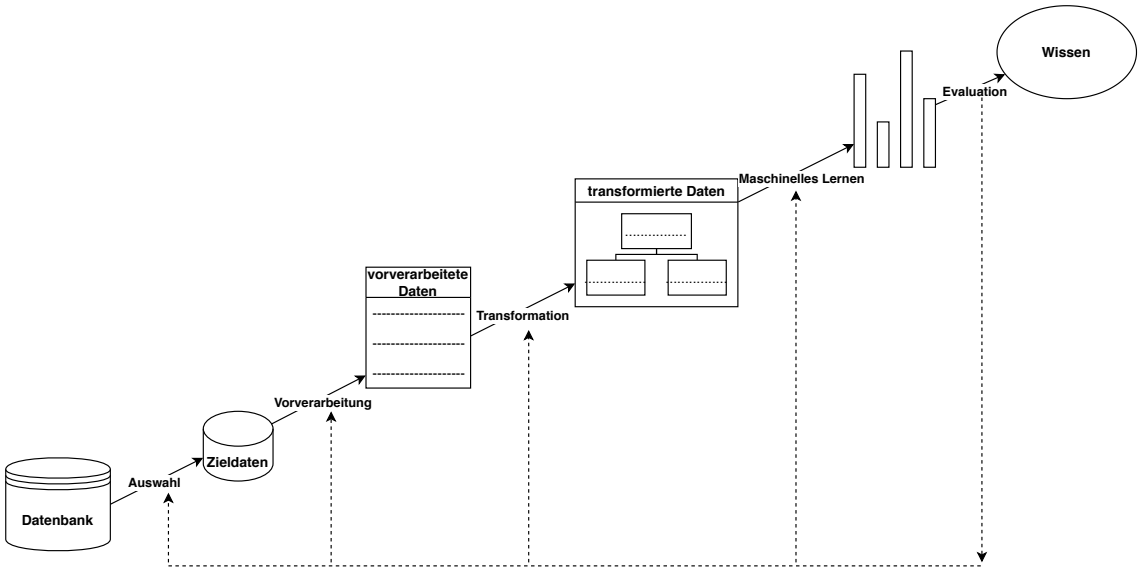

Abbildung 2.2: Darstellung des Prozesses der Knowledge Discovery.

Neben der künstlichen Intelligenz existiert noch Knowledge Discovery. Diese basiert auf einer gegebenen Sammlung von Daten. Als Knowledge Discovery wird dabei die Extraktion von Wissen aus dieser Datenmenge verstanden. Hierbei trifft ein Anwender immer wieder Entscheidungen, auf Basis derer dann die Schritte der Knowledge Discovery wiederholt werden. Dies wird in Abbildung 2.2 dargestellt. Für die Mustererkennung in Daten fällt *Data Mining* innerhalb der *Knowledge Discovery* mit dem überwachten Lernen zusammen. [6]

Überwachtes Lernen

Beim überwachten Lernen besteht der Datenstrom aus Paaren $(X_1, y_1), (X_2, y_2), ..., (X_n, y_n)$. Die X_i stellen die Eingaben und die y_i die zugehörigen Sollwerte für alle $i \in \{1, ..., n\}$ dar. Diese Sollwerte markieren den zuvor schon bekannten, korrekten Funktionswert bzw. bei Datensätzen das korrekte Label. Die Datenmenge wird in eine Trainings- und eine Validierungsmenge geteilt. Dabei wird anhand ersterer der Algorithmus trainiert und mithilfe von Vergleichen mit den Sollwerten in zweiterer optimiert. Es werden im Wesentlichen zwei Bereiche unterschieden: Regression und Klassifikation. Die Klassifikation verwendet eine diskrete Zielmenge $Y = y_1, y_2, ..., y_m$ für die Zuordnungsfunktion $f : X \to Y$. Die

Konstruktion dieser Funktion f löst das Klassifizierungsproblem. Dies ist auch die Rubrik, unter welche die später erwähnten *Scores* fallen und der Teil des maschinellen Lernens, der später praktisch verwendet wird. Die Regression hingegen geht von einer kontinuierlichen Zielmenge aus. Sie wird aber für diese Arbeit nicht weiter benötigt, da der Graph eine diskrete Menge an Labeln zur Verfügung stellt. [6] [31]

Bestärkendes Lernen

Wenn statt richtig und falsch nur bekannt ist, was wünschenswerte und nicht wünschenswerte Ergebnisse sind, kann auf bestärkendes Lernen zurückgegriffen werden. Beim bestärkenden Lernen wird ein sogenannter Agent verwendet, der aufgrund seiner Aktionen $a_1, a_2, ..., a_k$ nicht notwendigerweise positive Belohnungen $r_1, r_2, ..., r_k$ erhält. Zusätzlich zur Ein- und Ausgabe existiert also eine Belohnungsmenge. Das Ziel ist die Maximierung zukünftiger Belohnungen. Dies ist aber nicht das eigentliche Thema dieser Arbeit und wird daher hier nicht weiter erläutert. [6][31]

Unüberwachtes Lernen

Schließlich gibt es noch das unüberwachte Lernen. Dieses enthält keine Sollwerte, sondern nur Eingabewerte $X_1, X_2, ..., X_l$. Es wird versucht, versteckte Strukturen aufzudecken. Gesucht wird also eine passende Repräsentation anhand von Charakteristika der Daten. Die Daten werden im Vorhinein nicht klassifiziert, sondern es wird während des Lernens eine Klasseneinteilung erstellt. Dabei kann kein Fehler berechnet oder angegeben werden und es ist schwierig, die Ausgabe des Computers zu bewerten. Je nach Kontext können dabei auch sehr verschiedene Klassifikationen sinnvoll erscheinen. Dieses Clustering wird anhand von Merkmalen erstellt und kann auch mit deren Hilfe verändert werden. [6][31]

2.5.2 *Gradient Descent*, Newton und BFGS

In diesem Teil werden kurz mögliche Ansätze zur Lösung der Probleme des maschinellen Lernens vorgestellt, da der spätere Algorithmus für die *Conditional Random Fields* darauf beruht. Zunächst wird die Methode des *Gradient Descent* betrachtet. Sie findet häufig Anwendung bei der Bestimmung lokaler Extrema einer differenzierbaren Funktion f. Es wird ein beliebiger Startpunkt gewählt und schrittweise in Richtung des umgekehrten Gradienten iteriert. Es wird also stets die Richtung mit dem steilsten Fall der Funktion f gewählt. Dies lässt sich als $x_{k+1} = x_k + \eta \cdot \nabla f(x_k)$ schreiben. η beschreibt die Lernkurve, die auch die Schrittweite auf dem Weg zum lokalen Minimum festlegt. [32]
Eine Alternative zum *Gradient Descent* bietet das Newton-Verfahren. Dieses integriert zusätzlich die Hesse-Matrix H der Funktion f, wodurch das Verfahren weitere lokale Informationen erhält. Die allgemeine Form sieht dabei wie folgt aus: $x_{k+1} = x_k - H(x_k)^{-1} \nabla f(x_k)$.

Hier muss keine Schrittweite mehr bestimmt werden, da diese bereits vorgegeben ist. Dieses Verfahren konvergiert quadratisch, wodurch es wesentlich schneller zum Ziel führt als *Gradient Descent*, allerdings ist es auch instabiler hinsichtlich der Eingabeparameter. Der Startpunkt muss nah genug am Minimum liegen und die Hesse-Matrix sollte positiv definit sein, damit sie invertiert werden kann. Außerdem ist die Berechnung der Inversen der Hesse-Matrix in jedem Schritt mit $\mathcal{O}(n^3)$ sehr aufwendig. [33] [34]
Die Klasse der Quasi-Newton-Verfahren verbindet die beiden Methoden und bietet eine Hybridlösung. Hier wird die Hesse-Matrix mit einer positiv definiten Matrix approximiert. Aus der positiven Definitheit folgt die Konvexität der Funktion, die der Problematik mit dem Newton-Verfahren entgegenwirkt. Die neue approximierte Hesse-Matrix wird dann aus den Informationen der vorherigen Zeitschritte berechnet. Die Approximation A der Hesse-Matrix muss dabei die Quasi-Newton-Gleichung $A_{k+1}[x_{k+1} - x_k] = \nabla f(x_{k+1}) - \nabla f(x_k)$ erfüllen, die man aus der Taylor-Entwicklung von $\nabla f(x_{k+1})$ erhält. Im Weiteren führt das Ersetzen der zweiten Ableitung durch eine finite Differenz und einige Nebenbedingungen zu einer Optimierungsmethode. Diese benutzt das Verhalten der zweiten Ableitung, ohne die Hesse-Matrix wirklich berechnen zu müssen (für mehr Details dazu siehe [33]). Insgesamt ergibt sich also eine stabilere Methode, die gleichzeitig schneller konvergiert. [33] [34]

2.5.3 *Scores* auf der Basis der Topologie des Graphen

Link Prediction gehört zum Bereich der rechnerischen Analyse eines Netzwerks, bei dem die Knoten Personen oder Entitäten und die Kanten Relationen repräsentieren. Diese Netzwerke sind dynamisch und verändern sich über die Zeit. Das *Link Prediction*-Problem beschäftigt sich mit einem Ausschnitt eines solchen Netzwerks zu einem Zeitpunkt t und stellt die Frage nach möglichst akkuraten Vorhersagen für Kanten, die zum Zeitpunkt t noch nicht existieren und bis zu einem später liegenden Zeitpunkt t' hinzukommen. Dabei spielt unter anderem die netzwerkeigene Topologie eine entscheidende Rolle. Um diese Topologie quantifizieren zu können, werden verschiedene Nachbarschaftsmaße aus der Graphentheorie und ihre relative Effektivität untersucht. [35]
In [36] wird für das Maß dieser Effektivität ein sogenannter *Score* herangezogen, der sich unterschiedlich errechnen lässt. Beispiele hierfür sind:

- *Common Neighbours*: Für einen Graphen $G = (V, E)$ beschreibt $Score(x, y) := |\Gamma(x) \cap \Gamma(y)|$ die gemeinsamen Nachbarn zweier Knoten $x, y \in V$. [35] [36]
- *Preferential Attachment*: Gegeben sei erneut ein Graph $G = (V, E)$. Die hierbei zugrundeliegende Prämisse ist die Annahme, dass die Wahrscheinlichkeit, dass eine neue Kante den Knoten $x \in V$ enthält, proportional zu $|\Gamma(x)|$ ist. Da das Maß ursprünglich für die Vorhersage zukünftiger Kollaboration zweier Autoren erdacht

war, ergibt sich somit der $Score(x,y) := |\Gamma(x)| \cdot |\Gamma(y)|$. Dies baut auf der Idee auf, dass Knoten mit vielen Kanten eine höhere Wahrscheinlichkeit für noch mehr Kanten haben. [35] [36]

- *Adamic/Adar*: Der hier gefundene Koeffizient gibt ursprünglich ein Maß dafür an, dass zwei Homepages stark verbunden sind. Dafür werden Features z aus einer Feature-Grundmenge F der beiden Knoten, hier Webseiten, berechnet und die Gemeinsamkeit definiert:

$$\sum_{z:feature\ shared\ by\ x,y} \frac{1}{log(frequency(z))} \tag{2.1}$$

  Dadurch werden häufigere Features weniger stark gewichtet als seltenere. Wenn Features außen vor gelassen und nur die Topologie des Graphen betrachtet werden soll, wird für zwei Knoten $x,y \in V$ eines Graphen $G = (V,E)$ der folgende *Score* verwendet:

$$Score(x,y) := \sum_{z \in \Gamma(x) \cap \Gamma(y)} \frac{1}{log(|\Gamma(z)|)} \tag{2.2}$$

  [35] [36]

Die hier bereits angesprochenen *Scores* lassen sich auch in der Graphdatenbank *Neo4j* verwenden. *Neo4j* stellt dafür mit seiner *Neo4j Graph Data Science Library* verschiedene Algorithmen zur Verfügung. Zusätzlich zu den oben bereits genannten Methoden zur Ermittlung eines *Scores* werden noch folgende Funktionen bereitgestellt:

- *Resource Allocation*: In [37] wird erstmals *Ressource Allocation* als neues Ähnlichkeitsmaß für zwei Knoten eingeführt. Dafür werden ein Graph $G = (V,E)$ und zwei nicht benachbarte Knoten $x,y \in V$ betrachtet. Der Knoten x kann dem Knoten y Ressourcen übersenden und die gemeinsamen Nachbarn von x und y dienen als Transmitter. Zur Vereinfachung wird in [37] angenommen, dass jeder dieser Transmitter $u \in V$ eine Einheit der Ressource übermittelt und diese gleichmäßig auf alle Nachbarn $\Gamma(u)$ aufteilt. Die Ähnlichkeit von x und y wird dabei als Summe aller Ressourcen, die y von x erhält, definiert:

$$Score(x,y) = \sum_{z \in \Gamma(x) \cap \Gamma(y)} \frac{1}{|\Gamma(z)|} \tag{2.3}$$

  Dieser Index weist trotz unterschiedlicher Herkunft große Ähnlichkeit mit *Adamic/Adar* auf, unterscheidet sich aber für große Werte von $\Gamma(z)$ durch das lineare Wachstum im Nenner im Gegensatz zum logarithmischen Wachstum bei *Adamic/Adar*. [37]

- *Same Community*: Dieser Algorithmus bestimmt, ob zwei Knoten der gleichen

Community (siehe Definition 2.1.12.) angehören. Der zurückgegebene Wert, 1 oder 0, zeigt an, ob die beiden Knoten der gleichen Community angehören oder nicht. [38]

- *Total Neighbors*: Sei $G = (V, E)$ ein Graph. Die Nähe zweier Knoten $x, y \in V$ wird auf Basis der Anzahl ihrer Nachbarn $\Gamma(x), \Gamma(y)$ berechnet. Dabei wird jeder Nachbar, unabhängig von der Kantenanzahl zwischen ihm und dem ursprünglichen Knoten, nur einfach gewertet:

$$TN(x, y) = |\Gamma(x) \cup \Gamma(y)| \tag{2.4}$$

  Offensichtlich bedeutet ein Wert von 0 keine Nähe der beiden Knoten x, y. Je höher der Wert für TN wird, desto näher sind sich die Knoten. [38]

Für diese Maße lässt sich in *Neo4j* ebenfalls festlegen, welche Kanten der unterliegende Algorithmus berücksichtigen soll. Es ist also möglich, alle Kanten zu verwenden oder auch nur eine Teilmenge. [36] Diese Maße fallen unter die Rubrik *Methoden, die auf Knotennachbarschaft beruhen*. [35]
Sie werden in [36] auf zwei Arten als Grundlage der *Link Prediction* innerhalb des dort verwendeten Graphen präsentiert. Zum einen besteht die Möglichkeit, das Hinzufügen einer neuen Kante davon abhängig zu machen, ob der oben genannte *Score* eine zuvor festgelegte Schranke überschreitet. Falls dies der Fall ist, wird die Kante hinzugefügt. Zum anderen können die *Scores* mit *Supervised Learning* kombiniert werden: Sie werden als Features verwendet, um einen binären Klassifikator zu trainieren. Dieser sagt dann voraus, ob ein betrachtetes Knotenpaar mit hoher Wahrscheinlichkeit in der Zukunft durch eine Kante verbunden sein wird. Um den Klassifikator zu trainieren und zu bewerten, wird der verwendete Graph in Trainings-, Test- und Validierungsmenge unterteilt. Dann wird innerhalb des Trainingsgraphen trainiert und das Resultat auf den Testgraphen angewendet. Bei der Validierung zeigen sich für den dortigen Anwendungsfall vielversprechende Ergebnisse. [36] Mit dieser Arbeit wird, wie später erläutert wird, ein anderer Ansatz verfolgt, bei dem aber unter anderem auch diese *Scores* als Features oder als Kriterium zur Wahl eines Labels zum Einsatz kommen.
In [35] werden noch weitere Methoden betrachtet. Diese beruhen auf der Gemeinsamkeit aller Pfade oder werden als Meta-Herangehensweisen bezeichnet. Wegen des begrenzten Rahmens der Arbeit und der später noch präsentierten begrenzten Verfügbarkeit der Algorithmen in *Neo4j* wird in dieser Arbeit darauf nicht weiter eingegangen.

2.5.4 *Link Prediction* für Pfade anhand von Knotenattributen

Die in dieser Arbeit verwendete Herangehensweise bedient sich der *Conditional Random Fields*. Daher wird hier kurz ihre Entstehung untersucht und eine Einführung gegeben.

Markov-Prozesse

Als erstes werden einfache Markov-Prozesse n-ter Ordnung betrachtet. Dabei soll die Wahrscheinlichkeit für das Eintreten zukünftiger Zustände berechnet werden können. Die Ordnung gibt an, von wie vielen vorangegangen Zuständen der nächste abhängt. Bei einem Markov-Prozess 1. Ordnung hängt der Folgezustand also immer nur vom aktuellen Zustand ab. Zu Beginn befindet sich das System im Startzustand. [6]

Definition 2.5.1 *Unter einem Markov-Prozess wird ein Tupel* (S,A,δ) *verstanden. Dabei beschreibt S die endliche Zustandsmenge, A die Aktionsmenge und* δ *die Zustandsübergangsfunktion. [6]*

Für jedes Paar (s_t,a_t) mit $s_t \in S,\ a_t \in A$ wird mit $\delta(s_t,a_t)$ in den Zustand s_{t+1} gewechselt. Die Übergänge sind dabei in der Regel in Wahrscheinlichkeiten gegeben. Die Wahl der Aktion hängt vom aktuellen Zustand ab und lässt sich als Funktion $\pi : S \to A; \pi(s_t) = a_t$ darstellen. Sie wird auch Strategie genannt. [6]

Hidden-Markov-Modelle

Hidden-Markov-Modelle werden für die Repräsentation von Wahrscheinlichkeitsverteilungen über Sequenzen von Beobachtungen verwendet. Dabei wird zwischen der Beobachtung X_t und dem Zustand Z_t zum Zeitpunkt t unterschieden. Letzterer ist versteckt, daher auch der Name des Modells. Hier wird, wie in den 1-stufigen Markov-Ketten, die sogenannte Markov-Eigenschaft vorausgesetzt: Z_t zum Zeitpunkt t hängt nur von Z_{t-1} zum Zeitpunkt $t-1$ ab. Ein Beispiel dafür kann in Abbildung 2.3 betrachtet werden.
Der Zeitpunkt t muss keine explizite Zeitangabe sein und kann auch implizit als Ort innerhalb der Sequenz betrachtet werden. Die gesamte Wahrscheinlichkeitsverteilung einer Sequenz von Zuständen und Beobachtungen kann folgendermaßen als Gleichung ausgedrückt werden:

$$P(Z_{1:N},X_{1:N}) = P(Z_1)P(X_1|Z_1)\prod_{t=2}^{N} P(Z_t|Z_{t-1})P(X_t|Z_t) \tag{2.5}$$

Da die Zustände versteckt sind und nur die Beobachtungen betrachtet werden, die wiederum von den Zuständen abhängen, wird die Wahrscheinlichkeit einer N-elementigen Sequenz durch ein Produkt von bedingten Wahrscheinlichkeiten dargestellt. Überdies hängt, bis auf den Ausgangszustand, jeder Zustand von dem vorherigen ab. [39] [40]

Nach [39] und [40] gibt es fünf Elemente, die ein Hidden-Markov-Modell charakterisieren:

- Die Anzahl K der Zustände, die im Modell angenommen werden können. Die Zustände werden als $K \times 1$-Vektoren mit Binärwerten dargestellt, so dass der k-te Zustand zum Zeitpunkt t in der k-ten Zeile den Wert 1 und überall sonst 0 annimmt.

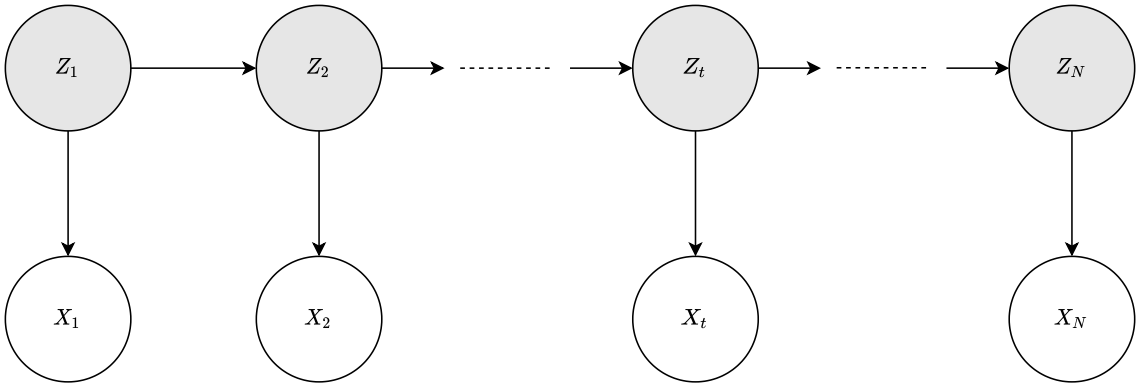

Abbildung 2.3: Beispiel für ein Hidden-Markov-Modell: Z_t beschreibt den Zustand und X_t die davon abhängige Beobachtung zum Zeitpunkt t.

- Die Anzahl Ω der verschiedenen Beobachtungen, die im Modell beobachtet werden können. Analog zu den Zuständen wird ein $\Omega \times 1$-Vektor verwendet.

- Das Zustandsübergangsmodell A: Dies wird auch Zustandsübergangswahrscheinlichkeitsverteilung genannt und beschreibt die Wahrscheinlichkeit, dass von einem Zustand $Z_{t-1,i}$ in einen Zustand $Z_{t,j}$ innerhalb eines Zeitschritts gewechselt wird. Dabei sind $i, j \in \{1, ..., K\}$. Dies kann wie folgt formuliert werden:

$$A_{i,j} = P(Z_{t,j} = 1 | Z_{t-1,i} = 1) \tag{2.6}$$

  Jede Zeile von A summiert sich dabei zu 1 auf.

- Das Beobachtungsmodell B ist eine $\Omega \times K$-Matrix, deren Elemente $B_{j,k}$ die Wahrscheinlichkeit angeben, die Beobachtung $X_{t,k}$ unter der Voraussetzung des Zustands $Z_{t,j}$ zu machen:

$$B_{j,k} = P(X_t = k | Z_t = j) \tag{2.7}$$

- Die anfängliche Zustandsverteilung π ist ein $K \times 1$-Vektor mit $\pi_i = P(Z_{1,i=1})$.

Das Modell wird in der Literatur oft als $\lambda = (A, B, \pi)$ abgekürzt. [39] [40]

Markov Random Fields

Sei $G = (V, E)$ ein ungerichteter Graph. Die Knoten $v \in V$ entsprechen den Zufallsvariablen, welche die Zustände annehmen können. Diese hängen dabei nur von den Zuständen der Zufallsvariablen u ihrer Markov-Decke $B_v = \{u : (v, u) \in E\}$ ab. Dies drückt sich in folgender Gleichung aus:

$$P(x_1, ..., x_n) = \frac{1}{Z} \prod_{c \in C} F_c(x_c) \tag{2.8}$$

C ist dabei die Menge der maximalen Cliquen des Graphen. Die Funktionen F sind nichtnegativ und hängen von den Variablen innerhalb der Clique c ab. Zur Normalisierung wird eine Funktion $Z = \sum_{x_1,...,x_n} \prod_{c \in C} F_c(x_c)$ verwendet, so dass die Verteilung sich insgesamt zu 1 aufsummiert. [41] [42]

Conditional Random Fields

Conditional Random Fields stellen einen besonderen Fall der Markov Random Fields dar und gehören zum Bereich des überwachten Lernens. Statt nur die Wahrscheinlichkeit für eine Labelsequenz y zu betrachten, wird hier die durch eine Beobachtungssequenz x bedingte Warhscheinlichkeit einer Labelsequenz y bestimmt:

$$P(y|x) = \frac{1}{Z(x)} \prod_{c \in C} F_c(x_c, y_c) \ , \ Z(x) = \sum_{y \in Y} \prod_{c \in C} F_c(x_c, y_c) \tag{2.9}$$

Auch die Normalisierungsfunktion *Z(x)* hängt jetzt von x ab. [42]

In anderer Literatur findet sich als Definition eines *(linear chain) Conditional Random Fields* die bedingte Wahrscheinlichkeit

$$p(y_{1:n}|x_{1:n}) = \frac{1}{Z} exp(\sum_{n=1}^{N} \sum_{i=1}^{F} \lambda_i f_i(y_{n-1}, z_n, x_{1:N}, n)) \tag{2.10}$$

Innerhalb der Exponentialfunktion wird erst über $n = 1,...,N$ summiert, was die Position eines Wortes, oder hier eines Knotens, innerhalb der Sequenz angibt. Die zweite Summe iteriert über die durch die Skalare λ_i gewichteten Features f_i, $i = 1,...,F$. Die Werte für die Gewichte müssen vorgegeben oder durch das *CRF*-Modell gelernt werden. Sie sorgen dafür, dass gewisse Labels bevorzugt oder auch vermieden werden. [43]
Für eine gegebene Sequenz können mehrere Features gleichzeitig aktiv sein, also ungleich 0. Dabei spricht man von überlappenden Features. Dies kann passieren, da, anders als bei Hidden-Markov-Modellen, auch auf darauffolgende oder vorherige Elemente der Sequenz geschaut werden kann. [43]
Zum Trainieren werden vollständig gelabelte Trainingssequencen $\{(x^{(1)}, y^{(1)}), ..., (x^{(m)}, y^{(m)})\}$ benötigt, wobei $x^{(i)} = x^{(i)}_{1:N_i} \ \forall \ i \in \{1,...,m\}$. Es wird also die bedingte Wahrscheinlichkeit der Trainingsdaten maximiert:

$$\sum_{j=1}^{m} logp(y^{(j)}|x^{(j)}) \tag{2.11}$$

Dies wird standardmäßig mithilfe von Algorithmen, die das *Gradient-Descend*-Verfahren verwenden, berechnet. [43]

Umsetzung der *CRFs* in Python

In Python wird für *CRFs* unter anderem die *sklearn-crfsuite* verwendet [44]. Sie baut auf der *scikit-learn*-Bibliothek für maschinelles Lernen auf [45]. Die Idee besteht darin, *Link Prediction* mithilfe von *NER* (*Named Entity Recognition*) zu verwenden. In [44] wird anhand eines Tutorials die Funktionsweise genauer erläutert. Bei den dort zugrunde liegenden Daten handelt es sich um einen Datensatz der CoNLL (Conference on Computanional Natural Language Learning, [46]) aus dem Jahr 2002. Unter *NER* werden dabei Sätze verstanden, die Namen von Personen, Organisationen, Orten, Zeiten und Mengen enthalten. Als Beispiel wird in [47] der folgende Satz angeführt: „*[PER Wolff], currently a journalist in [LOC Argentina], played with [PER Del Bosque] in the final years of the seventies in [ORG Real Madrid].*“. Der Satz enthält verschiedene benannte Entitäten, die in eckigen Klammern mit ihrer Bezeichnung versehen wurden. Diese Daten wurden zum Training eines *NER*-Systems verwendet, das anschließend auf einem zweiten Datensatz getestet wurde. [47]
In [44] wurde auf Grundlage dieser Daten ein beispielhaftes Trainingsmodell entwickelt. Dieses orientiert sich an vorher festgelegten Features für die einzelnen Worte innerhalb der Sätze. Diese wurden im Vorhinein als Listen eingelesen. Jedes Wort ist in den Trainingsdaten mit der passenden Annotation versehen. Danach findet das eigentliche Training anhand der Features und die anschließende Evaluation statt. Dafür werden durch die *sklearn-crfsuite* entsprechende Funktionen zum Lernen der Features bereitgestellt. Das gelernte Modell kann dann auf den Testdaten angewendet werden. [44]

Um die Güte der Vorhersage beurteilen zu können, wird der *F1-Score* (auch *balanced F-Score* oder *F-measure*) herangezogen. Dieser kann als gewichteter Durchschnitt der Präzision und des *Recalls* betrachtet werden. Dabei ist der beste Wert 1 und der schlechteste 0. Die dafür verwendete Formel ist:

$$F1 = 2 \cdot \frac{Präzision \cdot Recall}{Präzision + Recall} \tag{2.12}$$

Dabei gilt:

$$Präzision = \frac{wahre\ positive\ Resultate}{alle\ positive\ Resultate} \tag{2.13}$$

und:

$$Recall = \frac{wahre\ positive\ Resultate}{alle\ Stichproben,\ die\ als\ positiv\ hätten\ identifiziert\ werden\ sollen} \tag{2.14}$$

In Abbildung 2.4 wird die Entwicklung des *F1-Scores* in Abhängigkeit der Entwicklung der beiden Parameter Präzision und *Recall* gezeigt. [48] [49] Die Idee des zweiten Teils der Arbeit besteht darin, dieses Modell auf den Graphen zu übertragen. Anstelle der im Tutorial benutzten Sätze werden Pfade innerhalb des Graphen verwendet. Die Kurve in

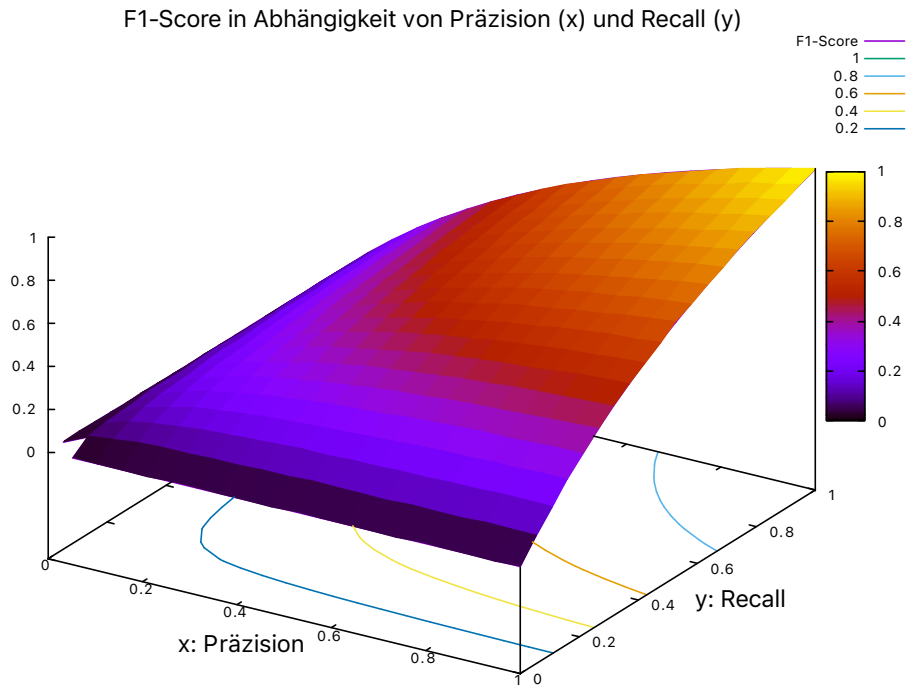

Abbildung 2.4: Entwicklung des *F1-Scores* in Abhängigkeit der Entwicklung von Präzision und *Recall*.

Abbildung 2.4 zeigt, dass der *F1-Score* auch steigen kann, wenn einer der beiden anderen Werte sinkt. Dies wird später in Abschnitt 4.2 noch auffallen. Die genauere Umsetzung in der Programmierung wird in Abschnitt 3.2 beschrieben.

Kapitel 3

Praktische Umsetzung

Nachdem im vorangegangenen Kapitel alle für diese Arbeit relevanten Definitionen der Bereiche Graphentheorie, Komplexitätstheorie und *Link Prediction* dargelegt wurden, sollen diese nun verknüpft werden. Es werden dabei mehrere Ziele verfolgt. Es soll erstens für Daten bildgebender Verfahren, in diesem Fall *DICOM*-Daten, ein generischer *Importer* für die Überführung in einen Graphen geschrieben werden. Dieser soll im Hinblick auf praktische Anwendung mit minimaler theoretischer Komplexität und praktischer Laufzeit arbeiten und zudem durch den Benutzer anpassbar sein. Zu diesem Zweck wird vor der Erstellung des *Importers* der Aufbau der Daten im folgenden Unterkapitel genauer untersucht. Das zweite Ziel ist die Anwendung der *Conditional Random Fields* auf den Graphen, um neue Kanten aufgrund der vorhandenen Zusammenhänge vorhersagen zu können. Dabei spielt die Form der Eingabedaten eine besondere Rolle, weshalb zunächst Pfade mit nur einem Knoten und anschließend Pfade mit mehreren Knoten betrachtet werden. Am Ende sollen die Ergebnisse verschiedener Variationen der Features und Labels untersucht und die Resultate präsentiert werden. Aus Gründen der Datensicherheit wird mit öffentlich zugänglichen Daten gearbeitet. Diese sind an den entsprechenden Stellen der Arbeit vermerkt.

3.1 *Importer* für klinische Daten aus bildgebenden Verfahren

3.1.1 Aufbau der *DICOM*-Dateien

Der *DICOM*-Standard gehört zu dem in Abschnitt 2.4 bereits allgemein beschriebenen Format. Da die Arbeit sich auf diesen konzentriert, wird hier weiter darauf eingegangen und das dem später folgenden Graphen zugrunde liegende Datenschema daraus abgeleitet.
DICOM-Dateien zeichnen sich durch ihre inhärente Baumstruktur aus. Diese besteht aus

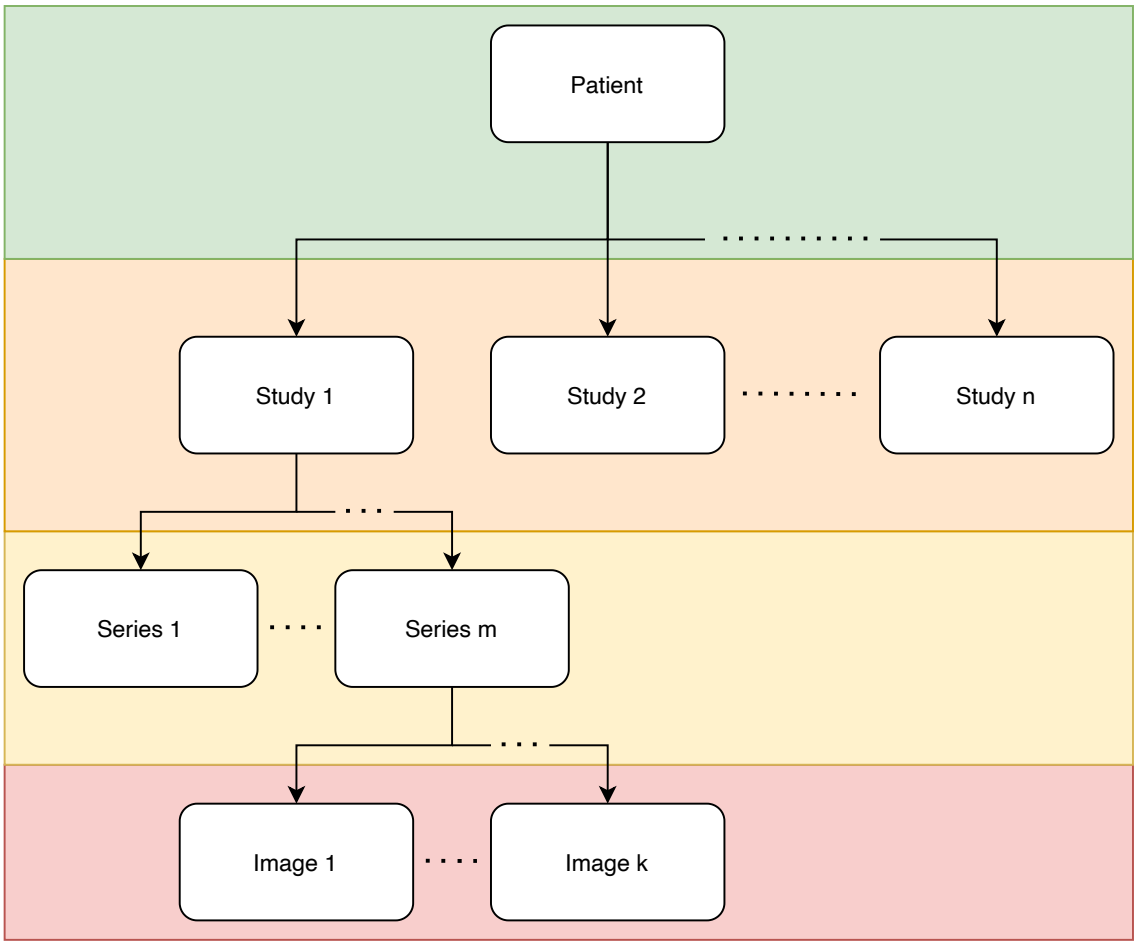

Abbildung 3.1: Struktur der *DICOM*-Daten. Der Patient stellt die oberste Stufe (grün) dar. Von dort aus verzweigt sich der Baum bis zur Ebene der eigentlichen Bilder (rot).

einer mehrschichtigen Hierarchie sowie verschiedenen Klassen und Klassenbezeichnungen. In der obersten Ebene wird der Patient mit einer eindeutigen Identität hinterlegt. Darunter können dann verschiedene Studien angeordnet werden, die selbst wiederum verschiedene Serien enthalten können. Jede Serie beinhaltet ihre zugehörigen Bilder. Daraus ergibt sich ein vierstufiges System, dessen Aufbau in Abbildung 3.1 visualisiert ist. [26]
Um nicht wahllos Datenelemente zu verknüpfen, wird eine Blockstruktur innerhalb des *DICOM*-Standards vorgegeben. Es werden *Information Modules*, *Information Entities* (IE) und *Information Object Definitions (IOD)* unterschieden. Die Module formen IEs, die dann wiederum *IODs* bilden. Dabei sind alle Module entweder *mandatory* (M), also unbedingt erforderlich, oder *conditional* (C), also hängt ihre Erforderlichkeit von weiteren Gegebenheiten ab, oder *user optional* (U), also dem Benutzer überlassen. Die Einordnung in diese drei Kategorien erfolgt anhand der sogenannten *image modality*. Ein Beispiel dafür ist das in dieser Arbeit vorwiegend verwendete *CT Image*. [26]
Den Modulen untergeordnet sind die einzelnen Datenelemente, die Attribute. Auch diese können, je nach vorliegenden Daten, in verschiedene Typen unterteilt werden:

- *Type 1 Required Data Elements*: Hierunter fallen die Datenelemente, die sowohl

vorhanden als auch mit einem nicht leeren Wert versehen sein müssen.

- *Type 1C Conditional Data Elements*: Diese Elemente werden nur unter bestimmten Bedingungen (engl. *conditions)* berücksichtigt. Sind diese jedoch erfüllt, gelten hier die gleichen Bestimmungen wie für *Type 1 Required Data Elements*.
- *Type 2 Required Data Elements*: Hierunter fallen alle Datenelemente, die zwar zwingend erforderlich sind, allerdings einen leeren oder keinen Wert besitzen können.
- *Type 2C Conditional Data Elements*: Diese Typen sind analog zu *Type 1C* zu betrachten. Falls die notwendigen Bedinungen erfüllt sind, sind auch sie verpflichtend, allerdings mit nicht notwendigerweise nicht-leerem Wert.
- *Type 3 Optional Data Elements*: Diese letzte Gruppe fasst alle Datenelemente zusammen, die optional sind. Sie können weggelassen werden, ohne den Standard zu verletzen. Sie können allerdings auch mit einem leeren Wert hinzugefügt werden.

Die genauen Bestimmungen, welche Module unter die Rubriken M, C und U und welche Datenelemente unter welchen Typ fallen, sind unter [50] zu finden. Das Lexikon listet die verschiedenen *IODs* auf und verweist darunter, ebenfalls in einer Baumstruktur, auf die einzelnen Module und innerhalb der Module wiederum auf die zugehörigen Attribute. Alle Datenelemente sind nicht nur über eine eindeutige standardisierte Bezeichnung abzurufen, sondern auch über einen eindeutigen *Tag*, der jedem Datenelement zugeordnet wird. Dies kann später auch innerhalb des *Importers* zur Abfrage der Attributswerte genutzt werden. [51] [26]
Weitere und genauere Informationen zum *DICOM*-Standard können unter [52] nachgelesen werden. Diese stehen aber nicht im primären Fokus dieser Arbeit und werden daher hier nicht weiter vertieft.

3.1.2 Benutzerspezifische Konfiguration

Wie bereits in Abschnitt 2.4 und in Unterabschnitt 3.1.1 beschrieben wurde, werden bei der Verwendung bildgebender Verfahren innerhalb der angelegten Dateien zusätzliche Informationen gespeichert. Im Rahmen dieser Arbeit wird hauptsächlich mit Daten aus der Radiologie, genauer gesagt aus der Computertomographie, gearbeitet. Die Daten selbst stammen aus der *SPIE-AAPM Lung CT Challenge* des *Cancer Imaging Archive*s [53].
Die Idee des *Importers* liegt in der generischen Übertragbarkeit auf andere Daten und einer benutzerspezifische Betrachtung und Konfiguration. Je nach gegebenem Kontext und vorhandenen Daten soll die Arbeit mit individuellen Daten und Graphen ermöglicht werden. Aus diesem Grund geht dem *Importer* eine *config*-Datei voraus. Diese ist eine *ini*-Datei, in der sich nach vorgegebenem Schema festlegen lässt, welche Knoten vom

Benutzer gewünscht sind und welche Attribute und Relationen sie haben sollen. Darüber hinaus eventuell in den Daten enthaltene Informationen werden dann beim Import nicht berücksichtigt.
Innerhalb der *config*-Datei werden die gewünschten Knoten als *Sections* in eckigen Klammern deklariert. Darunter werden dann jeweils die folgenden *key-value* Paare gespeichert, wobei für die vorliegende Arbeit beispielhaft mit einer selbst gewählten Konfiguration gearbeitet wird:

- *relationsto*: Hier werden alle *Sections* der Knotentypen angegeben, zu denen die Knoten der gegebenen *Section* im zu erstellenden Graphen Kanten haben sollen.

- *relationtypes*: Unter diesem *key* werden alle Relations-Typen angegeben. Dabei korreliert der $i-$te Relations-Typus mit der Kante zum $i-$ten in der darüber liegenden Zeile angegebenen Knoten. Zur Erläuterung kann Tabelle 3.1 herangezogen werden. Dort gibt es in der *Section Patient* unter *relationtypes* beispielsweise den Kantentypus `hasStudy`, der zum Eintrag `General Study` in *relationsto* gehört. Dabei orientiert sich die Benennung der Relationen am *Dublin-Core-Standard* (vgl. [54]).

- *attributes*: Alle Attribute, die der aktuell betrachtete Knoten erhalten soll, sind hier aufgelistet. Die spezielle Bezeichnung folgt den Vorgaben des *DICOM*-Standards (vgl. [55]).

- *class*: Es wird die Knotenklasse hinterlegt. Wie in Abschnitt 2.4 genauer erläutert wurde, ist diese durch den *DICOM*-Standard festgelegt. Darunter werden beispielsweise *IOD*-Module oder Attribute verstanden.

- *usage*: Ebenfalls in Abschnitt 2.4 beschrieben ist die *usage* eines *IOD*-Moduls. Über diese Eingabe werden die Knoten, die der *IOD*-Modul-Klasse angehören, mit dem entsprechenden *usage*-Knoten verbunden.

- *uid*: Hier ist hinterlegt, was als *unique identifier* für den Knoten verwendet werden soll. Bei Patienten wird dafür beispielsweise die *PatientID* genutzt.

- *uidfromds*: Das ist ein *Boolean*-Wert, der angibt, ob der in *uid* hinterlegte Wert direkt als *unique identifier* verwendet werden soll, wie z.B. bei `General Image`, oder der in Python eingelesenen Datei entnommen werden soll.

- *sequenceboolean*: Durch die in Abschnitt 2.4 beschriebene Baumstruktur der Header-Parts der *DICOM*-Dateien liegen Informationen teilweise verschachtelt in einer Sequenz vor. Um die Daten auszulesen wird hier für einen Knoten gespeichert, ob eine solche Sequenz berücksichtigt werden muss.

- *sequence*: Falls der *Boolean*-Wert für *sequenceboolean* auf 1 steht, wird hier angegeben, unter welchem Ast des Baums die Information gefunden wird. Eine andere Möglichkeit wäre eine zusätzliche Suche im *Importer*.
- *date*: Jede *DICOM*-Datei kann in ihren Header-Daten mehrere Daten enthalten, die zu unterschiedlichen Bereichen gehören, beispielsweise das Datum einer Studie oder eines Bildes. Falls für die entsprechende *Section* (bzw. für den entsprechenden Knoten) ein Datumsstempel vorgesehen ist, wird hier, mit einem Komma getrennt, angegeben, unter welchem Attribut der eingelesenen *DICOM*-Datei das zugehörige Datum gefunden werden kann und welcher Kantentyp dafür vorgesehen ist. Dies ist unter anderem relevant, weil ein Patient ein Geburtsdatum hat, eine Studie aber ein `StudyDate`. Falls kein Datum zum Knoten verwendet werden soll, kann es leer gelassen werden.
- *time*: Analog zum *date* wird hier optional ein Wert für eine Zeit hinterlegt. Ansonsten wird es ebenfalls leer gelassen.

```
 1  [Patient]
 2  relationsto = General_Study,Usage,Patient_Age,Patient_Weight
 3  relationtypes = hasStudy,is,hasAge,hasWeight
 4  attributes = PatientName,PatientIdentityRemoved,DeidentificationMethod
 5  class = IOD Module
 6  usage = M
 7  uid = PatientID
 8  uidfromds = 1
 9  sequenceboolean = 0
10  sequence =
11  date = PatientBirthDate,hasBirthDate
12  time =
```

Tabelle 3.1: Beispielhafter Ausschnitt aus der *ini*-Datei für den Patientenknoten.

In Tabelle 3.1 ist ein Ausschnitt der *config*-Datei für den Knoten `Patient` abgebildet. Die genaue Verwendung der einzelnen *key-value*-Paare wird in den folgenden Abschnitten erläutert. *DICOM*-Dateien enthalten auch *pixel data*. Es wird davon abgeraten, diese als *Section* in die Konfiguration mit aufzunehmen. Dieses *IOD*-Modul enthält einen von links nach rechts und von oben nach unten laufenden Datenstrom der Pixel des in der Datei enthaltenen Bildes [56]. Dies ist zum einen innerhalb eines Graphen nicht hilfreich, da es aufgrund der enormen Länge nicht sinnvoll darstellbar ist. Zum anderen wird aus dem gleichen Grund eine Ausgabe in Form der *CSV*-Dateien am Ende des *Importers* praktisch unmöglich. In der *pydicom*-Bibliothek wird noch die Funktion `Dataset.pixel_array` zur Verfügung gestellt, die es ermöglicht, statt der komprimierten Darstellung in einzelnen

Bytes die unkomprimierte Darstellung des Bildes in Form eines *Arrays* zu erhalten [57]. Das wird in dieser Arbeit allerdings nicht weiter verfolgt.

3.1.3 Datenschema

Das eigentliche Datenschema für den dieser Arbeit zugrunde liegenden Graphen wird in Abbildung 3.2 gezeigt. Hierfür wurden zunächst die verwendeten Daten unter Berücksichtigung der zuvor beschriebenen unterliegenden Datenstruktur betrachtet. Die gegebene Auswahl und Anordnung der einzelnen Knoten ist vom Autor als beispielhafte Instanz vorgenommen worden. Durch Anpassungen in der Konfigurationsdatei entstehen auch andere Schemata. Für den Graphen, der in dieser Arbeit verwendet wird, war jedoch die Entscheidung für ein Schema erforderlich. Wichtig war dabei zunächst, die vierstufige Hierarchie der *DICOM*-Daten beizubehalten. Dies ist in Abbildung 3.2 in dem mittleren Strang zu beobachten. Jeder Patient hat einen eigenen Knoten, der mit seinen Studien verknüpft ist. Diese wiederum enthalten die zugehörigen Serien, die dann die Bilder enthalten. Zusätzlich zu diesem Hauptstrang innerhalb des Schemas wurden dann weitere Informationen annotiert. Dabei wurde sich an den zuvor beschriebenen Typen orientiert. Alle Module, die als (M) klassifiziert wurden, wurden berücksichtigt. Darüber hinaus wurde allerdings auch aus den Klassen (C) und (U) mindestens ein Modul verwendet. Bei (C) ist es die Klasse `Contrast/Bolus`, bei (U) wurde `Patient Study` gewählt. Innerhalb des Datenschemas wurden die verwendeten *IOD*-Module, die eigene Knotengruppen bilden, zur Visualisierung gelb hinterlegt, die Attribute in gleichen Fällen rot. Die blaue Zeile *Node Group = True* impliziert, dass die darunter aufgeführten Knoten zu der Knotengruppe der Überschrift gehören. Diese werden im Graphen als Dreiecke dargestellt, die aber der Übersichtlichkeit halber hier als Knotengruppen abgebildet sind. Als Beispiel kann der Knoten `Manufacturer` betrachtet werden. Dieser gehört zur Knotengruppe `General Equipment` und schließt im Graphen mit dem Knoten `General Equipment` und dem Knoten `General Series` ein Dreieck. Ein Beispiel zeigt Abbildung 3.3. Im Unterschied dazu hat beispielsweise der Knoten `General Series` Attribute wie `Modality`, `Series Instance UID` und andere als knoteneigene Attribute gespeichert und nicht als separate Knoten.
Solche Spezifizierungen können allerdings über die Konfigurationsdatei frei gestaltet und modifiziert werden, wie bereits im Abschnitt zuvor erläutert wurde.
Zusätzlich zu den in den *DICOM*-Dateien enthaltenen Daten wurden noch zwei weitere Knoten hinzugefügt. `Source` gibt die Quelle der Daten an. Bei den gegebenen Daten ist diese unter dem *Tag* (0013, 1010) hinterlegt. `File` speichert den Dateinamen des Bildes und dient daher als eine Art *provenance*, womit die Knoten eindeutig einer jeweiligen Datei zugewiesen werden können. Damit die Knoten `Source` und `File` eingebunden werden können, ist bei der Konfiguration des *Importers* wichtig, dass der Patient im Graphen als

Knoten enthalten ist. Dies bedeutet zwar eine kleine Einschränkung im Sinne der freien Konfiguration, allerdings sollte ein solcher Graph ohne Patienten inhaltlich schwer zu rechtfertigen sein.

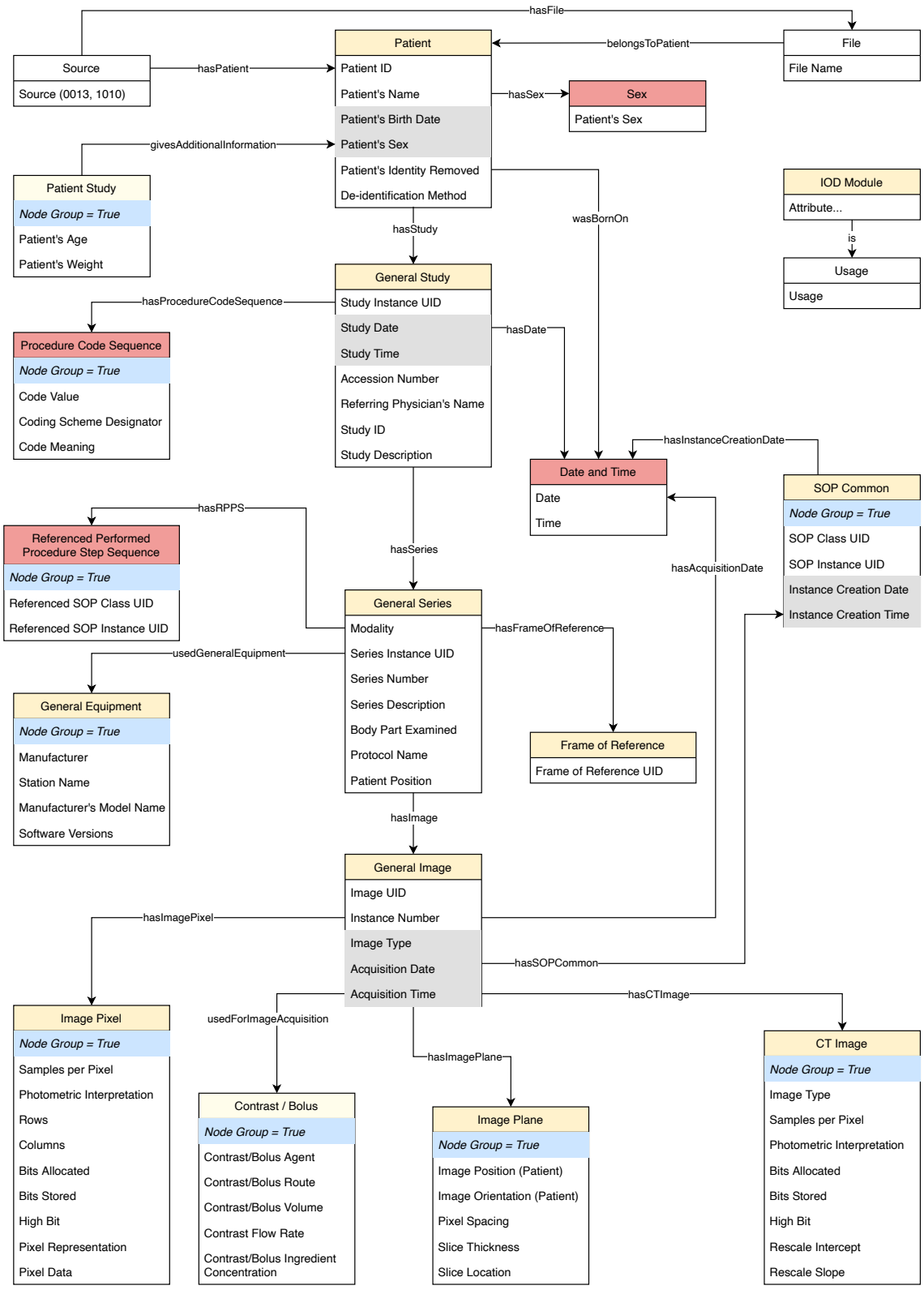

Abbildung 3.2: Datenschema für den Import der *DICOM*-Files.

Die grau unterlegten Knotenattribute wurden ausgelagert. Dies bezieht sich zum einen auf das Attribut `Image Type` des Knoten `General Image`, da sich dieses ebenfalls als eigener Knoten in der Knotengruppe `CT Image` befindet. Zum anderen betrifft es die Daten- und Zeitattribute der Knoten. Diese wurden in einen weiteren Knoten `Date and Time` ausgelagert, um dem Graphen eine einfachere, abrufbare zeitliche Struktur zu verleihen. Diese zusätzliche Dimension der Daten ermöglicht spezifischere Abfragen nach zeitlicher Reihenfolge.

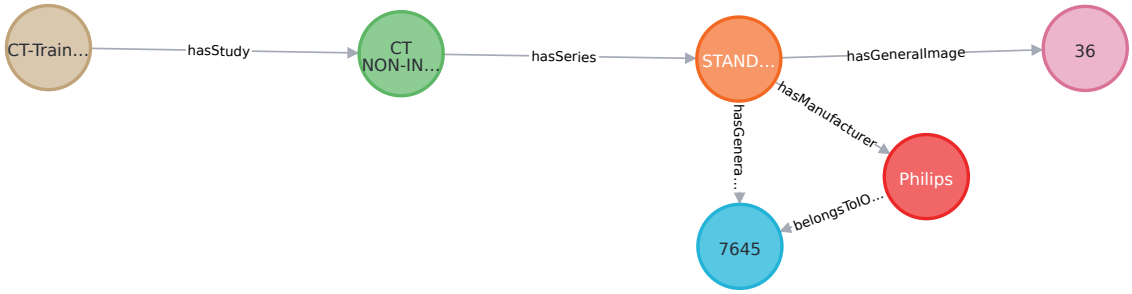

Abbildung 3.3: Teilausschnitt des Graphen: Beispielhaftes Dreieck im Graphen zwischen den genannten Beispielknoten `Manufacturer` (rot), `General Equipment` (blau) und `General Series` (orange).

Zunächst wird die *config*-Datei *dev.ini* mithilfe der *configparser*-Bibliothek (vgl. [58]) für Python eingelesen. Dann werden die *Sections* ausgelesen und in einer Liste gespeichert. Dies erleichtert später die Verwendung. Für jede dieser *Sections* werden dann die verschiedenen Relationstypen eingelesen und in einer Liste gespeichert. Diese wird noch um die *static relations*, also die Relationen, die durch das Programm vorgegeben werden, erweitert. Zusätzlich wird ein *Dictionary* von *Sets* angelegt, das zur Speicherung der Knoten-*uids* und zur Vermeidung von Knotendopplungen benötigt wird. Dafür bieten sich *Sets* an, da sie keine paarweise identischen Elemente zulassen.
Die Knoten `Sex` und `Usage` können schon im Vorhinein erstellt werden, da es sich um eine endliche und vordefinierte Anzahl handelt. Sie werden auch zu Beginn bereits als *CSV*-Dateien über die Methode `print_static(static_nodes, static_type)` ausgegeben.

```
# create "static" nodes (not dependant on input data)

# create sex nodes
class Sex:
    def __init__(self, s):
        self.ID = s
        self.Identifier = s

sexNodes = [Sex('M'), Sex('F')]

# create usage nodes
class Usage:
```

```
    def __init__(self, _usage):
        self.ID = _usage
        self.Identifier = _usage

usageNodes = [Usage('Mandatory'), Usage('Conditional'), Usage('UserOptional')]

[...]

def print_static (static_nodes, static_type):
     # create new dir if necessary
      export_nodes_to = '/Users/tobias/Library/Mobile Documents/com~apple~↩
          CloudDocs/Documents/Uni/Masterarbeit/Datengrundlage/Cancer Imaging ↩
          Archive/Import/nodes'

      if not os.path.exists(export_nodes_to):
          os.makedirs(export_nodes_to)

          os.chdir(export_nodes_to)

          # write file
          with open(f"{static_type}.csv", 'w') as csvfile:
           csv_node_writer = csv.writer(csvfile)
           for n in static_nodes:
              csv_node_writer.writerow([n.ID, n.Identifier])


print_static(usageNodes, 'Usage')
print_static(sexNodes, 'Sex')
```

Listing 3.1: Erstellung der *static nodes* `Sex` und `Usage`.

Darüber hinaus werden für die in Abbildung 3.2 weiß hinterlegten Knoten Klassen angelegt. Sie sind nicht in den eigentlichen *DICOM*-Files als *IOD*-Module oder Attribute enthalten und werden eigenständig hinzugefügt. Anschließend werden verschiedene Listen und *Sets* angelegt, die zur späteren Speicherung der erstellten Knoten dienen, bevor diese als *CSV*-Datei ausgegeben werden.

Unter *rootdir* wird das Verzeichnis angegeben, in dem die einzulesenden Dateien abgelegt sind. Über die *glob*-Bibliothek (vgl. [59]) wird dann im Folgenden auf alle Unterverzeichnisse zugegriffen.

Das Kernelement des Parsers bildet die doppelte *for*-Schleife, die über alle Elemente innerhalb des Verzeichnisses *rootdir* und darin jeweils über alle Dateien iteriert. Die verwendeten Daten liegen nach Patienten geordnet in Ordnerstrukturen vor. Durch die Variabilität des verwendeten Befehls aus der *glob*-Bibliothek lassen sich aber auch andere Unterverzeichnisse verwenden. Innerhalb der inneren Schleife wird zunächst sichergestellt, dass nur *DICOM*-Daten eingelesen werden. Anschließend werden für jede solche Datei nacheinander die Elemente aus dem Datenschema in Abbildung 3.2 ausgelesen.

Anfangs wird die eigentliche Datei eingelesen und die *provenance* erstellt. Diese dient der zeitlichen Eindeutigkeit der Zuordnungen und wird durch eine Kombination aus Patienten-ID und Dateiname für jede Kante generiert. Damit ist sie unter allen Daten eindeutig. Danach wird der `File`-Knoten erstellt, falls dieser noch nicht existiert. Gleiches gilt für

den `Source`-Knoten. Die Abfrage danach ist in eine *try-except*-Umgebung gekleidet, um möglichen Fehlern vorzubeugen. Falls das gesuchte Attribut nicht existiert, wird dieser Schritt übersprungen. Diese Vorgehensweise wird auch später noch verwendet und löst Probleme bei nicht vollständig einheitlichen Dateien.

```
try:
      sources.add(ds[0x013, 0x1010].value)
except KeyError:
      pass
```

Listing 3.2: Abfrage des `Source`-Attributes der Dateien über eine *try-except*-Umgebung.

Dann werden die über die Konfigurationsdatei festgelegten Knoten über eine weitere *for*-Schleife nacheinander betrachtet. Um Kanten zu sparen, wird die Variable *sex_boolean* angelegt. Durch die Annahme, dass ein Patient im Laufe seiner Untersuchungen das Geschlecht nicht wechselt, lassen sich für jeden Patienten redundante Kanten vermeiden. Aus einem ähnlichen Grund wird die Variable *date_boolean* erstellt: Falls das untersuchte Element eine Kante zu einem Datumsknoten erhalten soll, muss dies in der Konfigurationsdatei vermerkt werden. Dies wird später ausgelesen und dieser *boolean*-Wert dann auf 1 gesetzt.
Im Anschluss wird der Knoten `Usage` behandelt. Dazu nutzt man die entsprechende Angabe aus der Konfigurationsdatei, die bereits in Unterabschnitt 3.1.2 beschrieben wurde. Die eigentlichen `Usage`-Knoten wurden bereits erstellt, aber für die spätere Zuordnung wird die *usage* hier in *temp_usage* gespeichert. Genauso wird das Geschlecht des Patienten für die späteren Kanten in *temp_sex* gespeichert. Dann erfolgt die Abfrage, ob für die betrachtete *Section* ein Datumsknoten vorgesehen ist. Ist dies der Fall, werden die in Unterabschnitt 3.1.2 erläuterten abgefragt. Über den ersten dort gespeicherten Wert wird das Datum aus der *DICOM*-Datei ausgelesen. Analog wird mit der Zeit verfahren. Aus den beiden ausgelesenen Werten wird eine eindeutige ID für den Knoten `Date and Time` erstellt. Damit kann dann überprüft werden, ob der Knoten bereits existiert und später nur eine Kante erstellt werden soll, oder ob auch der Knoten kreiert und die ID als bereits erstellt abgespeichert werden muss.
Danach muss, wie zuvor beim Datum, für die eingelesene Datei und die entsprechende *Section* eine eindeutige ID (*uid*) erstellt werden, um Dopplungen zu vermeiden.
Der Knoten `General Image` wird separat betrachtet, um ihm eine eindeutige ID zuzuweisen, die durch eine Verkettung der `Series Instance UID` und der `Instance Number` entsteht. Dies ist sinnvoll, da ein Bild innerhalb der Baumstruktur des *DICOM*-Formats eine Instanz einer Serie ist (siehe Abschnitt 2.4).
Dann wird für alle anderen *Sections* die eindeutige ID gefunden. Falls innerhalb der Konfigurationsdatei für die jeweilige *Section* der Parameter $uidfromds = 1$ gesetzt ist, wird im *Importer* die Knoten-ID für den potentiell neu zu erstellenden Knoten der vorliegen-

den *Section* aus der zu dem Zeitpunkt eingelesenen *DICOM*-Datei ausgelesen. Dabei ist zu unterscheiden, ob es sich um einen Teil einer Sequenz handelt oder nicht. Liegt dieser Fall vor, wird über die entsprechenden Angaben aus der Konfigurationsdatei (siehe Unterabschnitt 3.1.2) die passende Sequenz an der entsprechenden Stelle ausgelesen. Falls es keine Sequenz ist, kann die eindeutige ID direkt ausgelesen werden. Dafür wird das *DICOM*-Attribut abgefragt, das in der Konfigurationsdatei unter *uid* hinterlegt ist. Um die Möglichkeit einer nicht vorhandenen Angabe auszuschließen, wird ein *KeyError* bei der Abfrage der *DICOM*-Dateien abgefangen.

```
if element == 'General_Image':
    uid = f"series{ds.SeriesInstanceUID}instance{ds.InstanceNumber}"

# check which Attribute shall be used as the unique ID for the node
elif parser[element]['uidfromds'] == '1':
    uidToTest = parser[element]['uid'] # gets back what shall be used as an uid
    if parser[element]['sequenceboolean'] == '0':
        try:
            uid = ds[uidToTest].value
        except KeyError:
            continue
    else:
        seq = parser[element]['sequence'].split(',')
        # list needs integer value
        try:
            uid = ds[seq[0]][int(seq[1])][uidToTest].value
        except KeyError:
            continue
else:
    uid = parser[element]['uid']
```

Listing 3.3: Abfrage der uid-Modalitäten und Erstellung der knotenspezifischen uid.

Nachdem die Knoten-ID erstellt wurde, wird über das *Dictionary* `UIDs{}` geprüft, ob der Knoten bereits vorhanden ist. Falls er das nicht ist, wird die Knoten-ID dem *Dictionary* hinzugefügt und der Knoten erstellt. Damit die Anzahl der Attribute des zu erstellenden Knoten variabel gehalten werden kann, wird an das Knotenobjekt ein *Dictionary* übergeben. Diesem wird zuerst die Knoten-*uid* übergeben. Dann wird aus der Konfigurationsdatei das *key-value*-Paar *attributes* für alle weiteren Knotenattribute verwendet. Es wird auch hier geprüft, ob es sich um Elemente einer Sequenz handelt und anschließend werden die zugehörigen Werte aus der Datei ausgelesen und in dem *Dictionary* gespeichert, das als Übergabeparameter für den zu erstellenden Knoten erzeugt wurde. Der erstellte Knoten wird anschließend in einer Liste für die spätere Ausgabe gespeichert.
Schließlich müssen noch die ausgehenden Kanten des zuvor betrachteten Knoten erstellt werden. Dafür wurde eine generische Klasse `Relation` erstellt, die einen Startknoten, einen Zielknoten, einen Relationstypen und die *provenance* speichert. Sie ist in Listing 3.4 abgebildet.

```
def __init__(self, _start, _end, _type, _prov):
    self.start_id = _start
    self.end_id = _end
    self.rel_type = _type
    self.provenance = _prov
```

Listing 3.4: Generische Klasse für Relationen.

Die *Section*-Bezeichnungen der jeweiligen Zielknoten sind in der Konfigurationsdatei unter *relationsto* angegeben. Falls dieser Teil der *Section* nicht leer ist, wird für jedes der Elemente analog zu vorher die Knoten-ID des Zielknotens gefunden. Dafür wird zunächst die *uid* des Zielknotens mithilfe einer Fallunterscheidung bestimmt. Diese bedient sich der gleichen Idee wie schon zuvor: Es werden potentielle Sequenzen berücksichtigt und je nach *Boolean*-Wert, der angibt, ob die Zielknoten-*uid* aus der *DICOM*-Datei ausgelesen werden soll oder explizit anders vorgegeben wurde, wird die *uid* bestimmt. Die Kanten, die zu den statischen Knoten, z. B. `File` des Datenschemas, führen, werden später betrachtet. Dann werden die Kanten aus den ausgelesenen *uids* zwischen dem Knoten der aktuell betrachteten *Section* und den Zielknoten-*uids* erstellt und in einer Liste für die spätere Ausgabe gespeichert. Die anfänglich erwähnte *provenance* kommt hier zum Tragen: Sie wird ebenfalls als Attribut in den erstellten Kanten gespeichert, um später die zeitliche Zugehörigkeit kenntlich zu machen.
Im Anschluss werden die statischen Knoten und die Kanten zu ihnen betrachtet. Die Relationen zu den Knoten `Date and Time` und `Sex` werden nur bei Bedarf erstellt und gespeichert, was über die zuvor genannten Boolean-Werte reguliert wird. Schließlich werden noch die Relationen zu den Knoten `File` und `Source` sowie die Relation zwischen `File` und `Source` generiert.
Der soeben beschriebene Hauptteil des Codes wird mit einer Zeitmessung versehen, die später in Abschnitt 4.1 genauer untersucht und in Verbindung mit der theoretischen Laufzeitanalyse gebracht wird.
Am Ende des *Importers* werden die gespeicherten Knoten und Kanten über dafür definierte Funktionen in *CSV*-Dateien gespeichert und diese in einem passenden Verzeichnis ausgegeben. Zusätzlich wird zu jedem Knoten- und Kantentyp eine Header-*CSV*-Datei erstellt. Beides zusammen ist notwendig für den Bulk-Import in *Neo4j*, da die Header-Dateien der Datenbank die Bezeichnung der Spalten in den *CSV*-Dateien, welche die eigentlichen Knoten und Kanten enthalten, vorgeben (siehe [38] unter der Rubrik *Neo4j Admin import*). Auch für die Ausgabe in *CSV*-Dateien wird die Zeit gemessen. Die Auswertung der dazu gefundenen Daten wird ebenfalls in Abschnitt 4.1 präsentiert.

3.1.4 Datenimport in *Neo4j*

Die Daten werden mithilfe des *Admin import tools* (vgl. [60]) von *Neo4j* importiert. Dies kann nur für eine zuvor leere Datenbank erfolgen und wird nur für das initiale Laden

der Daten in die Datenbank verwendet. Das Verfahren eignet sich für sehr große Datenmengen. Dafür wird zuerst aus dem *Neo4j Download Center* [61] der *Neo4j Community Server* Version 4.3.2 heruntergeladen. Dann werden für den Import jedes Knotens und jeder Relation nach der Anleitung unter [60] zwei verschiedene *CSV*-Dateien benötigt. Die erste enthält die eigentlichen Daten und die zweite stellt die Header-Datei dar. Letztere enthält die Spaltenüberschriften für erstere.
Diese *CSV*-Dateien wurden durch den *Importer* bereits erstellt. Sie werden unter dem Verzeichnis *$NEO4J_HOME/import* abgelegt. Dabei bezeichnet *$NEO4J_HOME* den lokalen Ordner, in dem sich der *Neo4j*-Server befindet. Über die Konsole wird im Anschluss in das entsprechende Verzeichnis navigiert und der Import-Befehl eingegeben. Dieser richtet sich nach dem Muster in [60] und ist unter `https://github.com/TbsHbnthl/master-s-thesis-link-prediction-on-large-scale-knowledge-graphs/tree/main/Neo4j_fertig` für die hier verwendeten Daten hinterlegt.
Die Funktionen aus 2.1.3 werden hierbei konkret betrachtet. ρ ordnet jeder Kante ein Knotenpaar zu. Dies spiegelt sich in den *CSV*-Dateien wider. Listing 3.5 zeigt einen beispielhaften Zusammenschnitt der Header-Datei *hasAge_header.csv* und eines Ausschnitts der zugehörigen *CSV*-Datei *hasAge.csv* der Kante `hasAge`. Die Knoten, die in der Datei unter *:START_ID* und *:END_ID* stehen, sind die beiden Knoten, die der Kante durch ρ zugeordnet werden.

```
:START_ID,:END_ID,type,provenance,relationUID
LUNGx-CT025,064Y,hasAge,LUNGx-CT0251-013.dcm,LUNGx-CT025hasAge064YLUNGx-CT0251-013.dcm
```

Listing 3.5: Beispiel aus den *CSV*-Dateien für Relationen.

Die Funktion λ weist den Knoten ein Label zu. Dies kann in dem oben genannten Import-Befehl nachvollzogen werden: Ein Knoten wird mit folgender Syntax importiert:

--nodes=label="HEADER_FILE_DIRECTORY,FILE_DIRECTORY"

Ein Beispiel hierfür ist der im *Git Repository* hinterlegte importierte Knoten, der das Label `Patient` erhält. Schließlich wird überdies die Funktion σ betrachtet, die den einzelnen Knoten ihre Attribute, also die *Properties* zuweist. Dies erfolgt schon im eigentlichen Python-*Importer*, wenn dem Knoten das *Dictionary* mit den Attributen als *key-value*-Paare übergeben wird. Nach dem Import können die Daten in der Graphdatenbank weiter verwendet und visualisiert werden. Ein Beispiel ist in Abbildung 3.4 abgebildet.

3.1.5 Kantenwiederholungen und zweiter *Importer*

Die erste Version des *Importers* führt zu einem Problem für die spätere Verwendung des Graphen. Jede eingelesene Datei enthält Daten wie z.B. die Patienten-ID, die zugehörige Studie und Serie. Da aber, wie in Abschnitt 2.4 beschrieben, zu einer Serie

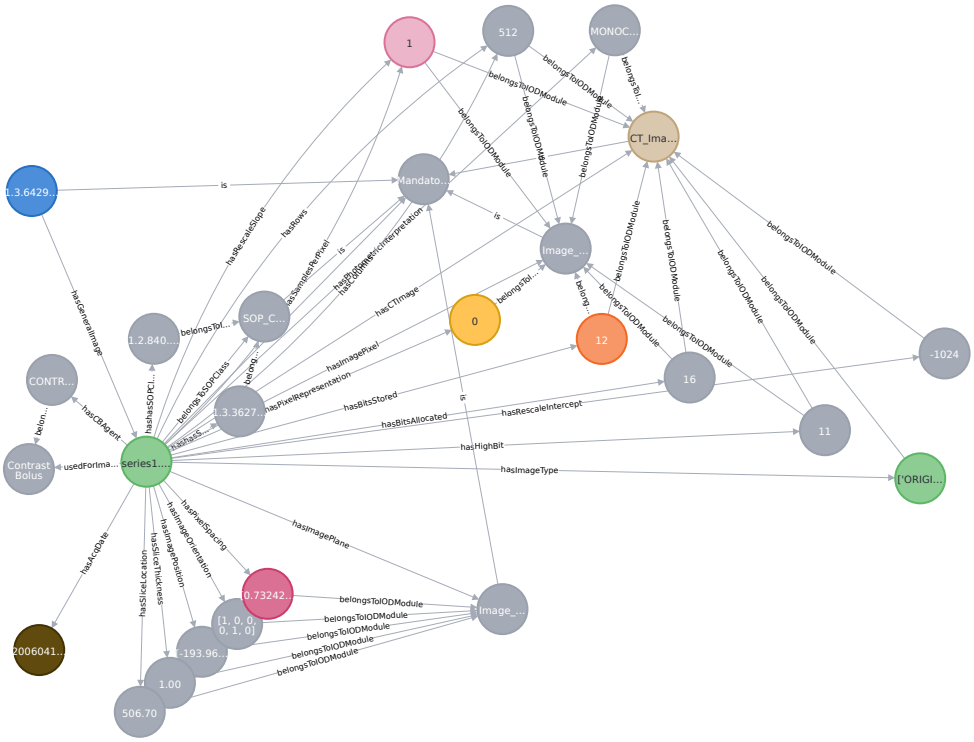

Abbildung 3.4: Ein Beispiel für die Knoten und Kanten in der Graphdatenbank nach abgeschlossenem Import: Es werden ein `General Image`-Knoten und seine direkte Nachbarschaft dargestellt.

einer Studie eines Patienten mehrere Bilder, also auch mehrere Dateien, gehören, entstehen wiederholte Relationen, die sich lediglich in der Zeit unterscheiden. Diese wird in den Relationen durch die *provenance* angegeben, die sich aus der Patienten-ID und dem Dateinamen zusammensetzt. In Abbildung 3.6 ist ein beispielhafter Ausschnitt des Graphen abgebildet, der das Problem zeigt. Die *Neo4j*-interne Navigation mithilfe von Querys wird damit erheblich erschwert oder sogar unmöglich. Dies zeigt sich auch in einer stark erhöhten Auslastung des Arbeitsspeichers. Um das Problem zu umgehen, wird eine zweite Version des *Importers* geschrieben, die bei derartigen Kantenwiederholungen nur noch die erste Kante in den eigentlichen Graphen integriert und alle weiteren Kanten der gleichen Art in einem separaten *Dictionary* speichert. Dieses wird am Ende in Form einer *CSV*-Datei ausgegeben. Die zweite Version unterscheidet sich nur marginal von der ersten. Der wichtigste Unterschied besteht darin, dass jede Kante vor der Erstellung eine eindeutige Kanten-ID zugewiesen bekommt. Diese setzt sich aus Start- und Zielknoten sowie der Art der Kante zusammen. Zum Beispiel wird unter der Kanten-ID `series1.3.139753033712679544934272869826037436076699699582 6instance13Mandatoryis,'LUNGx-CT0251-013.dcm'` die Relation der Art `is` zwischen Startknoten `series1.3.139753033712679544934272869826037436076699699582 6instance13` und Endknoten `mandatory` gespeichert. Innerhalb des *Dictionarys* werden diese Kanten-IDs als *keys* verwendet. Der dazugehörige *value* besteht aus einem *Set*, dem nach und

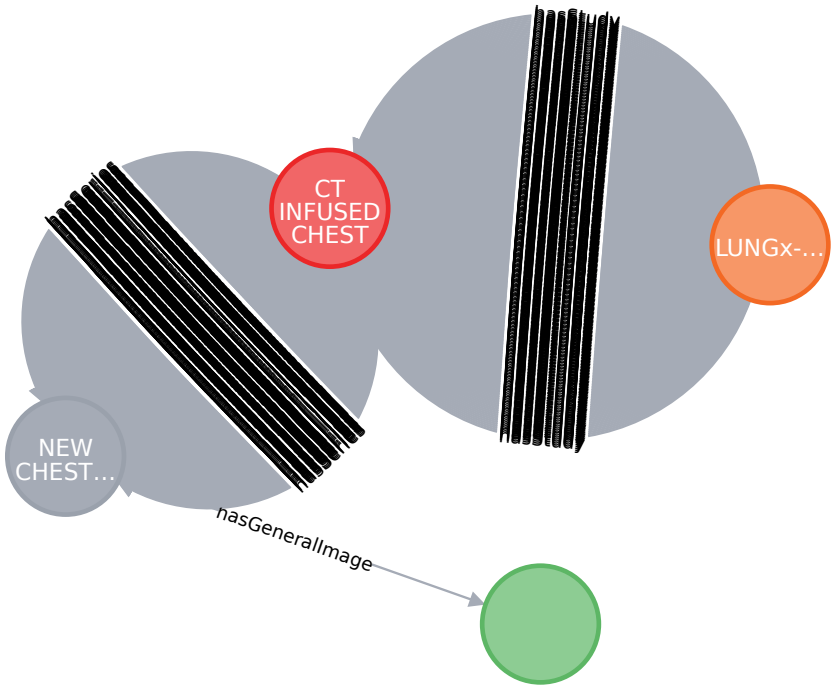

Abbildung 3.5: Kantenwiederholungen in der ersten Version des *Importers*: Vom Patienten *(orange)* führen mehrere hundert Relationen zur Studie *(rot)* und von dort wiederum mehrere hundert Relationen zur untergeordneten Serie *(grau)*, welche dann ebenfalls mehrere hundert Bilder *(grün)* enthält, von denen jedoch durch die verwendete Query nur eines zurückgegeben wurde.

nach *provenances* hinzugefügt werden, wenn eine weitere Kante mit dieser Kanten-ID erstellt werden soll. Gibt es also eine Kante, deren Kanten-ID bereits gespeichert ist, so wird keine neue Kante der gleichen Kanten-ID mit der aktuellen *provenance* erstellt, sondern lediglich die aktuelle *provenance* dem *Set* unter dem *key* der Kanten-ID hinzugefügt. Falls die Kanten-ID noch nicht im *Dictionary* als *key* enthalten ist, wird die Kante als Objekt erstellt und im *Dictionary* ein neuer *key* mit der verwendeten Kanten-ID angelegt und im *value-Set* die *provenance* als erster Wert gespeichert.
Am Ende des *Importers* wird das *Dictionary* in eine *CSV*-Datei geschrieben. Dabei wird mithilfe einer *for*-Schleife über alle *keys* iteriert und der *key* mit dem zugehörigen *Set* ausgegeben. Dadurch wird die immens hohe Kantendichte, die durch den ersten *Importer* die Graph-Navigation in *Neo4j* praktisch unmöglich macht, stark reduziert, ohne dass Informationen eingebüßt werden.

3.1.6 Proof of Concept

Anhand einer zweiten Datenquelle soll hier die tatsächliche generische Verwendbarkeit des *Importers* gezeigt werden. Zunächst werden dafür Daten aus einer anderen Quelle benötigt. Als Quelle wird das *SIMBA Image Management and Analysis System* (siehe [62]) verwendet. Aus den dort aufgeführten Projekten wurde die *ELCAP Database* und daraus

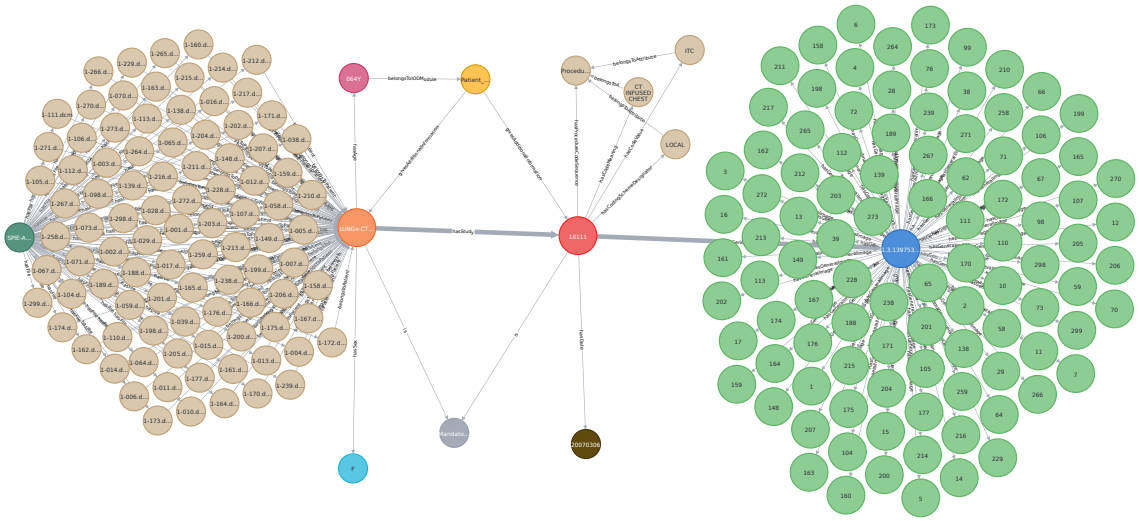

Abbildung 3.6: Resultat der zweiten Version des *Importers*: Vom Patienten *(orange)* führt eine Relation zur Studie *(rot)* und von dort wiederum eine Relation zur untergeordneten Serie *(blau)*. Um den Serien-Knoten angeordnet befinden sich die Bilder *(grün)* der Serie. Zusätzlich werden zum Patienten noch die *File*-Knoten *(beige)* und die Quelle *(dunkelgrün)* dargestellt sowie einige weitere Knoten, wie z.B. das Geschlecht *(hellblau)*.

.

wiederum das *Zero-Change Dataset* ausgewählt. Die Daten entstammen, wie der Webseite entnommen werden kann, der *Public Lung Database to Assess Drug Response*. Es wird für sie eine zweite Konfigurationsdatei mit dem Namen *dev2.ini* erstellt, die zum Teil andere Knoten als die erste enthält. Da es nur darum geht, konzeptionell zu zeigen, dass das Skript auch für andere Datensätze und Konfigurationen funktioniert, wird in der Konfigurationsdatei nur eine wesentlich kleinere Gesamtanzahl an Knotentypen verwendet. Dann wird das Skript für diese Daten aufgerufen.

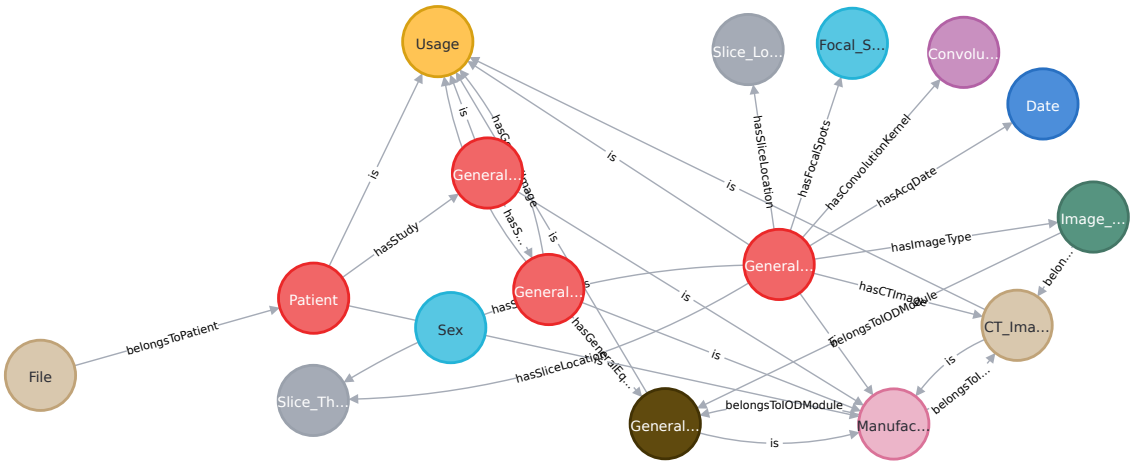

Abbildung 3.7: Das Schema der Datenbank der Zweitdaten für den Proof of Concept.

Die ausgegebenen *CSV*-Dateien werden wie zuvor schon in den Ordner *import* der Datenbank verschoben und anschließend der Import über den Befehl in der Kommandozeile

ausgeführt. Der zugehörige Import-Befehl ist unter dem gleichen Link wie oben im *Git Repository* zu finden. Nach dem Import der Daten und dem Start der zugehörigen Datenbank wird im Remote Interface mithilfe des Befehls `CALL db.schema.visualization` das Schema der Daten abgerufen. Das Resultat ist in Abbildung 3.7 zu sehen. Einige Knoten überschneiden sich mit denen der ursprünglichen Datenbank, allerdings wurden auch neue Knoten wie beispielsweise `Convolution Kernel` und `Focal Spots` hinzugefügt und andere Knoten ausgelassen.

3.2 *Link Prediction* mit *Neo4j* und *CRFs*

In diesem Kapitel sollen mithilfe der *Link Prediction* für den zuvor konstruierten Graphen mögliche, zunächst nicht vorhandene, Kanten vorhergesagt werden. Dafür wird der Graph aus *Neo4j* über die *py2neo*-Bibliothek (siehe [63]) in Python importiert. Dann werden mithilfe einer Query Pfade aus dem Graphen ausgelesen, die als Eingabe für die *Conditional Random Fields* und die *Link Prediction* verwendet werden sollen. Mit Orientierung am Beispiel aus [44] und [47] werden die Pfade in *NER*-kombatible Form gebracht (siehe Unterunterabschnitt 2.5.4) . Es wird also zunächst der Pfad p betrachtet. Dann werden für jeden in p enthaltenen Knoten v alle Nachbarknoten $u \in \Gamma(v)$ als mögliche Labels herangezogen. Dadurch kann jeder Knoten sowohl als Knoten eines Pfades als auch als Label für andere Knoten verwendet werden.

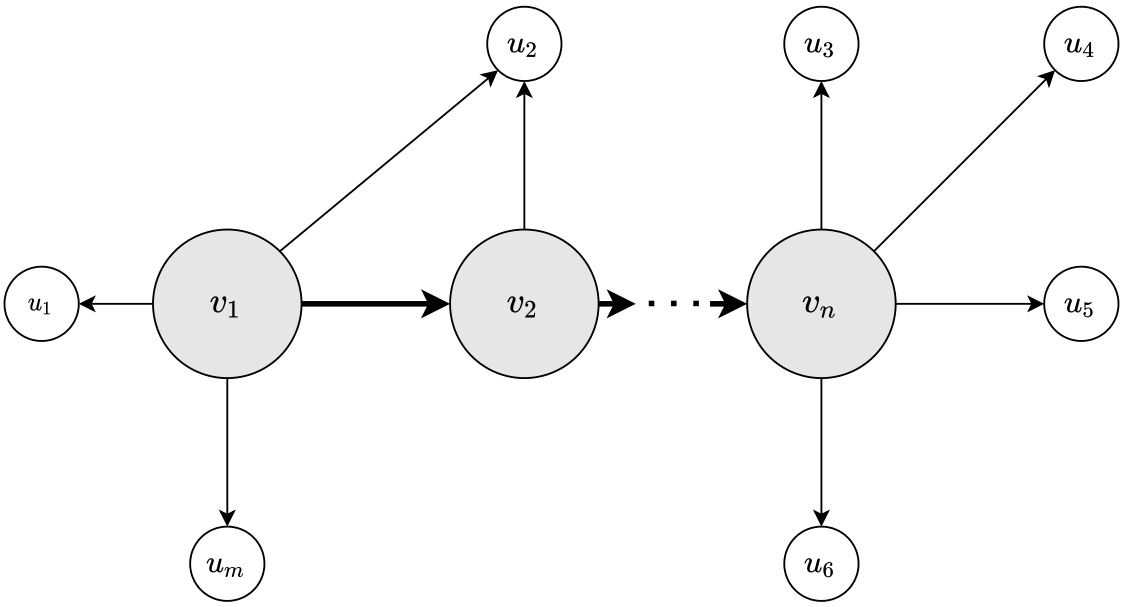

Abbildung 3.8: Der Eingabepfad p besteht aus den grau unterlegten Knoten $v_i,\ i = 1,...,n$. Die weißen Knoten $u_j,\ j = 1,...,m$ dienen als Label der Knoten von p.

.

3.2.1 Ein-Knoten-Pfade

Der verwendete Graph bietet mit fast 90.000 Knoten und nahezu 700.000 Kanten zu viele Möglichkeiten, Pfade zu wählen, als dass alle betrachtet werden können. Aus diesem Grund muss eine Entscheidung für wenige Beispiele und deren Untersuchung erfolgen. Die einfachste Form bietet ein Pfad der Länge eins, also ein einzelner Knoten und seine direkte Nachbarschaft. Dafür werden aus dem Graphen diese Ein-Knoten-Pfade ausgelesen. Dabei wird spezifiziert, welcher Knotentyp betrachtet wird, z. B. Patienten oder Bilder. Zunächst werden alle Patientenknoten behandelt. Dann wird über eine Graph-Query die direkte Nachbarschaft $\Gamma(v)$ dieser Knoten $v \in G$ gefunden und als Gruppe von Labels *l(v)* für *v* gespeichert. Da die *CRF*-Bibliothek jedem Knoten *v* immer nur ein Label zuweisen kann, müssen Kriterien zur Auswahl herangezogen werden. Dafür wird hier Unterabschnitt 2.5.3 herangezogen, um Knoten mit beispielsweise dem höchsten *Score* in einem der in *Neo4j* vorhandenen Link-Prediction-Algorithmen auszuwählen. Später wird als Alternative auch eine alphabetische Sortierung gegeben. Die Wahl der Methode zur Sondierung der Label beeinflusst dabei zum einen das Ergebnis und zum anderen auch die Laufzeit der Querys. Als erste Instanz werden einzelne Patientenknoten betrachtet, als deren Label der Nachbar mit dem jeweils höchsten *Score* bei *Common Neighbours* gewählt wird. Die dafür verwendete Query ist die folgende:

```
MATCH (p:Patient)-[]-(a) RETURN p.nodeUID as patientNode, a.nodeUID
as labelNode, gds.alpha.linkprediction.commonNeighbors(p,a) AS score
ORDER BY p.nodeUID,score DESC,a.nodeUID
```

Der erste Ausschnitt der Ausgabe kann in Tabelle 3.2 betrachtet werden. Mithilfe dieser Query werden in der Methode `create_sents()` dann die Sätze oder Pfade aus Knoten mit den knotenzugehörigen Labeln ausgelesen und passend gespeichert. Dies ist in Listing 3.6 dargestellt.

patientNode	labelNode	score
CT-Training-BE001	SPIE-AAPM Lung CT Challenge	251.0
CT-Training-BE001	1.2.840.113704.1.111.2112.1167842143.1	2.0
CT-Training-BE001	Patient_Study	2.0
CT-Training-BE001	073Y	1.0
CT-Training-BE001	1-001.dcm	1.0
CT-Training-BE001	1-002.dcm	1.0
CT-Training-BE001	1-003.dcm	1.0
...	...	...

Tabelle 3.2: Auszug der Ausgabe der Query *Q1*. Die Bezeichnung *score* bezieht sich auf den Wert, der für den Knoten und sein Label durch den *Common Neighbors*-Algorithmus von *Neo4j* berechnet wurde.

Anschließend werden die Daten in Test- und Trainingsmenge unterteilt. Dafür wird die Funktion `train_test_split()` aus der *sklearn*-Bibliothek herangezogen.

```
def create_sents():
    # get paths from query (sorted)
    x = 'patientNode'
    q_path = graph.run(f"MATCH (p:Patient)-[]-(a) RETURN p.nodeUID as patientNode, a.↩
        nodeUID as labelNode, gds.alpha.linkprediction.commonNeighbors(p,a) AS score ↩
        ORDER BY p.nodeUID,score DESC,a.nodeUID").data()

    # create sents from q_path

    # get stat nodes
    # go through each element of q_path, store it as a key in a dictionary and the list↩
         of labels as its value
    for element in q_path:

        start_node = element[x]

        if start_node not in sentsDic:
            sentsDic[start_node] = str(element['labelNode'])

    # for each (start)node in q_path store it together with the list of labels as a ↩
        small list. Append the whole
    # sentence as a list to sents (here sentence = 1 word = 1 node)
    for key in sentsDic:
        sents.append([[key, sentsDic[key]]])
```

Listing 3.6: Die `create_sents`-Methode erstellt die Sätze.

Dann müssen die Features festgelegt und dafür die Knotenattribute abgefragt werden. Die Attribute, die ein Knoten nicht enthält, werden auf *null* gesetzt. Dafür werden alle möglichen Attribute der Konfigurationsdatei entnommen und um die hiervon unabhängig im Programm vorhandenen erweitert. Im Folgenden beginnt dann das Training. Die Daten liegen in analoger Form zu [44] vor und werden dann an die Methoden `sent2features()` und `sent2labels()` übergeben. Wie auch zuvor wird bei den einzelnen Schritten die Zeit gemessen. Der zugehörige Programmausschnitt ist in Listing 3.7 zu sehen. Die Ergebnisse werden später in Abschnitt 4.2 präsentiert.

```
# returns the sentence with features found for each node in it
  def sent2features(sent):
      return [nodes_to_features(sent, i) for i in range(len(sent))]
  # returns the labels
  def sent2labels(sent):
      return [set_of_labels for node, set_of_labels in sent]

  # training part 1: setting up the data
  print(f"Sende die Knoten jedes sents aus X_train an die nodes_to_features-Methode, ↩
      die dann die Features für jeden Knoten des sents ausliest")
  tic4 = time.time()
  X_train = [sent2features(s) for s in train_sents]
  toc4 = time.time()
```

```
t4 = toc4-tic4
print(f"Die Methode sent_to_features hat für X_train {t4} Sekunden gebraucht.")
print("-------------------------------------------------------------------------")

print(f"Für jeden sent in y_train wird das Label für jeden enthaltenen Knoten ↩
    gefunden.")
tic5 = time.time()
y_train = [sent2labels(s) for s in train_sents]
toc5 = time.time()
t5 = toc5-tic5
print(f"Die Methode sent_to_labels hat {t5} Sekunden gedauert.")
print("-------------------------------------------------------------------------")

print(f"Sende die Knoten jedes sents aus X_test an die nodes_to_features-Methode, die↩
     dann die Features für jeden Knoten des sents ausliest")
tic6= time.time()
X_test = [sent2features(s) for s in test_sents]
toc6 = time.time()
t6 = toc6-tic6
print(f"Die Methode sent_to_features hat für X_test {t4} Sekunden gebraucht.")
print("-------------------------------------------------------------------------")

print(f"Für jeden sent in y_test wird das Label für jeden enthaltenen Knoten gefunden↩
    .")
tic7 = time.time()
y_test = [sent2labels(s) for s in test_sents]
toc7 = time.time()
t7 = toc7-tic7
print(f"Die Methode sent_to_labels hat {t5} Sekunden gedauert.")
```

Listing 3.7: `sent2features()` und `sent2labels()` werden auf die Trainings- und Testmenge angewendet. Dabei entstehen die X- und y-Vektoren.

Dann wird der *CRF-Learner* der *sklearn*-Bibliothek zum Einsatz gebracht. Für diesen stehen verschiedene Algorithmen bereit. In dieser Arbeit wird der standardmäßige *Gradient Descent lbfgs-Algorithmus* verwendet. Dieser wurde bereits in Unterabschnitt 2.5.1 beschrieben. Schließlich werden noch alle möglichen Übergänge erlaubt, so dass der Algorithmus selbstständig Zuordnungen schaffen kann, die er vorher noch nicht kannte. Der Algorithmus wird dann auf die Trainingsdaten `X_train` und `y_train` angewandt (siehe Listing 3.8).

```
# training part 2: actual crf
print("Das eigentliche Training mithilfe der CRFs wird durchgeführt.")
print("Training...")
tic8 = time.time()
crf = sklearn_crfsuite.CRF(algorithm='lbfgs', c1=0.1, c2=0.1, max_iterations=100, ↩
    all_possible_transitions=True)
crf.fit(X_train, y_train)
toc8 = time.time()
t8 = toc8-tic8
print(f"Das Training ist beendet. Es hat {t8} Sekunden gebraucht.")
```

Listing 3.8: Anwendung des *CRF-Learners* auf die Trainingsdaten.

Anschließend wird das Ergebnis anhand der Test-Daten ausgewertet. Dafür wird der *F1-Score* herangezogen, welcher in Unterunterabschnitt 2.5.4 erläutert wurde. Der entsprechende Ausschnitt aus dem Code ist in Listing 3.9 dargestellt. Bei jedem Schritt werden die Zeiten gemessen und in Abschnitt 4.2 ausgewertet.

```
print("Der F1-Score wird für alle Label berechnet")
tic10 = time.time()
m1 = metrics.flat_f1_score(y_test, y_pred, average='weighted', labels=labels)
print(m1)
toc10 = time.time()
t10 = toc10-tic10
print(f"Die Berechnung des F1-Scores hat {t10} Sekunden gedauert.")
```

Listing 3.9: Auswertung durch den *F1*-Score.

Abschließend sollen die Resultate genauer betrachtet werden. Dafür wird analog zum Verfahren in der Dokumentation über den Code in Listing 3.10 eine Tabelle erstellt, welche die Label genauer bezüglich Präzision und *Recall* betrachtet.

```
print("Betrachte einzelne Labels im Detail:")
tic11 =  time.time()
sorted_labels = sorted(labels, key = lambda name: (name[1:], name[0]))
print(metrics.flat_classification_report(y_test, y_pred, labels=sorted_labels, digits=4↩
    , zero_division=1))
toc11 = time.time()
t11 = toc11-toc11
print(f"Die Betrachtung der Labels im Detail hat {t11} Sekunden gedauert.")
```

Listing 3.10: Detailliertere Betrachtung der möglichen Labels.

Für die hier betrachteten Ein-Knoten-Pfade werden mehrere Querys verwendet und jeweils für `q_path` genutzt. Diese sind in Tabelle 3.3 dargestellt und werden im Weiteren mit *Q1* - *Q5* bezeichnet. *Q1* betrachtet Patientenknoten p und sortiert bei der Rückgabe der Nachbarn eines gegebenen Patientenknotens $\Gamma(p)$ nach dem Ergebnis des *Common Neigbors*-Algorithmus für p und $a \in \Gamma(p)$. *Q2* verwendet ebenfalls Patientenknoten p, sortiert die Nachbarn $a \in \Gamma(p)$ allerdings nach der Anzahl von deren Nachbarn, also nach $|\{u \in V : u \in \Gamma(a)\}|$. *Q3* lies sich zwar innerhalb von *Neo4j* abrufen, führte jedoch bei der Verwendung von *py2neo* immer zu einem Fehler in der Graph-Abfrage. *Q4* betrachtet den Knoten `General Image` und dessen Nachbarn. Es wird alphabetisch sortiert. Die letzte Ein-Knoten-Pfad Query *Q5* sucht alle Nachbarn des `Date`-Knotens.

3.2.2 Mehr-Knoten-Pfade

Nachdem Ein-Knoten-Pfade betrachtet wurden, soll ebenfalls für Pfade mit mehr Knoten ein Modell entwickelt werden. Dafür wird das Python-Skript verändert, um die längere Eingabe verarbeiten zu können. Hier wird ebenfalls nach dem Beispiel in [44] vorgegan-

#	Query
1	`MATCH (p:Patient)-[]-(a) RETURN p.nodeUID as patientNode, a.nodeUID as labelNode, gds.alpha.linkprediction.commonNeighbors(p,a) AS score ORDER BY p.nodeUID,score DESC,a.nodeUID`
2	`MATCH (p:Patient)-[]-(a) RETURN p.nodeUID as patientNode, a.nodeUID as labelNode, gds.alpha.linkprediction.totalNeighbors(p,a) AS score ORDER BY p.nodeUID,score,a.nodeUID`
3	`MATCH (p:General_Image)-[]-(a) RETURN p.nodeUID as imageNode, a.nodeUID as labelNode, gds.alpha.linkprediction.totalNeighbors(p,a) AS score ORDER BY p.nodeUID,score,a.nodeUID`
4	`MATCH (p:General_Image)-[]-(a) RETURN p.nodeUID as imageNode, a.nodeUID as labelNode ORDER BY p.nodeUID,a.nodeUID`
5	`MATCH (p:Date)-[]-(a) RETURN p.nodeUID as dateNode, a.nodeUID as labelNode ORDER BY p.nodeUID,a.nodeUID`

Tabelle 3.3: Querys für Ein-Knoten-Pfade.

gen. Die untersuchten Querys sind in Tabelle 3.4 abgebildet. *Q6* - *Q9* unterscheiden sich nur durch die Beschränkung der Menge der Ergebnisse. Dies ist aufgrund der großen Datenmenge und der hohen Laufzeit so gewählt worden. Später wird in einem weiteren, verbesserten Ansatz auch die gesamte Datenmenge getestet. Die Methode `create_sents()` wird daher angepasst. Sie ist in Listing 3.11 dargestellt.
Auch hier wird zunächst der Graph mithilfe einer Query abgefragt. Dann werden die Liste `path_Nodes = []` und die Variable `path_length = 2` angelegt. Letztere gibt die Länge der in der Query untersuchten Pfade erneut an, damit in ersterer die Knoten `pathNode1`, `pathNode2` usw. abgelegt werden können. Über die folgende Schleife werden damit die *sents* erstellt, indem den `path_Nodes`-Elementen der Wert aus dem Query-Resultat zugeordnet werden kann. Die innere *for*-Schleife war bisher nicht notwendig, da die Pfade nur aus jeweils einem Knoten bestanden. Über die Variablen `path_Nodes` und `path_length` kann dadurch allerdings auch eine längere Query als Eingabe verarbeitet werden. Zusätzlich werden auch die Features erweitert. Da die Pfade mehrere Knoten enthalten, wird in den Variablen `nxt` und `prev` der Nachfolger und der Vorgänger des jeweiligen Knotens gespeichert und anschließend an das Feature-*Dictionary* weitergegeben. Sollte der betrachtete Knoten selbst der Start- oder Endknoten des Pfades sein, wird dies in `nxt` bzw. `prev` gespeichert.
Wie sich durch die in Unterabschnitt 4.2.1 beschriebenen Laufzeitmessungen zeigen wird, arbeitet diese Variation des Programms zur *Link Prediction* zwar korrekt, benötigt aber zu viel Zeit, um praktikabel zu sein. Dies liegt an der Methode `sent2features()`, die dafür

#	Query
6	`MATCH (n1)-[]-(p1:Patient)-[]-(p2:General_Study)-[]-(n2) RETURN p1.nodeUID as pathNode1, p2.nodeUID as pathNode2, n1 as NeighbourOfp1, n2 as NeighbourOfp2 ORDER BY p1.nodeUID, p2.nodeUID, n1.nodeUID, n2.nodeUID LIMIT 1000`
7	`MATCH (n1)-[]-(p1:Patient)-[]-(p2:General_Study)-[]-(n2) RETURN p1.nodeUID as pathNode1, p2.nodeUID as pathNode2, n1 as NeighbourOfp1, n2 as NeighbourOfp2 ORDER BY p1.nodeUID, p2.nodeUID, n1.nodeUID, n2.nodeUID LIMIT 10000`
8	`MATCH (n1)-[]-(p1:Patient)-[]-(p2:General_Study)-[]-(n2) RETURN p1.nodeUID as pathNode1, p2.nodeUID as pathNode2, n1 as NeighbourOfp1, n2 as NeighbourOfp2 ORDER BY p1.nodeUID, p2.nodeUID, n1.nodeUID, n2.nodeUID LIMIT 50000`
9	`MATCH (n1)-[]-(p1:Patient)-[]-(p2:General_Study)-[]-(n2) RETURN p1.nodeUID as pathNode1, p2.nodeUID as pathNode2, n1 as NeighbourOfp1, n2 as NeighbourOfp2 ORDER BY p1.nodeUID, p2.nodeUID, n1.nodeUID, n2.nodeUID`

Tabelle 3.4: Querys für Mehr-Knoten-Pfade.

verantwortlich ist, dass die einzelnen *sents* in die passende Form gebracht werden. Das bedeutet zum einen, dass für jeden Knoten eines jeden *sents* die Features aus dem Graphen ausgelesen werden, und zum anderen, dass die Knoten und die abgefragten Features passend gespeichert werden. Anschließend können die *Conditional Random Fields* zum Trainieren mithilfe der *sklearn-crfsuite* angewendet werden.

```
def create_sents():

    # get paths from query (sorted)
    x = 'pathNode1'
    y = 'pathNode2'

    #Q6
    q_path = graph.run(f"MATCH (n1)-[]-(p1:Patient)-[]-(p2:General_Study)-[]-(n2) ↩
        RETURN p1.nodeUID as pathNode1, p2.nodeUID as pathNode2, n1 as NeighbourOfp1, ↩
        n2 as NeighbourOfp2 ORDER BY p1.nodeUID, p2.nodeUID, n1.nodeUID, n2.nodeUID ↩
        LIMIT 50000").data()

    # create sents from q_path

    # regulate length of paths from query for later creation of sents
    path_Nodes = []
    path_length = 2
    for i in range(path_length):
            path_Nodes.append('pathNode'+str(i+1))

    # go through each element of q_path (i.e. through each dictionary, which is ↩
        equivalent to a full path, in the list)
    for element in q_path:
```

```
        sentsDic = {}

        # stores sentence from nodes
        s = []

        # path_Nodes has elements like pathNode1, pathNode2, etc.
        # so it loops through every available path node
        for node in path_Nodes:
            if node not in sentsDic:
                sentsDic[element[node]] = element['NeighbourOfp' + node[-1]]['nodeUID']

        # for each node/word the node and its label are appended as a tuple to the ↩
            sentence
        for key in sentsDic:
            s.append((key, sentsDic[key]))

        # avoid the same sentence multiple times with different labels
        check_string = str(s)
        if check_string not in check_for_doubles:
            sents.append(s)
            check_for_doubles.add(check_string)
```

Listing 3.11: Die `create_sents()` Methode für Mehr-Knoten-Pfade.

Die problematische Methode arbeitet für die an sie übergebene Liste von *sents* für jedes Element die gleichen Schritte ab. Diese können also unabhängig voneinander bearbeitet werden. Daher wird hier auf eine Parallelisierung des Teilprogramms zurückgegriffen. Dieser Teil des Programms wird dafür umgeschrieben und in Listing 3.12 gezeigt.

```
def sent2features(sent, d):
    return [nodes2features(sent, i, d) for i in range(len(sent))]


def nodes2features(s, i, d):

    node = s[i][0]
    try:
            nxt = s[i + 1][0]
    except:
            nxt = 'this_is_the_end_node'
    if i > 0:
            prev = s[i - 1][0]
    else:
            prev = 'this_is_the_start_node'


    ft = d
    graph = Graph("bolt:///localhost:7474/", auth=("neo4j", "0000"))

    for key in ft:
        if key != 'previous' and key != 'next':
            temp1 = '{'
            temp2 = '}'

            ft[key] = str(list(graph.run(f"MATCH (n {temp1}nodeUID:'{node}'{temp2}) ↩
                RETURN n.{key}").data()[0].items())[0][1])
```

```
    ft['next'] = nxt
    ft['previous'] = prev

    return ft


def main(t_sents, ft):

    pool = Pool(mp.cpu_count())
    prt = partial(sent2features, d=ft)
    p = pool.map(prt, t_sents)

    pool.join
    return p
```

Listing 3.12: Die parallelisierte `sent2feature()` Methode für Mehr-Knoten-Pfade.

Hier wird die Python-Bibliothek *multiprocessing* (siehe [64]) verwendet. Es wird die neue Methode `main(t_sents, ft)` erstellt, der die Menge der *sents* als `t_sents` und das Feature-*Dictionary* als `ft` übergeben werden. Ersteres sind die `train_sents` und später dann die `test_sents`. Dann werden mit `pool = Pool(mp.cpu_count())` parallele Prozesse erstellt. Deren Anzahl richtet sich nach den vorhandenen Prozessorkernen und beläuft sich für den hier verwendeten Rechner auf 8 physische und 16 logische Kerne. An diese Prozesse wird eine Funktion und eine sogenannte *iterable*, also eine Variable die iteriert werden kann, übergeben. Da aber zusätzlich noch ein fester Wert, nämlich das Feature-*Dictionary*, übergeben werden soll, wird auf die Methode `partial()` (siehe [65]) zurückgegriffen. Diese gibt ein neues partielles Objekt zurück, das sich beim Aufruf wie die übergebene Methode verhält, allerdings noch zusätzliche Argumente mit übergeben kann. Über die Methode `pool.map(prt, t_sents)` wird dann dieses partielle Objekt an die Prozesse übergeben. Der Befehl `pool.join` am Ende ist dafür verantwortlich, dass alle Prozesse abgeschlossen werden, bevor weiterer Code ausgeführt werden kann. Das vermeidet ungewollte, parallele Ausführung des Folgecodes.

#	Query
3	`MATCH (p:General_Image)-[]-(a) RETURN p.nodeUID as imageNode, a.nodeUID as labelNode, gds.alpha.linkprediction.totalNeighbors(p,a) AS score ORDER BY p.nodeUID, score DESC, a.nodeUID`
10	`MATCH (n1)-[]-(p1:Patient)-[]-(p2:General_Study)-[]-(n2) RETURN p1.nodeUID as pathNode1, p2.nodeUID as pathNode2, n1.nodeUID as NeighbourOfp1, n2.nodeUID as NeighbourOfp2, gds.alpha.linkprediction.commonNeighbors(p1,n1) as scoren1, gds.alpha.linkprediction.commonNeighbors(p2,n2) as scoren2 ORDER BY p1.nodeUID, p2.nodeUID, scoren1 DESC, scoren2 DESC`

Tabelle 3.5: Querys mit Fehlern in *py2neo*.

Die Methoden `sent2features(sent,d)` und `nodes2features(s,i,d)` unterscheiden sich kaum von ihrer bisherigen Version. Es wird lediglich das Feature-*Dictionary* jeweils weitergegeben und zusätzlich das Objekt `graph` erneut erstellt. Letzteres ist durch die Umstrukturierung des Programms bedingt. Diese ist notwendig, da die teilweise bestehende Parallelisierung des Programms an mehreren Stellen einen geänderten Aufbau und eine neue Reihenfolge erforderlich macht. Darauf wird hier jedoch nicht weiter eingegangen, da dieser Aspekt für die Arbeit nicht weiter interessant ist.
Durch die Veränderung des Programms werden die Listen `X_train` und `X_text` jeweils nicht mehr durch einen ausschließlich nacheinander ablaufenden Prozess erstellt, sondern durch parallel geschaltete Prozesse. Der Geschwindigkeitsgewinn wird später in Unterabschnitt 4.2.1 präsentiert. Dort werden auch die Laufzeiten der linearen und der parallelen Version anhand der hier verwendeten Querys *Q6* - *Q9* untersucht.
Schließlich soll noch das Problem betrachtet werden, bei welchem die Query in *py2neo* einen Fehler birgt, im Remote Interface von *Neo4j* hingegen einwandfrei funktioniert. Daher wird die Query dort gestellt und dann als *CSV*-Datei exportiert. Dann kann diese Datei in Python importiert werden und mit einer modifizierten Version der Methode `create_sents()` in die passende Form gebracht werden. Dieses Problem tritt, wie oben schon genannt wurde, bei *Q3* und ebenfalls bei *Q10*, einer modifizierten Version von *Q9*, auf. Beide Querys sind in Tabelle 3.5 dargestellt.

```
def create_sents_from_import():

    q_path_import = []
    row_counter = 0
    with open('exportQ10.csv', newline='') as csvfile:
        reader = csv.reader(csvfile)
        for row in reader:
            row_counter = row_counter + 1
            if row_counter > 1:
                q_path_import.append(row)


    for element in q_path_import:
        sent = []
        node1 = element[0]
        node2 = element[1]
        neighOfp1 = element[2]
        neighOfp2 = element[3]
        sc1 = element[4]
        sc2 = element[5]
        tup1 = (node1, neighOfp1)
        tup2 =  (node2, neighOfp2)
        sent = [tup1, tup2]
        sents.append(sent)
```

Listing 3.13: Die modifizierte `create_sents()` Methode, die über den externen Export-Import arbeitet.

Die veränderte Methode `create_sents_from_import` ist in Listing 3.13 dargestellt. Sie überspringt die erste Zeile, da diese nur die Überschrift enthält. Nach der Methode verläuft das Programm wie zuvor auch schon.
Alle verwendeten Skripte für den Import der Daten sowie die verwendeten Konfigurationsdateien sind vollständig im *Git Repository* unter `https://github.com/TbsHbnthl/master-s-thesis-link-prediction-on-large-scale-knowledge-graphs/tree/main/ImportDICOM_fertig` abgelegt. Dort unter `https://github.com/TbsHbnthl/master-s-thesis-link-prediction-on-large-scale-knowledge-graphs/tree/main/CRF_fertig/scripts_for_ssh_fertig` sind die vollständigen Skripte für die *Link Prediction* mithilfe der *Conditional Random Fields* abgelegt.

Kapitel 4

Evaluation

4.1 Laufzeitanalyse für den *Importer*

In diesem Kapitel soll die theoretische Zeitkomplexität des ersten *Importers* untersucht und mit der tatsächlichen Laufzeit verglichen werden. Der hier beispielhaft verwendete Datensatz verwaltet über 70 verschiedene Patienten, denen wiederum zwischen 202 und 468 Bilder zugeordnet werden. Insgesamt werden $n = 22.512$ Dateien bearbeitet. Zunächst wird die Konfigurationsdatei eingelesen. Deren Größe hängt von der Anzahl der gewünschten verschiedenen Knotentypen (≤ 60), allerdings nicht von der eigentlichen Eingabe der Dateien ab, liegt also in konstanter Zeit $\mathcal{O}(1)$. Im Anschluss werden Listen, *Dictionarys*, *Sets* und Objekte erstellt, die jedoch zunächst auch von der Eingabegröße unabhängig sind und nur von der Anzahl der Knotentypen in der Konfigurationsdatei abhängen. Dies liegt also ebenfalls in $\mathcal{O}(1)$. Danach werden die *static nodes* `Sex` und `Usage`

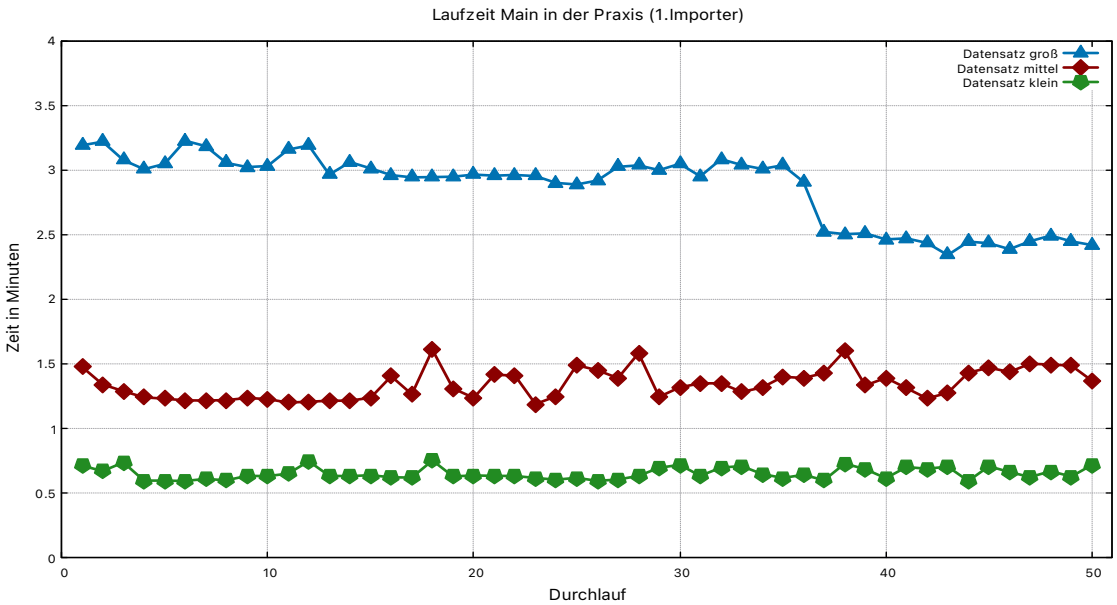

Abbildung 4.1: Laufzeiten (in Minuten) des *main*-Teils des ersten *Importers* über 50 Testinstanzen für kleine, mittlere und große Daten.

ausgegeben. Beide sind ebenfalls unabhängig von der Eingabe und bestehen nur aus zwei bzw. drei verschiedenen Knoten. Daher liegt dies auch in $\mathcal{O}(1)$, gleiches gilt für den folgenden Wechsel in das Verzeichnis *rootdir*.
Anschließend beginnt die erste *for*-Schleife. Diese durchläuft alle k Verzeichnisse der Dateien. Für die hier verwendeten Beispieldaten sind das $k = 70$ Verzeichnisse, d.h. eines je Patient, die nacheinander aufgerufen werden. Die nächste *for*-Schleife betrachtet jede Datei innerhalb des jeweiligen Verzeichnisses, in diesem Fall also im Mittel $\frac{n}{k} = \frac{22512}{70} = 321,6$ Dateien. Wenn man diese beiden *for*-Schleifen ineinander geschachtelt betrachtet, erhält man also die Komplexität $\mathcal{O}(k \cdot \frac{n}{k}) = \mathcal{O}(n)$. Die *if*-Abfrage ist ebenfalls in $\mathcal{O}(1)$, da nur für die aktuelle Datei der Dateiname überprüft wird. Dann wird die *DICOM*-Datei eingelesen und ein Objekt erzeugt, auf das zugegriffen werden kann. Die Erzeugung des Objekts liegt auch in $\mathcal{O}(1)$. Für *provenance* werden zwei *Strings* miteinander über den +-Operator verknüpft. Nach [66] liegt dies zwar in $\mathcal{O}(s^2)$, wenn s der Zeichenanzahl in den *Strings* entspricht, aber die *String*-Länge hängt nicht von der Eingabegröße n, sondern von den Bezeichnungen innerhalb der *DICOM*-Datei ab. Diese sind allerdings auch begrenzt und daher liegt die Verkettung der *Strings* in $\mathcal{O}(1)$. Die nächste *if*-Abfrage

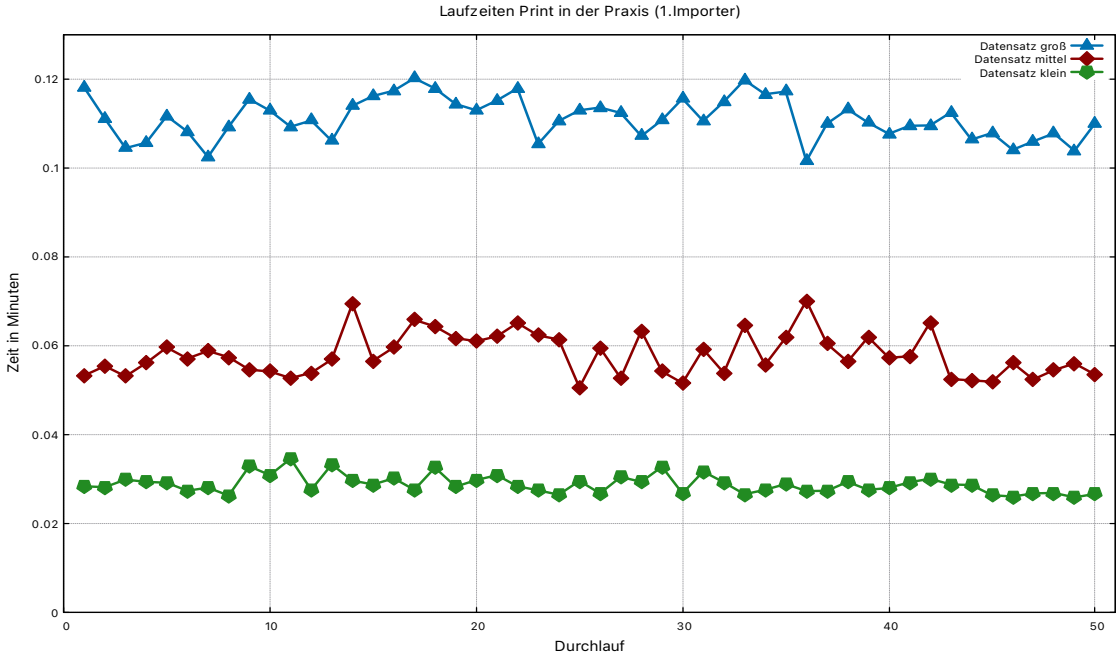

Abbildung 4.2: Laufzeiten (in Minuten) des *print*-Teils des ersten *Importers* über 50 Testinstanzen für kleine, mittlere und große Daten.

greift auf ein *Set* zu. Nach [67] und [68] sind *Sets* als Hashtabellen implementiert und der Average-Case für die *lookup*-Operation ist $\mathcal{O}(1)$, der Worst-Case $\mathcal{O}(l_{File})$, wobei l_{File} die Länge des *Sets* beschreibt. Diese ist aber durch die verwendeten Daten nach oben durch 468 beschränkt. Auch für andere Datensätze wird diese ähnlich beschränkt sein, wodurch auch der Worst-Case $\mathcal{O}(1)$ bezüglich n ist. Innerhalb der *if*-Anweisung und danach innerhalb der *try-except*-Umgebung werden nur $\mathcal{O}(1)$-Anweisungen ausgeführt: Zuweisung,

Speicherung in einer Liste durch Anhängen und Zugriff auf einen *value* über den zugehörigen *key*. [67] [68]

Es folgt eine weitere *for*-Schleife, die über die *Sections* läuft. Auch diese sind, unabhän-

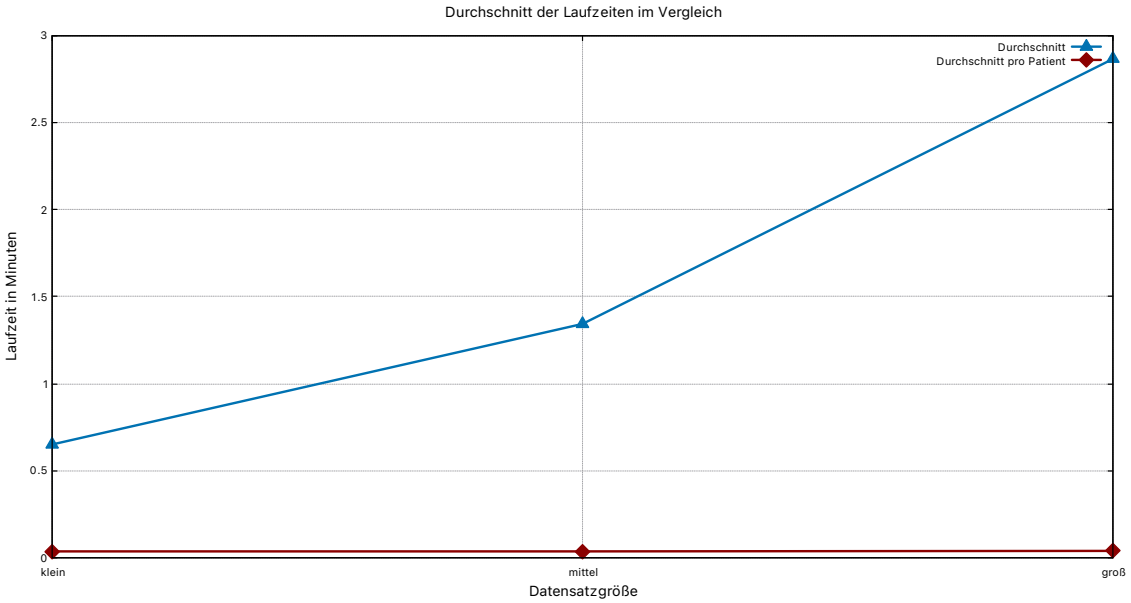

Abbildung 4.3: Durchschnittliche Laufzeit (in Minuten) des *main*-Teils des ersten *Importers* über 50 Testinstanzen für kleine, mittlere und große Daten.

gig von der Eingabedatei, zuvor durch den Benutzer in der Konfigurationsdatei festgelegt worden, also in $\mathcal{O}(1)$. Innerhalb der Schleife gibt es mehrere *if*-Abfragen. Die erste verwendet ein *Set* in einem *Dictionary*. Die *lookup*-Komplexitäten der beiden stimmen überein. Die *keys* des *Dictionarys* stimmen mit den *Sections* überein, sind also unabhängig von der Eingabegröße. Die darin enthaltenen *Sets* umfassen die Patienten-IDs, sind also durch k beschränkt. Im Hinblick auf n liegt diese Schleife insgesamt also auch in $\mathcal{O}(1)$ bezüglich n. Die zweite *if*-Abfrage greift nur auf ein Attribut der eingelesenen Konfigurationsdatei zu, auch in $\mathcal{O}(1)$. Die folgende *if*-Abfrage prüft ebenfalls den Eintrag eines bestimmten *keys* der eingelesenen Konfigurationsdatei. Anschließend werden Werte zugewiesen, auf Attribute der *DICOM*-Datei zugegriffen, ein Objekt erstellt und ein Knoten in einer Liste gespeichert. Alle diese Operationen liegen in $\mathcal{O}(1)$. Sie sind zum Teil an Bedingungen geknüpft und die bei dieser Betrachtung einzig relevante ist die Abfrage der *date_time_id* im *Set dates*. Im schlimmsten Fall wurde jedes Bild jeder Serie jeder Studie jedes Patienten zu einem anderen Zeitpunkt aufgenommen. In diesem Fall hätte das *Set* $\mathcal{O}(n)$ Elemente. Die Average-Case-Komplexität der Abfrage liegt dann immer noch in $\mathcal{O}(1)$, die Worst-Case-Komplexität allerdings in $\mathcal{O}(n)$. Dies hängt stark mit der durch die Hashfunktion verursachten Anzahl an Kollisionen zusammen. Auf Basis der weiter unten dargestellten Ergebnisse der praktischen Untersuchungen der Laufzeit kann hier von sehr wenigen Kollisionen und der Average-Case-Laufzeit ausgegangen werden. Die nächste *if*-Abfrage inklusive der *else*-Optionen weist nur Werte zu und fragt solche ab.

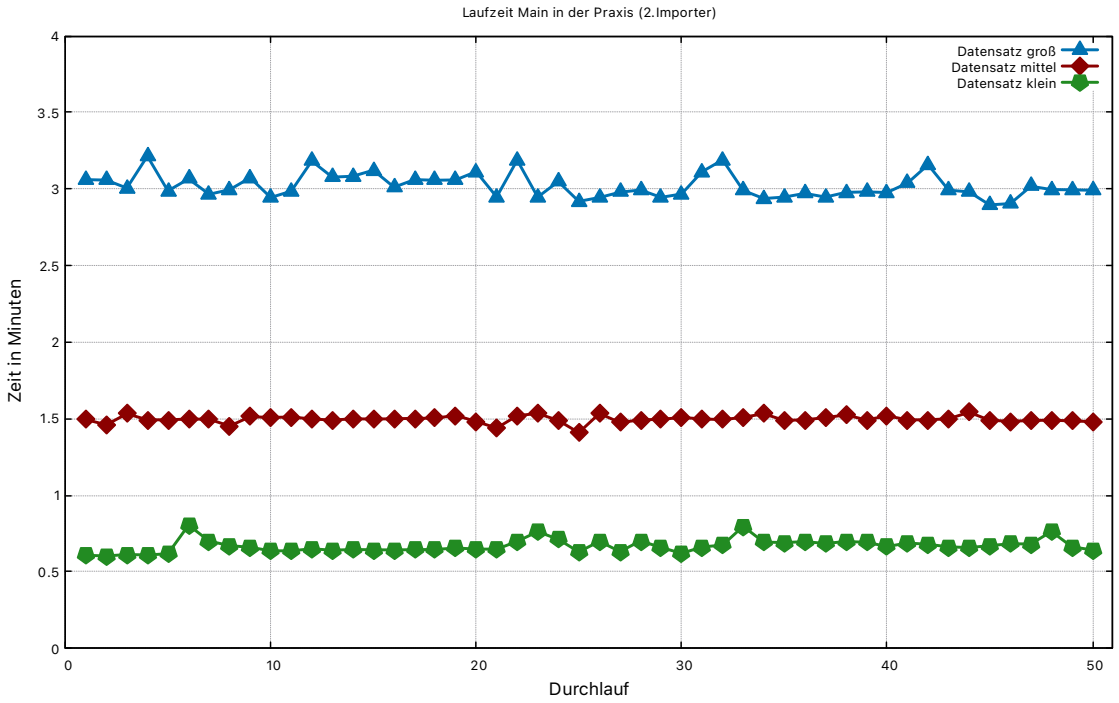

Abbildung 4.4: Laufzeiten (in Minuten) des *main*-Teils des zweiten *Importers* über 50 Testinstanzen für kleine, mittlere und große Daten.

Wie zuvor schon befinden sich diese beiden Prozesse in $\mathcal{O}(1)$. Es folgt die *if*-Abfrage, ob die gefundene *uid* bereits vorhanden ist. Dies wird wie oben über das *Dictionary* geprüft, welches als *keys* die *Sections* und als *values* die *uids* innerhalb der *Sections* enthält. Daher erhält man erneut eine Average-Case-Laufzeit von $\mathcal{O}(1)$, allerdings aufgrund der Elemente innerhalb der *Sets* mit einer möglichen Worst-Case-Laufzeit von $\mathcal{O}(n)$. Auch hier zeigt die praktische Anwendung später, dass die Python-interne Hashfunktion wenige Kollisionen verursacht und die Gesamtlaufzeit nicht beeinträchtigt wird. Die folgende Erstellung des Knotens weist Werte zu und verwendet zwei *if*-Abfragen und eine *for*-Schleife. Die *if*-Abfragen beziehen sich jedoch nur auf Attributsabfragen von vorhandenen Objekten. Die Schleife läuft über die Anzahl der für den Knoten relevanten Attribute. Dies hängt nicht von n ab und ist nach oben durch eine kleine ganze Zahl beschränkt, daher liegt der hier betrachtete Teil, wie auch die folgende Objekterstellung und Speicherung des Knotens, ebenfalls in $\mathcal{O}(1)$. Im darauffolgenden Teil des Codes werden die Relationen erstellt. Dabei wird zunächst auf die Konfigurationsdatei zugegriffen. Anschließend werden anhand des Wertes der Konfigurationsdatei mehrere Fälle untersucht. Je nach Fall wird die Stelle übersprungen oder ein Wert aus der eingelesenen *DICOM*-Datei verwendet. Wie zuvor schon wird hier die *split()*-Funktion von Python verwendet, um den eingelesenen Attributs-*String* aus der Konfigurationsdatei aufzuspalten. Nach dem Sourcecode in [69] läuft dies in $\mathcal{O}(l_{String})$, wobei l_{String} die Länge des *Strings* angibt. Auch das ist unabhängig von n und daher in $\mathcal{O}(1)$. Schließlich werden noch die nicht aus den *DICOM*-Files herauslesbaren Kanten erstellt. Dabei werden Objekte erzeugt und deren Zei-

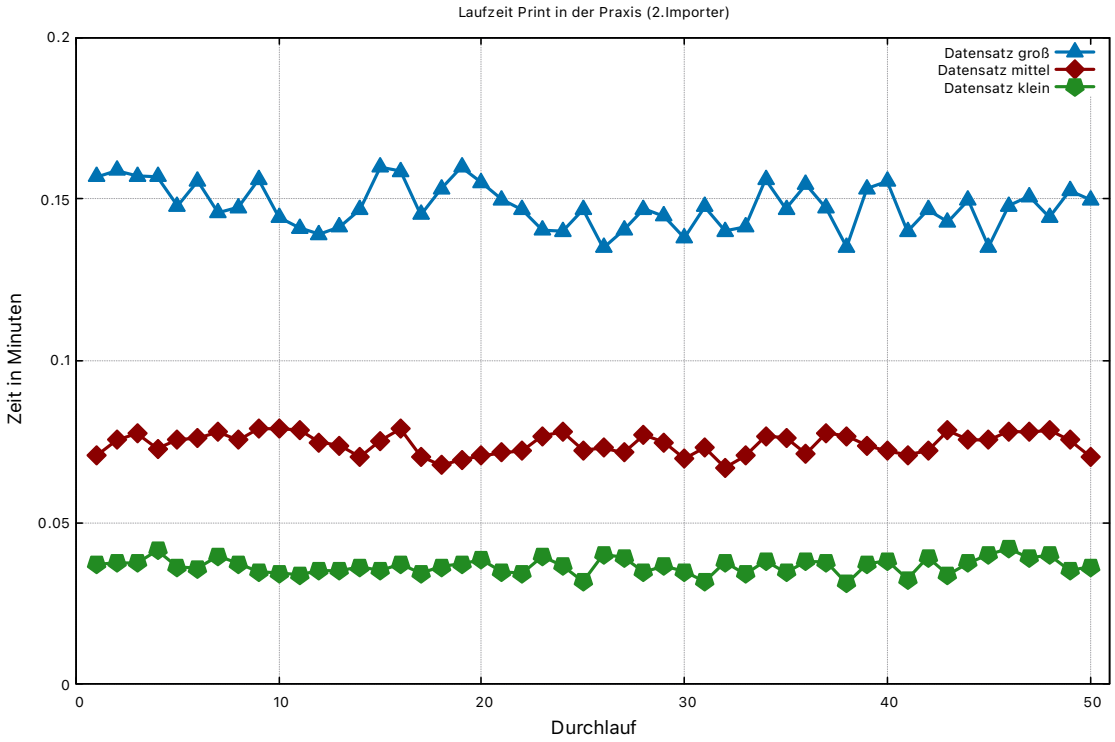

Abbildung 4.5: Laufzeiten (in Minuten) des *print*-Teils des zweiten *Importers* über 50 Testinstanzen für kleine, mittlere und große Daten.

ger in den passenden Listen gespeichert. Beides geschieht in $\mathcal{O}(1)$. Am Ende werden noch die `Source`-Knoten erstellt. Da die Anzahl der Quellen nicht mit der Dateienanzahl zusammenhängt und die Annahme, dass jede einzelne Datei eine eigene Quelle hat, nicht sinnvoll erscheint, liegt auch diese *for*-Schleife in $\mathcal{O}(1)$.

Die finale Ausgabe der *CSV*-Dateien liegt offensichtlich in $\mathcal{O}(n)$, da die Listen mit den hinterlegten Knoten und Kanten vollständig durchlaufen werden müssen. Dies ist jedoch additiv zur bisherigen Zeitkomplexität. Insgesamt ergibt sich nach Theorem 2.3.2 eine Gesamtkomplexität von $\mathcal{O}(n)$ für den Average-Case und $\mathcal{O}(n^2)$ für den Worst-Case. Dies verändert sich auch nicht für die zweite Version des *Importers*. Die dort eingeführte Hilfsvariable *relationID_without_prov* wird verwendet, um Relationen, die bis auf die *provenance* gleich sind, abzufangen und in einem *Dictionary* zu speichern. Die Zuweisung und Speicherung liegen beide in $\mathcal{O}(1)$, genauso wie die Average-Case-*lookup*-Zeit für *Dictionarys*. Am Ende werden die zusätzlichen *provenances* auch als *CSV*-Datei ausgegeben. Diese Operation liegt grundsätzlich in $\mathcal{O}(|E|)$, wobei $|E| \widehat{=} |V|^2$ gilt. Dies könnte die Laufzeit des *print*-Bereichs des *Importers* verlängern. Allerdings ist der Graph nur dünn besetzt, da die knapp 700.000 Kanten für die 80.000 Knoten weit von den möglichen 6,4 Milliarden Kanten entfernt sind. Der Rest des zweiten *Importers* ist identisch mit dem ersten.

Wie sich in der praktischen Laufzeitanalyse zeigt, wird sowohl für den Hauptteil als auch für den *print*-Teil eine lineare Laufzeit erreicht. Dafür wurde der Datensatz in ver-

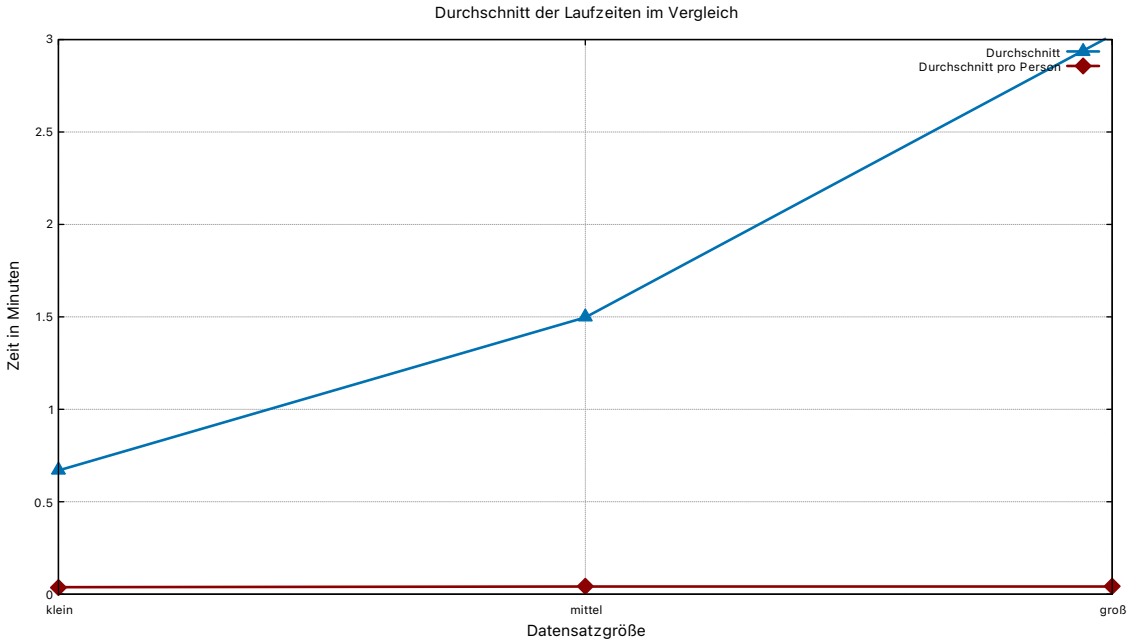

Abbildung 4.6: Durchschnittliche Laufzeit (in Minuten) des *main*-Teils des zweiten *Importers* über 50 Testinstanzen für kleine, mittlere und große Daten.

schiedene Größen unterteilt (volle, halbierte und geviertelte Größe) und die expliziten Laufzeiten wurden gemessen. Alle Laufzeitmessungen wurden auf einem iMac (Retina 5K, 27-inch, 2020) mit einem Intel Core i7-10700K durchgeführt. Die genauen Daten können den Excel-Tabellen im *Git Repository* unter `https://github.com/TbsHbnthl/master-s-thesis-link-prediction-on-large-scale-knowledge-graphs/tree/main/Excel_Data` entnommen werden.

In Abbildung 4.1 wird gezeigt, dass der Hauptteil des *Importers*, also des Teils, der die Daten einliest, konvertiert und speichert, bis auf einige Abweichungen für eine feste Datensatzgröße konstante Laufzeiten erreicht. Gleiches gilt für die in Abbildung 4.2 visualisierten Laufzeiten des *print*-Teils, der für die Ausgabe der konvertierten Daten in den *CSV*-Dateien verantwortlich ist. Diese beiden Ergebnisse spiegeln sich auch in Abbildung 4.3 wider. Dort werden die durchschnittlichen Laufzeiten des Hauptteils der verschieden großen Datensätze miteinander verglichen. Die blaue Kurve beschreibt näherungsweise eine Gerade und zeigt den linearen Anstieg der Laufzeit. Die rote Kurve untermauert dies, da sie die durchschnittliche Laufzeit pro Patient darstellt. Diese ist nahezu konstant über die verschieden großen Datensätze. In Abbildung 4.4, Abbildung 4.5 und Abbildung 4.6 werden analoge Resultate für den zweiten *Importer* gezeigt. Die expliziten Werte sind ebenfalls dem oben genannten *Git Repository* zu entnehmen. Auch hier stimmen die praktischen Ergebnisse mit den theoretischen Überlegungen überein. Die Laufzeit unterscheidet sich im Durchschnitt nur minimal von der ersten Version.
Abbildung 4.7 zeigt die Laufzeiten des *Admin imports* über die Konsole. Diese Tests wurden nur für den vollständigen Datensatz durchgeführt und zeigen bis auf leichte Schwan-

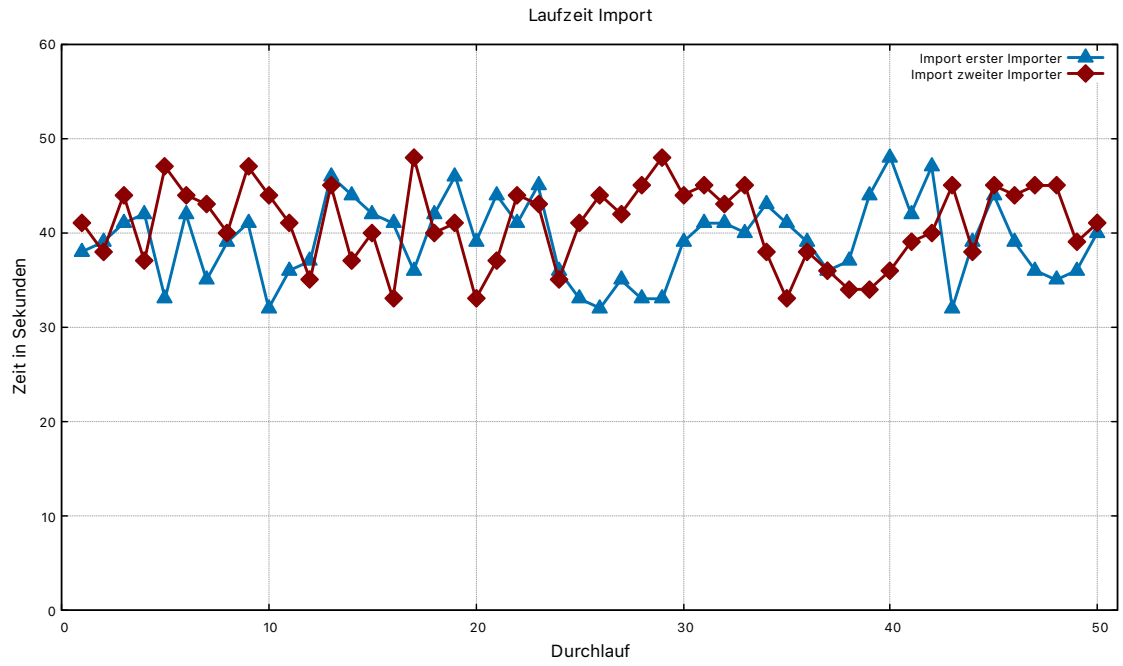

Abbildung 4.7: Laufzeiten des Imports (in Sekunden) der *CSV*-Daten in die Graphdatenbank *Neo4j*.

kungen einen gleichbleibenden Bereich, und zwar sowohl für den ersten als auch für den zweiten *Importer*. Zudem scheint das Resultat der zweiten Version des *Importers* hier keine signifikante Verbesserung zu erbringen. Dies ist bemerkenswert, da die Anzahl der importierten Kanten durch die Reduktion auf jeweils eine *provenance* pro Kanten-ID eine starke Verringerung der zu importierenden Kantenmenge nach sich zieht. Hier zeigt sich die besondere Effizienz des *Admin imports* bei der Verwendung großer Datenmengen.

4.2 Auswertung der *Link Prediction*

4.2.1 Laufzeitanalyse

Dieser Unterabschnitt beschäftigt sich mit der Betrachtung der erzielten Laufzeiten der Programme zur *Link Prediction*. Zunächst wird das Laufzeitresultat der Querys *Q1* - *Q5* präsentiert. Innerhalb des Programms werden für alle einzelnen Abschnitte die Zeiten gemessen.

Da *Q3* innerhalb von Python durch die *py2neo*-Bibliothek einen Fehler verursacht hat, ist hier und im Weiteren mit *Q1* - *Q5* nur *Q1*, *Q2*, *Q4* und *Q5* gemeint. Der problematische Teil des Programms ist die Methode `sent2features()`. Zur Veranschaulichung ist die Laufzeit der zeitlich relevanten Teile in Abbildung 4.8 dargestellt. Alle anderen Teile des Programms haben eine zu vernachlässigende sehr geringe Laufzeit. Vor allem für *Q4* wird dies ersichtlich. Bei dieser Query steht der Knoten `General Image` im Zentrum, welcher schon aufgrund des Aufbaus des Graphen im Vergleich zu anderen Knoten, wie bei-

spielsweise `Patient`, sehr viele Nachbarn hat. Die Laufzeit, welche nahezu vollständig durch `sent2features()` entsteht, beläuft sich hier insgesamt auf etwas unter 11 Stunden. Die gesamten Werte sind im *Git Repository* unter `https://github.com/TbsHbnthl/master-s-thesis-link-prediction-on-large-scale-knowledge-graphs/tree/main/Excel_Data` gegeben.
Die oben genannte Laufzeitproblematik kommt noch stärker zum Tragen, wenn Pfade mit mehr als einem Knoten betrachtet werden. Zu diesem Zweck wurden zunächst *Q6* - *Q9* entworfen, welche sich paarweise nur durch die Einschränkung der Anzahl der Ergebnisse unterscheiden. Dies wurde gemacht, um zunächst die generelle Funktionalität des Programms zu gewährleisten.
In Abbildung 4.9 wird dies besonders deutlich. Wie zuvor schon werden hier nur die für die Laufzeit relevanten Bereiche gezeigt. Alle Daten befinden sich ebenfalls im oben genannten *Git Repository*. Auch hier nimmt die Methode `sent2features()` den größten Teil der Gesamtlaufzeit in Anspruch. Gerade bei der nicht parallelisierten Query *Q8* sticht dies heraus. Diese setzt das Limit auf 50.000 bei den Resultaten und erreicht trotzdem eine mehrtägige Laufzeit. Da die Anzahl der vollständigen Resultate etwa dreimal so hoch ist, limitiert dies die praktische Anwendung massiv. Die Query *Q9* wurde zwar getestet, ist allerdings nach über fünf Tagen nicht fertig geworden und wurde dann aus diesem Grund abgebrochen. Die in Unterabschnitt 3.2.2 eingeführte Parallelisierung zeigt hier ihren großen Vorteil. Statt nur einem Prozessor konnten hier sieben bis acht verwendet werden, was die Laufzeit auf etwa ein Siebtel reduziert hat. Dies zeigt sich in Abbildung 4.9 auch im Vergleich der linearen und der parallelen Programmierung. Durch die hier beschriebene Abänderung des Codes konnte *Q9* nach ungefähr 32 Stunden fertig bearbeitet werden. Das schränkt den praktischen Nutzen immer noch ein, stellt allerdings eine starke Verbesserung dar.

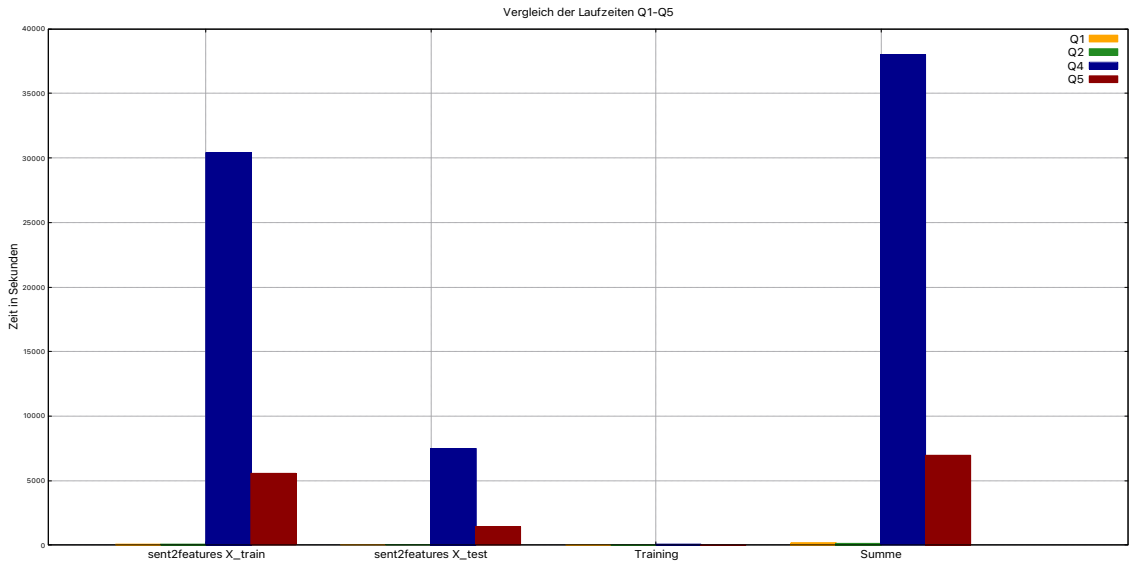

Abbildung 4.8: Durchschnittliche Laufzeit der relevanten Teile der Querys *Q1* - *Q5*.

#	Query
11	`MATCH (n1)-[]-(p1:Patient)-[]-(p2:General_Study)-[]-(n2) RETURN p1.nodeUID as pathNode1, p2.nodeUID as pathNode2, n1.nodeUID as NeighbourOfp1, n2.nodeUID as NeighbourOfp2, gds.alpha.linkprediction.adamicAdar(p1,n1) as scoren1, gds.alpha.linkprediction.adamicAdar(p2,n2) as scoren2 ORDER BY p1.nodeUID, p2.nodeUID, scoren1 DESC, scoren2 DESC`
12	`MATCH (n1)-[]-(p1:General_Study)-[]-(p2:General_Series)-[]-(n2) RETURN p1.nodeUID as pathNode1, p2.nodeUID as pathNode2, n1.nodeUID as NeighbourOfp1, n2.nodeUID as NeighbourOfp2 ORDER BY p1.nodeUID, p2.nodeUID, n1.nodeUID, n2.nodeUID`
13	`MATCH (n1)-[]-(p1:General_Study)-[]-(p2:General_Series)-[]-(n2) RETURN p1.nodeUID as pathNode1, p2.nodeUID as pathNode2, n1.nodeUID as NeighbourOfp1, n2.nodeUID as NeighbourOfp2, gds.alpha.linkprediction.adamicAdar(p1,n1) as scoren1, gds.alpha.linkprediction.adamicAdar(p2,n2) as scoren2 ORDER BY p1.nodeUID, p2.nodeUID, scoren1 DESC, scoren2 DESC`

Tabelle 4.1: Zusätzliche Querys zum Testen.

Zu späteren Testzwecken bezüglich der Güte der Ergebnisse werden noch weitere Querys herangezogen. Diese sind in Tabelle 4.1 dargestellt und beziehen sich ebenfalls auf Mehr-Knoten-Pfade. *Q11* verwendet zur Sortierung der Labels den *Adamic/Adar*-Algorithmus von *Neo4j*. *Q12* und *Q13* bezeichnen andere Pfade im Graphen, diesmal mit den Knoten `General Study` und `General Series`. Bei *Q12* sind die Labels alphabetisch sortiert und bei *Q13* erneut nach dem *Score* aus dem *Adamic/Adar*-Algorithmus. Zur Auswertung der Ergebnisse der *Link Prediction* wird aufgrund der hohen Laufzeiten in weiten Teilen das *High Performance Computing System CHEOPS (HPC)* des Regionalen Rechenzentrums der Universität zu Köln (RRZK) herangezogen (siehe [70]). Das Cluster ist in Knoten mit unterschiedlicher Hardware unterteilt. Für die Ausführung eines Jobs auf dem Cluster werden Ressourcen angefordert. Um mehrere Knoten gleichzeitig nutzen zu können, wäre zunächst die Einbindung dafür vorgesehener Bibliotheken erforderlich. Ein Beispiel dafür bietet *MPI for Python* [71]. Der begrenzte Rahmen dieser Arbeit macht die Beschränkung auf nur jeweils einen Knoten erforderlich. Die einzelnen Knoten bieten dabei maximal einen Prozessor mit 4 Kernen und 32 Threads.
Es fällt auf, dass mit den längeren und anders gewählten Pfaden insbesondere das eigentliche Training in der Laufzeit und den erforderlichen Ressourcen steigt. Dieser Teil war für die Ein-Knoten-Pfade noch verschwindend gering, tritt nun aber stärker zu Tage. Für *Q9 - Q11* werden noch Ergebnisse erzielt, für *Q12* und *Q13* sieht dies anders aus. Der Teil des Programms, in dem das Training mithilfe der *CRFs* stattfindet, benötigt eine enorme Menge an Hauptspeicher. Das verwendete eigene System verfügt über 40 GB und auf dem *HPC* wurde jeweils mit 128 GB gearbeitet. Trotzdem reichte der Speicher nicht aus, um

das Training vollständig zu absolvieren. Die Prozesse wurden durch das jeweilige System vorzeitig beendet.

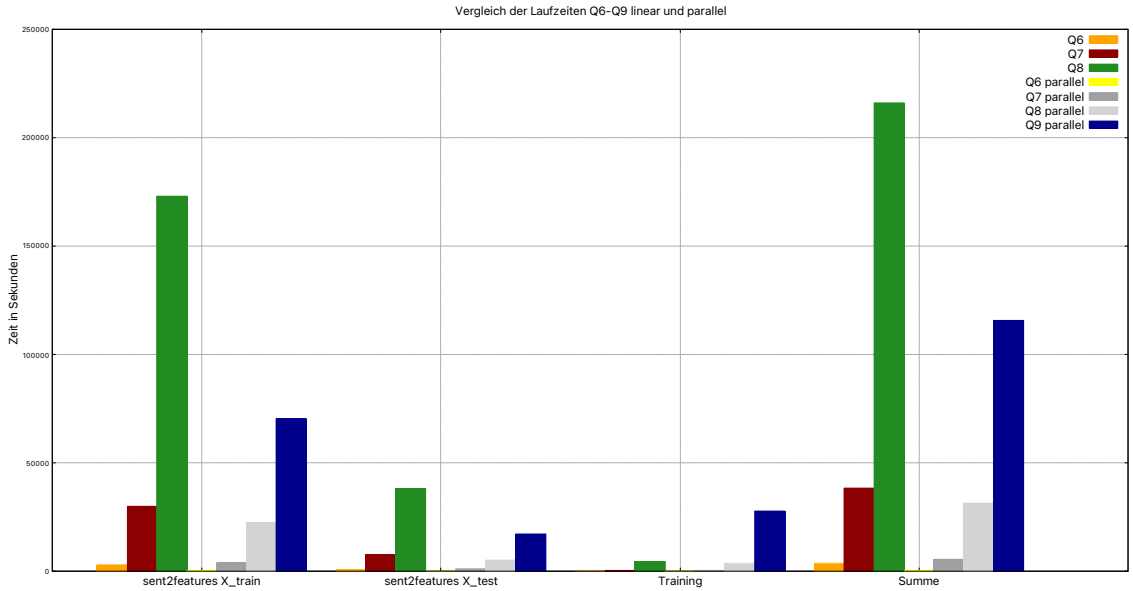

Abbildung 4.9: Durchschnittliche Laufzeit der einzelnen Teile der Querys *Q6* - *Q9*.

Aus diesem Grund wird eine weitere Variation betrachtet. Dabei verwendet das Programm von den Pfaden der Form $P_{l_i^{n_1},l_j^{n_2}} = (l_i^{n_1}, n_1, n_1, l_j^{n_2})$ für jedes Knotentupel (n_1, n_2) nur einen Pfad $P_{l_{i_1}^{n_1},l_{j_1}^{n_2}}$. Die $l_i^{n_k}$ sind dabei die Label des Knotens n_k des Pfades $P_{l_i^{n_1},l_j^{n_2},\ldots,l_k^{n_k},\ldots,l_m^{n_m}}$.
Dies reduziert die Anzahl der zu verarbeitenden Pfade enorm. Statt wie bisher etwa 150.000 Pfade werden dann nur 70 Pfade betrachtet. Für *Q9* und *Q10* betragen die Gesamtlaufzeiten in der parallelen Programmierung auf dem *Intel Core i7-10700K* auf etwa 80 und 40 Sekunden. Ähnliche Zeiten werden für die Querys *Q11* - *Q13* erreicht. Die zugehörigen Messwerte sind im *Git Repository* `https://github.com/TbsHbnthl/master-s-thesis-link-prediction-on-large-scale-knowledge-graphs` in den Ordnern *Excel_Data* und *CRF_fertig* verzeichnet.
Die meisten hier dargestellten Werte sind Durchschnitte aus mehreren gemessenen Durchläufen. Lediglich die sehr großen Durchläufe wurden aus Zeitgründen nicht wiederholt. Die genauen Daten der Laufzeiten können ebenfalls im *Git Repository* betrachtet werden.
Eine weitere Auffälligkeit stellen die verwendeten Algorithmen aus *Neo4j* für die Wahl der Label dar. Die Querys werden sortiert zurückgegeben und das Programm in Python wählt als Label den zuerst auftretenden Nachbarknoten. In manchen hier betrachteten Fällen wurden diese alphabetisch ausgewählt. Es gibt jedoch auch die Möglichkeit, Kriterien zur Güte eines Labels heranzuziehen. Darunter fallen Algorithmen wie *Total Neighbors* und *Common Neighbors*, die in *Neo4j* bereitgestellt werden. Es fällt schnell auf, dass diese Algorithmen die Laufzeit der eigentlichen Query stark beeinträchtigen. Dies ist vor allem bei der Erstellung der *sents* ersichtlich. *Q1* benötigt dafür fast eine Minute, während andere Querys wesentlich schneller verarbeitet werden können. Dies wird aufgrund

der hohen Laufzeiten der Methode `sent2features()` in den Diagrammen hier nicht gezeigt, ist aber in den Tabellen im *Git Repository* erkennbar. Die Verwendung der Methode `create_sents_from_import()` für die sonst mit *py2neo* Fehler produzierenden Querys verändert die Laufzeit des Programms nicht. Lediglich die Query selbst wird in *Neo4j* durchgeführt. Daher wird diese Variante hier nicht weiter untersucht.

4.2.2 Güte der Ergebnisse

Ein-Knoten-Pfade

Die ersten Versuche der *Link Prediction* werden mit Ein-Knoten-Pfaden durchgeführt. Dabei steht zunächst der Patient im Zentrum. Für *Q1* zeigt die *Link Prediction* die Zuordnung der Daten zur zugehörigen Studie *SPIE-AAPM Lung CT Challenge*. Dabei wird ein *F1-Score* von 1 erreicht. Diese Vorhersage ist sehr genau, allerdings ist dies nach der Datenlage nicht überraschend. Der `Patienten`-Knoten hat nur eine sehr begrenzte Art und Anzahl von Nachbarn. Durch die Sortierung nach Anzahl gemeinsamer Nachbarn bleibt nur die Quelle übrig. Dies spiegelt sich auch in der Detailbetrachtung der Labels wieder, wie in Tabelle 4.2 zu sehen ist.

node	precision	recall	f1-score	support
SPIE-AAPM Lung CT Challenge	1.0000	1.0000	1.0000	14
accuracy			1.0000	14
macro avg	1.0000	1.0000	1.0000	14
weighted avg	1.0000	1.0000	1.0000	14

Tabelle 4.2: Detailbetrachtung des Ergebnisses für *Q1*.

In diesen Tabellen sind unter *node* die verfügbaren Labels angezeigt und rechts daneben wird die Präzision, der *Recall* und der *F1-Score* gezeigt (vgl. Unterabschnitt 2.5.4). Der Wert bei *support* gibt die Häufigkeit des Funds an. Gegenteilige Ergebnisse erhält man für die Sortierung nach *Total Neighbors*. Hier wird ein *F1-Score* von 0 erreicht. Die Vorhersage ist hier also vollkommen fehlgeschlagen. Das Resultat kann in Tabelle 4.3 betrachtet werden. Auch hier ist das eigentliche Ergebnis aufgrund der Daten nicht verwunderlich. Die Patienten haben unterschiedliche Alter und durch die kleine Gruppe von Personen ist eine Häufung unwahrscheinlich. Da *Q3* durch einen Fehler in *py2neo* nicht untersucht werden kann, werden hier direkt Grenzen dieser Methode deutlich aufgezeigt und die Weiterarbeit erfolgt mit *Q4*. Die genauen Ergebnisse werden aufgrund des Umfangs nicht hier, sondern im *Git Repository* unter `https://github.com/TbsHbnthl/master-s-thesis-link-prediction-on-large-scale-knowledge-graphs/tree/main/CRF_fertig/Resultate_fertig` dargestellt.

node	precision	recall	f1-score	support
1-414.dcm	1.0000	1.0000	1.0000	0.0
1.2.840.113704.1.111.6436.1168891603.1	1.0000	1.0000	1.0000	0.0
...	...	...	...	...
060Y	1.0000	0.0000	0.0000	2.0
061Y	1.0000	0.0000	0.0000	1.0
062Y	1.0000	1.0000	1.0000	0.0
063Y	1.0000	1.0000	1.0000	0.0
064Y	1.0000	0.0000	0.0000	1.0
...	...	...	...	...
micro avg	0.0000	0.0000	0.0000	8.0
macro avg	0.9730	0.8108	1.7838	8.0
weighted avg	1.0000	0.0000	0.0000	8.0

Tabelle 4.3: Detailbetrachtung des Ergebnisses für *Q2*.

Die vorletzte Ein-Knoten-Pfad-Query ist *Q4*. Hier wurden Label für den Knoten `General Image` untersucht. Die Auswahl des Labels basiert auf alphabetischer Reihenfolge. Aus

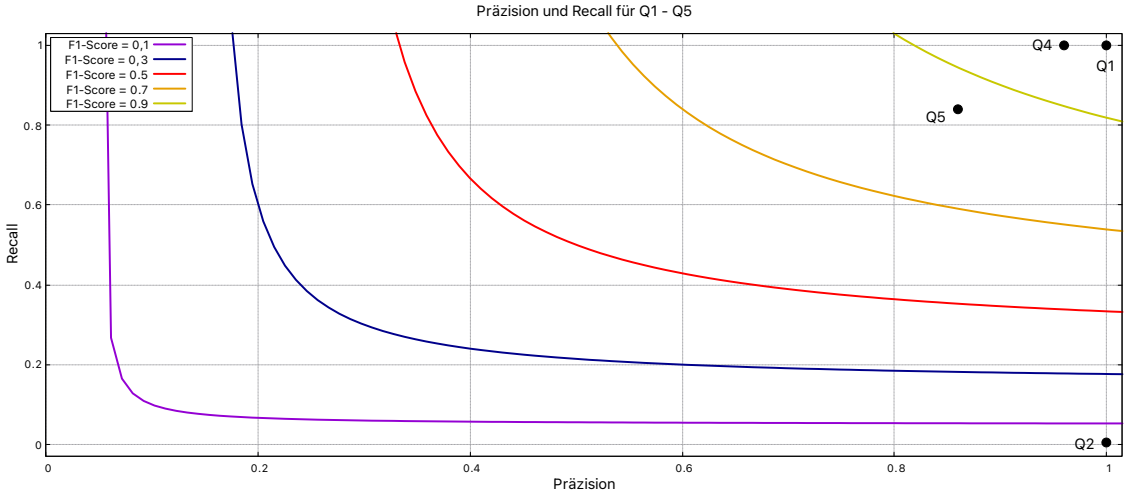

Abbildung 4.10: Präzision-*Recall*-Diagramm für die Querys *Q1* - *Q5*.

biologischer Sicht ist möglicherweise eine andere Gewichtung sinnvoller, es sollten aber aus informatischen Gründen mehrere Methoden zur Wahl des Labels ausprobiert werden. Bei *Q4* wurde mit 0.7737 ein nominell sehr guter Wert für den *F1-Score* erzielt. Die Detailbetrachtung des Ergebnisses wird ausschnitsweise in Tabelle 4.4 präsentiert. Das vollständige Ergebnis ist ebenfalls im *Git Repository* dargestellt. Es zeigt, dass in der Vorhersage verschiedene Label für die Bilder ausgewählt wurden, dabei vorrangig das Label `-1024`. Auch *Q5* steht durch die Knotenauswahl und die gegebenen Daten nur eine begrenzte Menge der Knoten und Kanten des Graphen zur Verfügung. Das Programm liefert mit ihr nominell mit einem *F1-Score* von 0,9819 einen sehr hohen Wert. Das für

node	precision	recall	f1-score	support
-0.10	1.0000	1.0000	1.0000	0
...	...	...	...	...
-100.70	1.0000	1.0000	1.0000	0
-1000	1.0000	0.0000	0.0000	653
-1000.00	1.0000	1.0000	1.0000	0
...	...	...	...	...
-1024	0.8417	1.0000	0.9141	3786
micro avg	0.8417	0.8464	0.8441	4473
macro avg	0.9988	0.7664	0.7658	4473
weighted avg	0.8660	0.8464	0.7737	4473

Tabelle 4.4: Ausschnitt der Detailbetrachtung des Ergebnisses für *Q4*.

den Knoten `Date` vorhergesagte Label ist `SOP_Common`. Auch hier sind die vollständigen Ergebnisse im *Git Repository* zu sehen.
Die Werte der Präzision und des *Recalls* im Vergleich zum *F1-Score* sind für die untersuchten Querys *Q1* - *Q5* in Abbildung 4.10 und Abbildung 4.11 dargestellt. Erstere setzt Präzision und *Recall* zueinander ins Verhältnis. Wie später auch liefern die Höhenlinien in farblicher Anpassung an Abbildung 2.4 einen visuellen Eindruck des zugehörigen *F1-Scores*. Letztere stellt die drei Werte für Präzision, *Recall* und *F1-Score* nebeneinander dar.

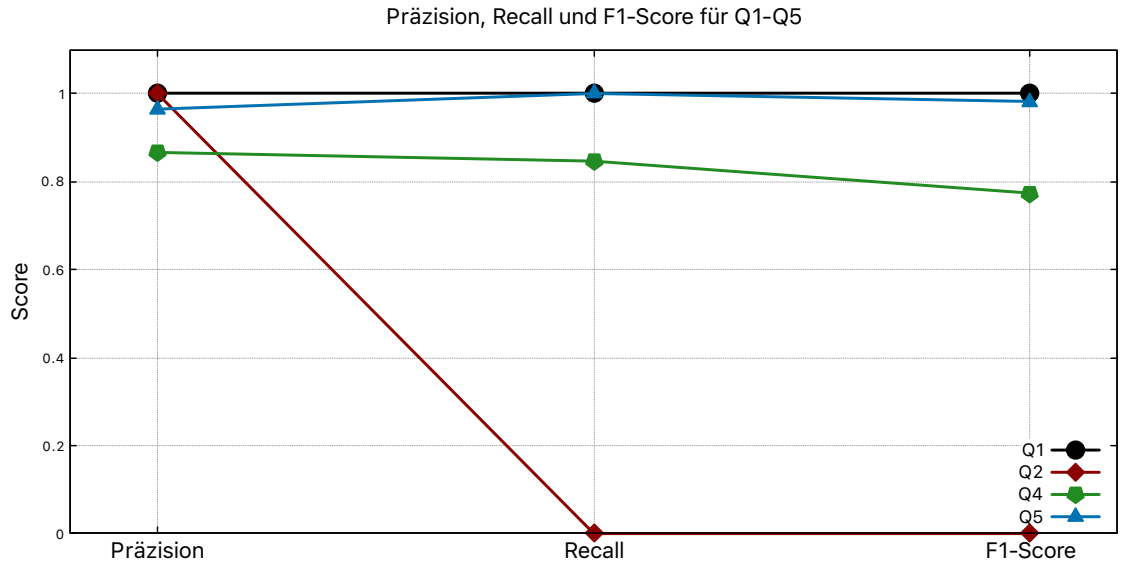

Abbildung 4.11: Vergleich der Präzision, des *Recalls* und des *F1-Scores* der Querys *Q1* - *Q5*.

Mehr-Knoten-Pfade

Nach den Ein-Knoten-Pfaden werden jetzt die Resultate der Querys für Mehr-Knoten-Pfade betrachtet. Dafür wurden in Unterabschnitt 4.2.1 die zusätzlichen Querys (*Q11* - *Q13*) vorgestellt. Da *Q6* - *Q8* nur Verkürzungen von *Q9* zur Laufzeitbewertung und Funktionalitätstestung sind, sind ihre Resultate weniger aussagekräftig als die der anderen. Sie können jedoch im *Git Repository* nachgelesen werden. Hier nur Ausschnitte der Ergebnisse von *Q9* (siehe Tabelle 4.5) gezeigt. Die Ergebnisse von *Q10* und *Q11* sind sehr ähnlich. Die gesamte Ausgabe aller drei Querys ist unter `https://github.com/TbsHbnthl/master-s-thesis-link-prediction-on-large-scale-knowledge-graphs/tree/main/CRF_fertig/Resultate_fertig` zu finden.

node	precision	recall	f1-score	support
F	0.0000	0.0000	0.0000	54
...	...	...	...	...
CT INFUSED CHEST	0.0000	0.0000	0.0000	2353
CT NON-INFUSED CHEST	0.0000	0.0000	0.0000	415
CT PRE&POST UPPER ABDOMEN	0.0000	0.0000	0.0000	67
ITC	0.0000	0.0000	0.0000	2317
NTC	0.0000	0.0000	0.0000	359
Mandatory	0.1439	0.2548	0.1839	4686
Patient_Study	0.1416	0.4856	0.2193	4555
Procedure_Code_Sequence	0.1485	0.2346	0.1819	4633
micro avg	0.0732	0.0732	0.0732	62464
macro avg	0.0008	0.0028	0.0011	62464
weighted avg	0.0323	0.0732	0.0435	62464

Tabelle 4.5: Ausschnitt der Detailbetrachtung des Ergebnisses für *Q9*.

Diese Resultate zeigen, dass die Vorhersage bisher wenig präzise ist. Ein Grund dafür könnte die große Zahl zur Verfügung stehender Labels sein. Diese wird in den hier präsentierten Ausschnitten nicht richtig deutlich, beläuft sich aber, wie die vollständigen Ergebnisse zeigen, auf über 600. Viele dieser Labels sind null- bis hundertmal vertreten, sehr wenige kommen immerhin tausendfach vor. Unter Berücksichtigung der enormen Anzahl eingelesener Pfade ist das Ergebnis sehr breit gestreut. Dadurch werden für *Q9* - *Q11* niedrige *F1-Scores* von 0,0435, 0,0386 und 0,0447 erzielt.
Neben der Auswahl der Label können auch Features variiert werden. Diese werden, wie schon in Unterabschnitt 3.2.1 beschrieben, in der Methode `nodes2features(s,i,d)` festgelegt. Zusätzlich zu den Attributen der Knoten aus dem Graphen können noch weitere Features und ihre Auswirkungen auf das Ergebnis getestet werden.
Die Implementation der Erweiterung ist in Listing 4.1 zu sehen. Die zusätzlichen Features umfassen den Knotengrad, das in *Neo4j* gespeicherte Knotenlabel (das nicht zu verwechseln mit den hier verwendeten Labeln ist und daher als Knotentyp behandelt wird) sowie

die Typen der am häufigsten vertretenen ein- und ausgehenden Relationen der Knoten.

```
def nodes2features(s, i, d):

    ...

    # add feature of node degree
    ft['ndeg'] = graph.run(f"MATCH (n {temp1}nodeUID:'{node}'{temp2})-[r]-() RETURN ↩
        COUNT(r)").data()[0]['COUNT(r)']
    # add feature of node type
    ft['ntype'] = graph.run(f"MATCH (n {temp1}nodeUID:'{node}'{temp2}) RETURN labels(n)↩
        ").data()[0]['labels(n)']
    # type of the most common outgoing relationship
    ft['routtype'] = graph.run(f"MATCH(n {temp1}nodeUID:'{node}'{temp2})-[r]->() RETURN↩
         type(r) as type, COUNT(r) as score ORDER BY score DESC LIMIT 1")
    # type of the most common incoming relationship
    ft['rintype'] = graph.run(f"MATCH(n {temp1}nodeUID:'{node}'{temp2})<-[r]-() RETURN ↩
        type(r) as type, COUNT(r) as score ORDER BY score DESC LIMIT 1")
    # use closeness
    ft['prefAttach'] = graph.run(f"MATCH (n1 {temp1}nodeUID: '{node}'{temp2}) MATCH(n2 ↩
        {temp1}nodeUID: '{s[i][1]}'{temp2}) RETURN gds.alpha.linkprediction.↩
        preferentialAttachment(n1,n2)").data()[0]['gds.alpha.linkprediction.↩
        preferentialAttachment(n1,n2)']
```

Listing 4.1: Die modifizierte `nodes2features(s,i,d)`-Methode mit zusätzlichen Features.

Darüber hinaus können weitere Kriterien herangezogen werden, wie z.B. der wichtigste Nachbar des Knoten. Der Begriff des „wichtigsten“ Nachbarn ist dabei eine Frage der Definition. Es wurden bereits verschiedene Ähnlichkeitsmaße in Unterabschnitt 2.5.3 vorgestellt und als Kriterium zur Wahl der Label herangezogen. Zusätzlich können diese jedoch auch als Features für den Knoten herangezogen werden. Als Beispiel wird an dieser Stelle der Algorithmus für *Preferential Attachment* herangezogen und unter dem Begriff *prefAttach* als Feature gespeichert. Um die verschiedenen neuen Varianten unterscheiden zu können, werden die folgenden Bezeichnungen verwendet:

- *2addFeat*: Diese Variation nutzt zusätzlich zu den Attributen der Knoten noch den Knotengrad und den Knotentyp.
- *4addFeat*: Diese Version erweitert *2addFeat* um die am häufigsten auftretenden Typen der ein- und ausgehenden Kanten.
- *5addFeat*: Hier wird schließlich noch der *Score* des Algorithmus *Preferential Attachment* hinzugefügt.
- *only5addFeat*: Für diese Version werden nur die hier zusätzlich genannten Features verwendet und alle Knotenattribute ignoriert. Dadurch wird insgesamt nur mit sehr wenigen Features gearbeitet.

Die erzielten Ergebnisse unterscheiden sich jedoch nur sehr geringfügig von den bereits ohne Variation gefundenen. Für *Q9* - *Q11* bleiben Präzision, *Recall* und *F1-Scores* unter einem Wert von 0,1. Auch diese Ergebnisse sind im *Git Repository* zu sehen. Die Programme, die Variationen von *Q12* und *Q13* verwenden, scheitern erneut an nicht ausreichend vorhandenem Hauptspeicher.
Ein anderes Bild ergibt sich, wenn man, wie zuvor beschrieben, Pfadduplikate ausschließt. Auch hier werden die Resultate der *Link Predictions* einzeln untersucht und miteinander verglichen. Dabei werden wie zuvor auch grundsätzlich fünf verschiedene Varianten untersucht. Die Kennzeichnung verläuft analog, allerdings wird den Varianten das Präfix *noDups* vorangestellt, um zu kennzeichnen, dass es sich um die Programmvarianten handelt, die keine Pfadduplikate zulassen.

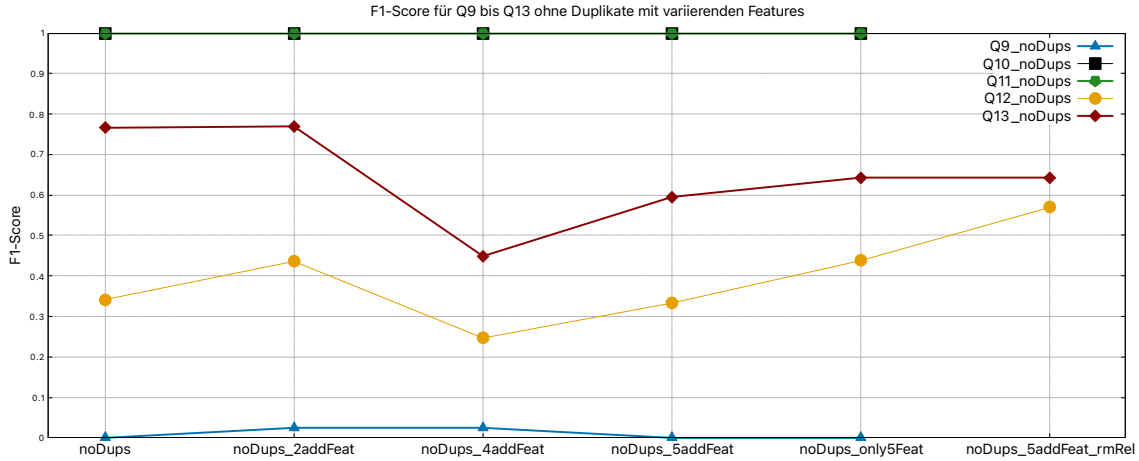

Abbildung 4.12: Vergleich der *F1-Scores* der Querys *Q9* - *Q13* ohne Duplikate.

Für die letzten beiden Querys *Q12* und *Q13* wird noch eine weitere Variante ergänzt, in der zusätzliche Features verwendet werden, allerdings nicht die, die auf die ein- und ausgehenden Relationen eingehen (*noDups_5addFeat_rmRel*). Dieses Addendum ergibt sich aus der Tatsache, dass im direkten Vergleich zwischen der Variante *noDups_2addFeat* und *noDups_4addFeat* bei diesen beiden Querys eine Verschlechterung des *F1-Scores* beobachtet wurde, obwohl die beiden weiteren Features hinzugefügt wurden. Dies könnte unter Umständen durch *Overfitting* hervorgerufen werden, also durch zu genaue Anpassung der Parameter an die Testdaten. Eine andere Erklärung kann auch eine möglicherweise ungünstige Wahl der Features sein.
Insbesondere *Q12* und *Q13* jeweils ohne Pfadduplikate unterscheiden sich an dieser Stelle auch. Während sich das Ergebnis für *Q12* durch die Entfernung der Relations-Features verbessert, ist dies bei *Q13* nicht der Fall. Diese Query liefert mit ausschließlich den knoteneigenen Attributen als Features das beste Ergebnis. Es scheint also auch von der Query abzuhängen, wie die Wahl der Features für ein optimales Ergebnis zu treffen ist.
Insgesamt ergibt sich für die Querys ohne Pfadduplikate das Resultat, das in Abbildung 4.12 abgebildet ist. In Abbildung 4.13 sind zusätzlich für *Q12* und *Q13* die Verläufe

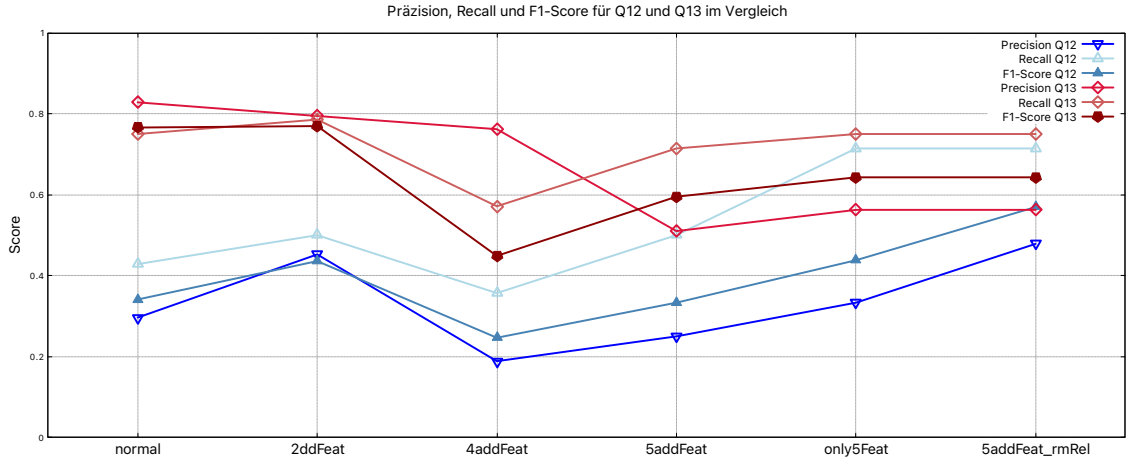

Abbildung 4.13: Vergleich des *F1-Scores*, der Präzision und des *Recalls* der Querys *Q12* und *Q13* ohne Duplikate.

für Präzision und *Recall* (vgl. Unterabschnitt 2.5.4) im Vergleich zum *F1-Score* abgebildet. Die Kürzel an der x-Achse sind wie oben zu verstehen. *Q9*, *Q10* und *Q11* werden nicht erneut dort abgebildet, da die zuvor in Abbildung 4.12 dargestellten Werte sich mit variierenden Features kaum oder gar nicht verändern.
Während bei *Q12* (blau) eine Kohärenz zwischen dem Wachstum der drei Kurven festgestellt werden kann, zeigt sich bei *Q13* mitunter ein anderes Bild. Gerade im Vergleich der Varianten *4addFeat* und *5addFeat* wird dies deutlich: Die Präzision nimmt in diesem Schritt stark ab, allerdings steigt der *Recall*, wodurch insgesamt auch der *F1-Score* steigt.

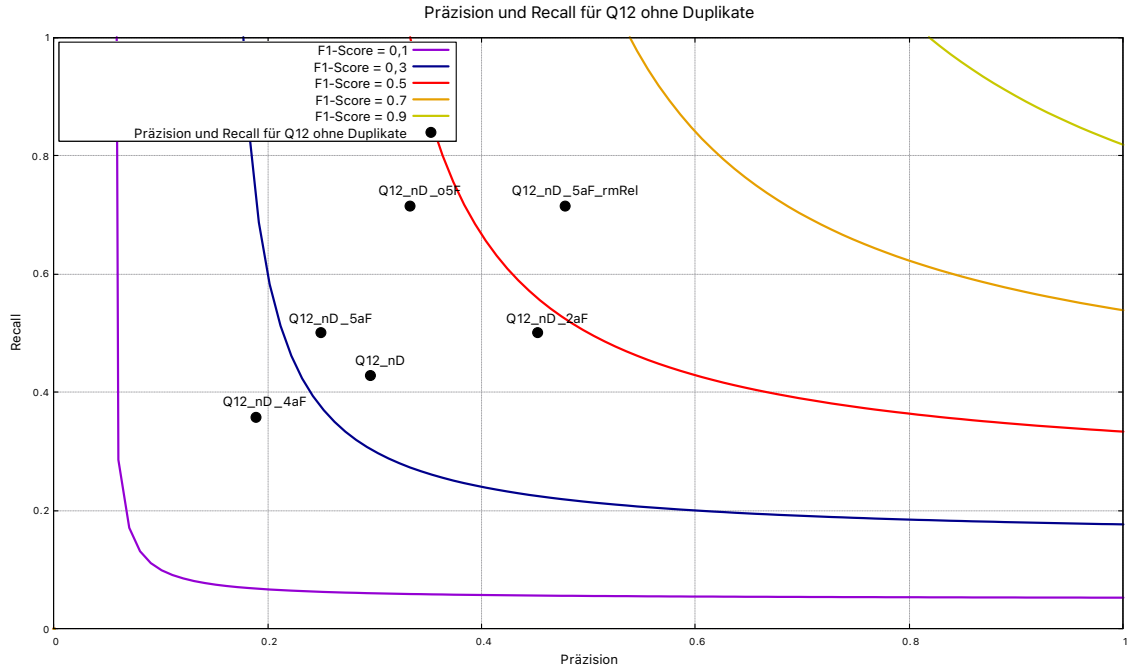

Abbildung 4.14: Präzison-*Recall*-Diagramm für die Versionen von *Q12*.

In Abbildung 4.14 und in Abbildung 4.15 sind die Ergebnisse noch einmal anders darge-

stellt. Dort werden Präzision und *Recall* gegeneinander geplottet. Die Höhenlinien korrespondieren mit der Darstellung des *F1-Scores* aus Abbildung 2.4 und dienen der visuellen Einordnung der Güte des Ergebnisses. Für *Q9* - *Q11* wurde wegen der nur sehr schwach fluktuierenden Ergebnisse, die zudem sehr positiv oder sehr negativ ausfallen, auf eine solche zusätzliche Visualisierung verzichtet.

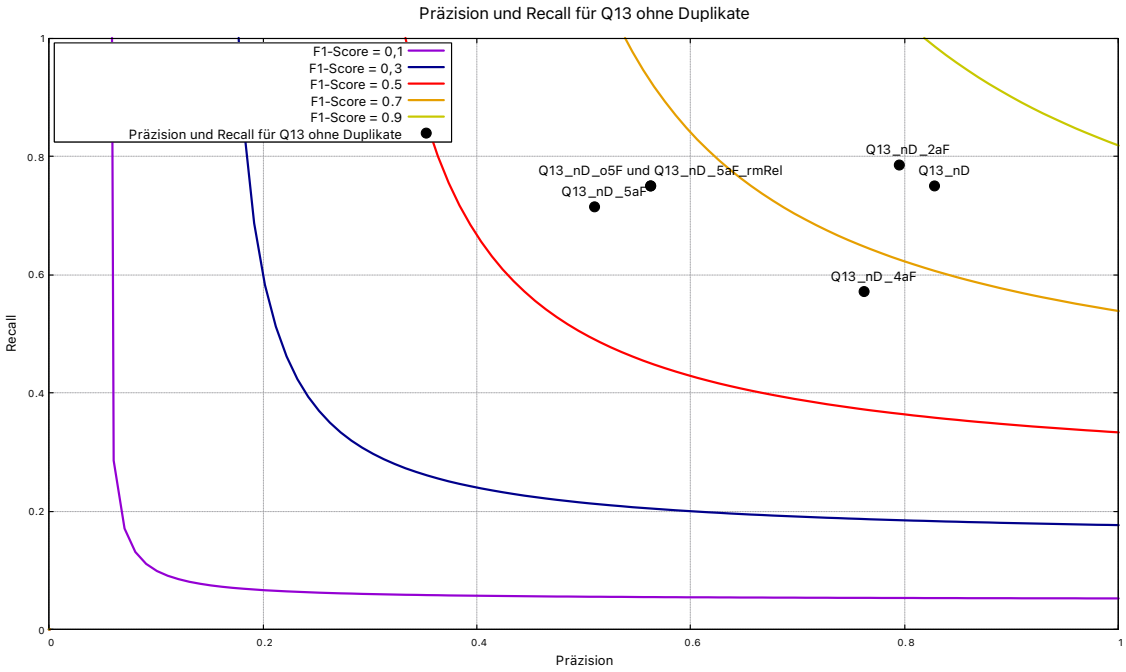

Abbildung 4.15: Präzison-*Recall*-Diagramm für die Versionen von *Q13*.

Für *Q12* erzeugt die Variante, die vier zusätzliche Features verwendet, das mit Abstand schlechteste Ergebnis. Sowohl die Grundvariante *noDups* als auch die Erweiterung *noDups_5addFeat* verzeichnen ebenfalls kein besonders gutes Ergebnis. Die Varianten *noDups_2addFeat* und *noDups_only5Feat* liegen immerhin in fast der Hälfte der Fälle richtig. Dabei ist interessant, dass die Variante, die keinerlei Knotenattribute als Features verwendet, ähnlich gut abschneidet wie die, die alle Knotenattribute und zwei zusätzliche Features nutzt. Das beste Ergebnis bietet *noDups_5addFeat_rmRel*. Es zeigt sich also, dass die am meisten ein- und ausgehenden Kantentypen keine gute Wahl für zusätzliche Features waren. Knotengrad und Knotentyp hingegen haben eine starke Verbesserung bewirkt. Für *Q13* zeigt sich in den Diagrammen ein anderes Bild. Als erstes springt sofort ins Auge, dass die Ergebnisse insgesamt deutlich besser ausfallen. Im Gegenzug ist hier allerdings zu beobachten, dass die zusätzlichen Feature-Variationen das Ergebnis zumindest nicht stark verbessern. Bis auf die Variante mit zwei zusätzlichen Features wird es sogar verschlechtert. Die genauen Resultate können im *Git Repository* abgerufen werden.

4.2.3 Fazit

Mit dieser Arbeit wurden mehrere Ziele verfolgt. Das erste war der generische Ansatz für den Import von Daten bildgebender Verfahren in einen Graphen. *Neo4j* stellt mit dem *Admin import* eine einfache Möglichkeit für den Import großer Datenmengen zur Verfügung. Dieser ist mithilfe des hier vorgestellten Skripts und der zugehörigen Konfigurationsdatei durch den Benutzer individuell konfigurierbar. Die Gestaltung des Graphen kann sehr stark durch den Benutzer vorgegeben werden. Das Programm ist flexibel und durch die lineare Laufzeit anwenderfreundlich und praktisch einsetzbar. Für die Kombination mit bereits vorhandenen Graphen und Datensystemen lässt sich mit wenigen Zeilen Code eine Schnittstelle bilden. Dafür sind lediglich die möglicherweise überlappenden Knotentypen zu identifizieren. Die zugehörigen *CSV*-Dateien des hier vorgestellten Programms können in einem nachfolgenden Programm eingelesen und die Knoten-IDs in *Sets* gespeichert werden. Ein Vorschlag zur Umsetzung ist in Listing 4.2 gegeben. Die einzulesende Datei wird *csv_file* genannt. Am Ende der Methode wird eine Liste der bereits vorhandenen Knoten zurückgegeben. Das vorgestellte Skript ist also auch mit anderen Skripten nach Einbindung der hier vorgestellten Methode kompatibel. Dies ist im Hinblick auf die Verknüpfung verschiedener Projekte, die im gleichen inhaltlichen Bereich angesiedelt sind, besonders wichtig.

```
def read_in_possible_duplicates(csv_file):
    # set to store already existing nodes
    read_in_nodes = set()
    # read in file
    with open(csv_file, newline='') as csvfile:
        reader = csv.reader(csvfile)
        # store first input in each row in set
        # (first value in the csv files is the node id)
        for row in reader:
            read_in_nodes.add(row[0])

    return read_in_nodes
```

Listing 4.2: Vorschlag zur Duplikatsvermeidung von Knoten und Kanten bei Verwendung mehrerer Datensätze und Skripte.

Das zweite Ziel bestand in der Anwendung von *NER* und *CRFs* auf Pfade aus einem Graphen. Für Ein-Knoten-Pfade wurden hervorragende Resultate für die ausgewählten Knoten erzielt. Die einzige Schwierigkeit stellte die Kommunikation zwischen *Neo4j* und Python durch die Bibliothek *py2neo* dar. Diese konnte durch eine kleine Anpassung des Codes umgangen werden.
Bei den Mehr-Knoten-Pfaden zeigen sich gemischte Ergebnisse und verschiedene Hürden. Auf der einen Seite sind die Schwierigkeiten zu nennen. Das erste Problem ist die Laufzeit für große Datenmengen. Die Abfrage der Features aus dem Graphen gestaltet

sich als sehr zeitaufwendig und skaliert dementsprechend mit der Menge der Daten. Das zweite Problem liegt in der steigenden Laufzeit für das maschinelle Lernen bei steigender Anzahl verwendeter Knoten im Eingabepfad. Gleichzeitig nehmen auch die Anforderungen an den zur Verfügung gestellten Hauptspeicher enorm zu. Beides hängt allerdings nicht nur mit der Länge des Eingabepfades, sondern auch mit der lokalen Umgebung der Pfade zusammen. Dünn besetzte Stellen des Graphen ermöglichen bessere Vorhersagen und liefern schneller Ergebnisse. Zudem hat eine zu große Anzahl an verfügbaren Labels gezeigt, dass die Ergebnisse unter der breiten Streuung der Zuordnung von Knoten und Labels leiden.
Auf der anderen Seite zeigen sich für Mehr-Knoten-Pfade sehr gute Ergebnisse, wenn die Pfadduplikate ausgeschlossen werden und somit die Pfadanzahl stark reduziert wird. Insbesondere die Wahl des Labels anhand der durch *Neo4j* zur Verfügung gestellten Algorithmen hat dabei die Güte des Ergebnisses stark beeinflusst. Bei der Wahl der Features zeigt sich, dass je nach Eingabepfad verschiedene Versionen zum jeweils besten Ergebnis führen. Interessant ist vor allem die Tatsache, dass für *Q12* und *Q13* ohne Duplikate bereits die ausschließliche Verwendung der zusätzlichen Features, also die *only5addFeat*-Variente, keine schlechten Ergebnisse liefert.

Kapitel 5

Ausblick

In der Zukunft können mit dem *DICOM-Importer* auch andere Patientendaten in einen Graphen importiert werden. Dieser kann durch zusätzliche Kontextumgebungen, wie beispielsweise fachspezifische Ontologien, mit Wissen angereichert werden. Darüber hinaus können mehrere Projekte verbunden werden. Darunter fällt neben dieser Arbeit zum Beispiel das Paper [72], das sich mit der effizienten Speicherung der Entitäten der PubMed Datenbank (siehe [73]) beschäftigt. Ein weiteres Beispiel ist das Paper [74], welches sich der effizienten Speicherung klinischer Daten von Patienten, die an Studien zur Demenzerkrankung teilgenommen haben, annimmt. Für diesen Zweck muss die bereits in Unterabschnitt 4.2.3 angesprochene Duplikatsvermeidung von Knoten beachtet werden.
Neben der Verknüpfung des Graphen mit anderen Datenquellen bietet sich aber vor allem eine erweiterte Untersuchung der Resultate der *Link Prediction* an. Dies gilt sowohl für den gleichen Datensatz mit anderen Querys als auch für ähnliche Datensätze. Dabei muss natürlich nicht nur die Wahl der Query, sondern vor allem auch die der Features weiter untersucht werden.
Das zentrale Problem hierbei ist die Laufzeit. Insbesondere die verwendete Methode `sent2features()` spielt dabei eine große Rolle. Bei Betrachtung der Mehr-Knoten-Pfade stieg allerdings auch die eigentliche Trainingszeit mit Veränderung der ausgewählten Pfade stark an. Es bleibt also zu untersuchen, wie diesen Problemen beizukommen ist. Möglicherweise bietet die Speicherung von Zwischenergebnissen einen ersten Ansatz, um bei Wiederholungen Zeit zu sparen. Bei Variation der Features führt dies allerdings zu Problemen und erfordert unter Umständen eine Umstrukturierung des Programms, so dass bereits aus dem Graphen ausgelesene Features aus einer gespeicherten Datei eingelesen und dann durch zusätzliche Features ergänzt werden können.
Ein weiteres technisches Problem stellt die Python Bibliothek *py2neo* dar. Es bleibt zu untersuchen, ob der von *Neo4j* selbst bereitgestellte *Neo4j Driver* (siehe [75]) eine bessere Möglichkeit zur Interaktion zwischen Python und der Datenbank bietet.
Schließlich können noch die durch *Neo4j* vorgegebenen Algorithmen zur *Link Prediction* verwendet werden. Die Resultate lassen sich anschließend mit denen der *Conditional*

Random Fields vergleichen. Es bleibt dafür aber zu prüfen, inwiefern die Algorithmen der umfangreichen Datenmenge gewachsen sind.

Literaturverzeichnis

[1] Edsger W. Dijkstra. Notes on Structured Programming. 1970.

[2] Biomedizinische Wissensgraphen. *https://www.scai.fraunhofer.de/de/geschaeftsfelder/bioinformatik/projekte/biomedizinische-wissensgraphen.html*, 2021.

[3] Heiko Paulheim. Knowledge graph refinement: A survey of approaches and evaluation methods. *Semantic Web*, 8:489–508, 2017.

[4] Chris Kemper. *Beginning Neo4j*. Apress, 1st edition, 2015.

[5] Ethem Alpaydin. *Maschinelles Lernen*. De Gruyter, 2014.

[6] Jörg Frochte. *Maschinelles Lernen - Grundlagen und Algorithmen in Python*. Hanser, 2021.

[7] Giuseppe Bonaccorso. *Machine Learning Algorithms*. Packt Publishing Ltd., 2017.

[8] Jenni A. M. Sidey-Gibbons and Chris J. Sidey-Gibbons. Maschine learning in medicine: a practical introduction. *BMC Medical Research Methodology*, 2019.

[9] Jørgen Bang-Jensen, Gregory Gutin. *Digraphs Theory, Algorithms and Applications*. Springer-Verlag, 2007.

[10] Reinhard Diestel. *Graph Theory*. Springer-Verlag, 2017.

[11] Renzo Angles. The Property Graph Database Model. *Alberto Mendelzon Workshop on Foundations of Data Management*, 2018.

[12] Jens Dörpinghaus, Andreas Stefan. Knowledge Extraction and Applications utilizing Context Data in Knowledge Graphs. *Proceedings of the 2019 Federated Conference on Computer Science and Information Systems*, 2019.

[13] Heiko Paulheim. Knowledge Graph Refinement: A Survey of Approaches and Evaluation Methods, year = 2017,. *Semantic Web*.

[14] Giannis Nikolentzos, George Dasoulas, and Michalis Vazirgiannis. k-hop Graph Neural Networks. *Neural Networks*, 130:195–205, 2020.

[15] D.A. Marcus. *Graph Theory: A Problem Oriented Approach*. Mathematical Association of America, 2015.

[16] M. Girvan and M. E. J. Newman. Community Structure in Social and Biological Networks. *Proceedings of the National Academy of Sciences*, 99(12):7821–7826, Jun 2002.

[17] Michael Kaufmann Andreas Meier. *SQL- NoSQL-Datenbanken*. Springer Vieweg, 8th edition, 2016.

[18] Greg Jordan. *Practical Neo4j*. Apress, 1st edition, 2014.

[19] Rene Steiner. *Grundkurs Relationale Datenbanken - Einführung in die Praxis der Datenbankentwicklung für Ausbildung, Studium und IT-Beruf*. Springer Vieweg, 9th edition, 2017.

[20] Andreas Meier Daniel Fasel. *Big Data - Grundlagen, Systeme und Nutzungspotentiale*. Springer Vieweg, hmd edition, 2016.

[21] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein. *Introduction to Algorithms - Third Edition*. Massachusetts Institute of Technology, 2009.

[22] Ingo Wegener. *Complexity Theory - Exploring the Limits of Efficient Algorithms*. Springer-Verlag, 2005.

[23] Martin Eric Müller Kurt-Ulrich Witt. *Algorithmische Informationstheorie*. Springer-Spektrum, 2020.

[24] Bildgebende Verfahren in der Krebsmedizin: Der Blick ins Körperinnere. 2021.

[25] Olaf Dössel. *Medizinische Bildgebung*. De Gruyter, 2014.

[26] Oleg S. Pianykh. *Digital Imaging and Communications in Medicine (DICOM): A Practical Introduction and Survival Guide*. Springer Publishing Company, Incorporated, 1st edition, 2010.

[27] Schlag P.M. Schlemmer, HP. Erweiterte digitale Bildgebung und molekulare Diagnostik - Aufbruch zu neuen Ufern in der Onkologie. 2020.

[28] Murray J.M. Strack C. et al. Kleesiek, J. Künstliche Intelligenz und maschinelles Lernen in der onkologischen Bildgebung. 2020.

[29] Peter Mildenberger and Marco Echelberg. Introduction to the DICOM standard. *European Radiology*, 2001.

[30] David G. Myers. *Psychologie*. Springer-Verlag Berlin Heidelberg, 3rd edition, 2014.

[31] Uwe Lorenz. *Reinforcement Learning*. Springer Vieweg, 1st edition, 2020.

[32] Sebastian Ruder. An Overview of Gradient Descent Optimization Algorithms. *Computing Research Repository*, 2016.

[33] Stephen Wright Jorge Nocedal. *Numerical Optimization*. Springer Vieweg, 2nd edition, 2006.

[34] G. Maeß H. Kieswetter. *Elementare Methoden der numerischen Mathematik*. Springer-Verlag Wien, 1974.

[35] David Liben-Nowell and Jon Kleinberg. The link prediction problem for social networks. In *Proceedings of the Twelfth International Conference on Information and Knowledge Management*, CIKM '03, pages 556–559, New York, NY, USA, 2003. Association for Computing Machinery.

[36] Mark Needham. Link Prediction with Neo4j Part 1: An Introduction. *https://medium.com/neo4j/link-prediction-with-neo4j-part-1-an-introduction-713aa779fd9*, 2019.

[37] Tao Zhou, Linyuan Lu, and Yi-Cheng Zhang. Predicting missing links via local information. *The European Physical Journal B*, 71(4):623–630, Oct 2009.

[38] Neo4j Graph Data Science Documentation. *https://neo4j.com/docs/graph-data-science/current/*, 2021.

[39] Alperen Degirmenci. Introduction to hidden markov models. 2015.

[40] L. Rabiner and B. Juang. An Introduction to Hidden Markov Models. *IEEE ASSP Magazine*, 3:4–16, 1986.

[41] Andrew Blake, Pushmeet Kohli, and Carsten Rother. *Markov Random Fields for Vision and Image Processing*. The MIT Press, 2011.

[42] Ermon Group Stanford Computer Science. Markov Random Fields - Ermon Group - Stanford Computer Science. *https://ermongroup.github.io/cs228-notes/representation/undirected/*, June 2021.

[43] Xiaojin Zhu. CS838-1 Advanced NLP: Conditional Random Fields. *Technical report, The University of Wisconsin Madison*, 2007.

[44] Mikhail Korobov. sklearn-crfsuite. *https://sklearn-crfsuite.readthedocs.io/en/latest/*, 2015.

[45] David Cournapeau et al. sklearn. *https://scikit-learn.org/stable/*, 2015.

[46] Conference on Computational Language Learning. *https://conll.org*, 2021.

[47] Erik F. Tjong Kim Sang. Introduction to the CoNLL-2002 Shared Task: Language-Independent Named Entity Recognition. pages 155–158, 2002.

[48] David M. W. Powers. What the F-measure doesn't measure: Features, Flaws, Fallacies and Fixes. 2019.

[49] sklearn: Metrics F1 Score. *https://scikit-learn.org/stable/modules/generated/sklearn.metrics.f1_score.html*, 2021.

[50] DICOM Standard Browser. *https://dicom.innolitics.com/ciods/ct-image*, 2021.

[51] DICOM Data Element Type. *http://dicom.nema.org/dicom/2013/output/chtml/part05/sect_7.4.html*, 2021.

[52] DICOM Model. *https://www.dicomstandard.org/current*, 2021.

[53] Justin Kirby. SPIE-AAPM Lung CT Challenge. *https://wiki.cancerimaging-archive.net/display/Public/SPIE-AAPM+Lung+CT+Challenge#190391977bccd3f8115f4374915bec3e6400b818*.

[54] Dublin Core Meta Initiative. *https://www.dublincore.org/specifications/dublin-core/*, April 2021.

[55] Registry of DICOM Data Elements. *http://dicom.nema.org/medical/dicom/current/output/chtml/part06/chapter_6.html*, 2021.

[56] DICOM Standard Browser. *https://dicom.innolitics.com/ciods*, April 2021.

[57] Dokumentation pydicom. *https://pydicom.github.io/pydicom/dev/index.html*, April 2021.

[58] configparser - Configuration file parser. *https://docs.python.org/3/library/configparser.html*, April 2021.

[59] glob - Unix Style Pathname Pattern Extension. *https://docs.python.org/3/library/glob.html*, April 2021.

[60] Neo4j Admin import. *https://neo4j.com/docs/operations-manual/current/tutorial/neo4j-admin-import/*, 2021.

[61] Neo4j Download Center. *https://neo4j.com/download-center/*, 2021.

[62] Simba database - public lung database. *http://www.via.cornell.edu/visionx/simba/*, 2021.

[63] py2neo. *https://py2neo.org/2021.1/*, 2021.

[64] multiprossessing - Process based parallelism. *https://docs.python.org/3/library/multiprocessing.html*, 2021.

[65] functools - Higher-order functions and operations on callable objects. *https://docs.python.org/3/library/functools.html*, 2021.

[66] Python Docs - Runtime Concatenating Strings. *https://docs.python.org/3/library/stdtypes.html*, April 2021.

[67] Python - Time Complexity. *https://wiki.python.org/moin/TimeComplexity*, 2021.

[68] Micha Gorelick and Ian Ozsvald. *High Performance Python: Practical Performant Programming for Humans*. O'Reilly Media, 2014.

[69] CPython Source Code - Split() Funktion. *https://github.com/python/cpython/blob/main/Objects/stringlib/split.h*, 2021.

[70] Regionales Rechenzentrum - High Performance Computing. *https://rrzk.uni-koeln.de/hpc-projekte/hpc*, 2021.

[71] MPI for Python. *https://mpi4py.readthedocs.io/en/stable/index.html*, 2021.

[72] Jens Dörpinghaus, Andreas Stefan, Bruce Schultz, and Marc Jacobs. Towards context in large scale biomedical knowledge graphs. *CoRR*, abs/2001.08392, 2020.

[73] Pubmed. *https://pubmed.ncbi.nlm.nih.gov/*, 2019.

[74] Jens Dörpinghaus, Sebastian Schaaf, Vera Weil, and Tobias Hübenthal. An efficient approach towards the generation and analysis of interoperable clinical data in a knowledge graph. In Maria Ganzha, Leszek A. Maciaszek, Marcin Paprzycki, and Dominik Slezak, editors, *Proceedings of the 16th Conference on Computer Science and Intelligence Systems, Online, September 2-5, 2021*, pages 59–68, 2021.

[75] Neo4j python driver. *https://neo4j.com/docs/api/python-driver/current/*, 2021.

Eidesstattliche Erklärung

Name: Tobias Hübenthal

Hiermit versichere ich an Eides statt, dass ich die vorliegende Arbeit selbstständig und ohne die Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe. Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten und nicht veröffentlichten Schriften entnommen wurden, sind als solche kenntlich gemacht. Die Arbeit ist in gleicher oder ähnlicher Form oder auszugsweise im Rahmen einer anderen Prüfung noch nicht vorgelegt worden. Ich versichere, dass die eingereichte elektronische Fassung der eingereichten Druckfassung vollständig entspricht.

_____ _____

Unterschrift Köln, den