Praktikum beim Deutschen Forschungszentrum für Künstliche Intelligenz

Forschungsgruppe Sprachtechnologien (DFKI-LT)
Projekt "Smart Data for Mobility" (SD4M)

DFKI Projektbüro Berlin
Alt-Moabit 91c
10559 Berlin

Zeitraum 15.02.2016 - 07.05.2016 (12 Wochen)

Praxisbericht

Tom Oberhauser

Matrikelnummer 798158
Studiengang Bachelor Medieninformatik
7. Fachsemester
Beuth Hochschule für Technik Berlin

E-Mail: tom@devfoo.de

Betreuer Prof. Christoph Knabe
Fachbereich VI - Informatik und Medien
Beuth Hochschule für Technik Berlin

Dr. Philippe Thomas
Forschungsgruppe Sprachtechnologie (DFKI-LT)
Deutsches Forschungszentrum für Künstliche Intelligenz

03.06.2016

Inhaltsverzeichnis

1 Einleitung

1.1 Vorstellung des Praktikumsbetriebes

Das Deutsche Forschungszentrum für Künstliche Intelligenz GmbH, im folgenden DFKI genannt, wurde 1988 gegründet. Es unterhält Standorte in Kaiserslautern, Saarbrücken, Bremen und ein Projektbüro in Berlin. Mit seinen 478 Mitarbeiten sowie 337 studentischen Mitarbeiten erforscht und entwickelt das DFKI innovative Softwaretechnologien auf der Basis von Methoden der Künstlichen Intelligenz. Die notwendigen Gelder werden durch Ausschreibungen öffentlicher Fördermittelgeber wie der Europäischen Union, dem Bundesministerium für Bildung und Forschung (BMBF), dem Bundesministerium für Wirtschaft und Technologie (BMWi), den Bundesländern und der Deutschen Forschungsgemeinschaft (DFG) sowie durch Entwicklungsaufträge aus der Industrie akquiriert.[1]

Ich absolvierte mein Praktikum innerhalb der *Forschungsgruppe Sprachtechnologie*, einer von 15 Forschungsgruppen[2] des DFKI, im Projektbüro Berlin. Die Gruppe wird geleitet durch Prof. Dr. Hans Uszkoreit.[3].

Meine Aufgabengebiete konzentrierten sich um das Projekt *"SD4M - Smart Data for Mobility"*. Das DFKI ist hier Teil eines Konsortiums aus 5 Partnern unter der Konsortialführung der *DB Systel GmbH*.[4]. Das Projekt *SD4M* wird in Abschnitt 2.1.1 auf Seite 3 näher erläutert.

1.2 Weg zur Praktikumsstelle

Herr Prof. Dr. habil. Alexander Löser aus dem Fachbereich VI der Beuth Hochschule für Technik Berlin machte mich auf den Praktikumsplatz aufmerksam. Durch seine Mitarbeit in Projekten im DFKI Projektbüro Berlin hatte er wargenommen, dass Bedarf und Interesse an Praktikanten und studentischen Mitarbeitern besteht und mich benachrichtigt. Ich habe mich daraufhin auf der Website des DFKI über die aktuellen Projekte informiert. Da ich mich sehr für Datenintegration interessiere und

[1] http://www.dfki.de/web/ueber
[2] http://www.dfki.de/web/ueber/orgaeinheiten
[3] http://www.dfki.de/lt/
[4] http://sd4m.net/konsortium

ich vor meinem Studium bereits Berufserfahrung auf diesem Gebiet gesammelt habe, fand ich das Projekt *SD4M - Smart Data for Mobility* sehr interessant. Es umfasst zwei Themenkomplexe: zum einen die Verknüpfung unterschiedlicher Datenquellen und zum anderen Methoden des *Natural Language Processings* bzw. des *Text Minings*. Das Thema Text Mining wurde kurz im Modul Datenbanksysteme im zweiten Semester angeschnitten und es hatte mich auch sehr interessiert. Nach einem persönlichen Gespräch mit Herr Uszkoreit, in dem ich mein Interesse für das Projekt darlegen konnte, kam es zur Vertragsunterzeichnung.

2 Tätigkeitsbereiche und Aufgaben

2.1 Überblick

Ich habe im Rahmen meines Praktikums am Projekt *SD4M - Smart Data for Mobility* mitgearbeitet. Meine konkrete Aufgabe war die Aufbereitung und Integration verschiedener Daten und Datenbanken, damit diese im Projekt Verwendung finden können. Meine Hauptdatenquelle waren die Geodaten des OpenStreetMap[1] Projekts.

2.1.1 Das Projekt SD4M

Das Projekt *Smart Data for Mobilty*[2], im folgenden *SD4M* genannt, ist ein Verbundprojekt eines Konsortiums aus 5 Partnern und wird vom Bundesministerium für Wirtschaft und Energie gefördert. Das Konsortium besteht aus 4 Wirtschaftsunternehmen und dem DFKI als Forschungseinrichtung.

- DB Systel GmbH (Konsortialführung)
- Deutsches Forschungszentrum für Künstliche Intelligenz GmbH
- idalab GmbH
-]init[AG für digitale Kommunikation
- PS-Team Deutschland GmbH Co. KG

Ziel des SD4M Projekts ist eine branchenübergreifende Serviceplattform, welche Daten der unterschiedlichen Mobilitätsanbieter (z.B. der Fahrplan der Deutschen Bahn) sowie öffentliche verfügbare strukturierte und unstrukturierte Daten (z.B. Twitter oder Facebook) miteinander verknüpft. Diese verknüpften Daten sind für Endnutzer, aber auch für Unternehmen oder die öffentliche Verwaltung von Interesse. In Abbildung 1 wird verdeutlicht, wie sich aus unstrukturierten Twitter-Daten Verspätungsinformationen für konkrete Verkehrsmittel extrahieren lassen. Diese können dann Endnutzern oder den Mobilitätsanbietern zur Verfügung gestellt werden.

[1] http://www.openstreetmap.org/
[2] http://sd4m.net/

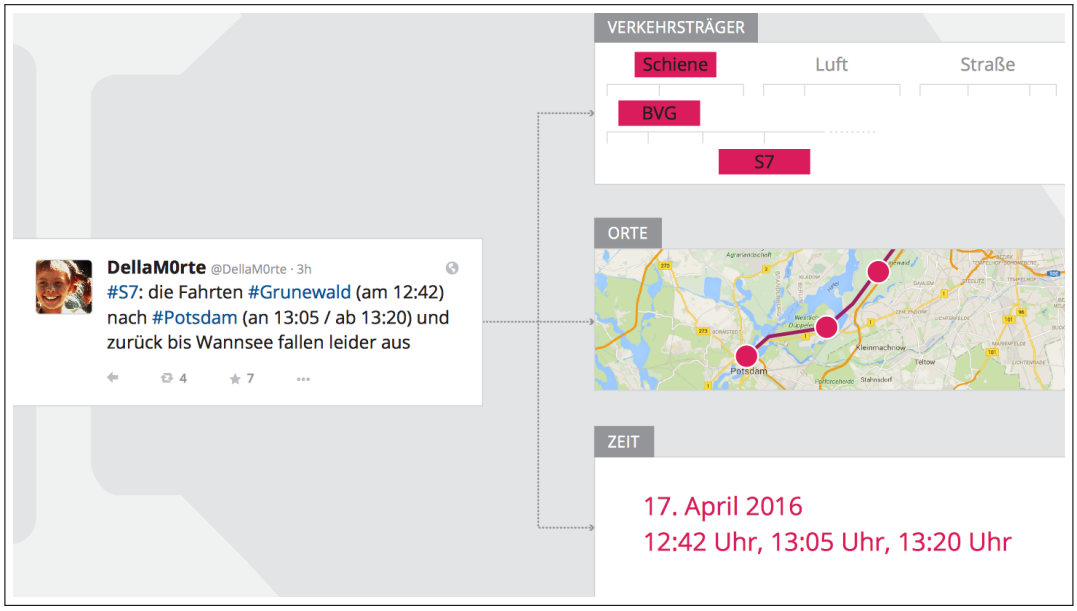

Abb. 1.: Verknüpfung eines Tweets mit Fahrplandaten[@Ing16]

2.2 Vorbereitung

Beim ersten Gespräch mit meinem Praktikumsbetreuer Dr. Philippe Thomas informierte ich mich, welche Programmierumgebungen und Programmiersprachen beim DFKI üblich sind. Ebenfalls erkundigte ich mich nach einer vorhandenen OpenStreetMap Datenbank und weiterer vorhandener Infrastruktur. Wir klärten, dass Java die geeignetste Sprache zur Lösung meiner Aufgaben war. Ebenfalls war eine OpenStreetMap Datenbank mit dem Datenbestand von Deutschland, sowie ein GitLab Repository Server vorhanden. Da ich auf meinem eigenen Notebook entwickeln wollte, installierte ich mir einen virtuellen Linux-Server mit einer PostgreSQL Datenbank um einen kleinen Teil der OpenStreetMap Daten lokal auf meinem Rechner zu haben. So konnte ich schneller entwickeln und mit einer wesentlich kleineren Datenbank testen. Als Java-Entwicklungsumgebung entschied ich mich für *JetBrains IntelliJ IDEA*[3] und zur Arbeit mit den Datenbanken für *JetBrains DataGrip 2016*[4].

2.3 Aufgaben

Meine Aufgaben während des Praktikums lassen sich in drei Teilbereiche gliedern. Zunächst sollte ich eine Straßenliste aus OpenStreetMap extrahieren. Anschließend verknüpfte und aggregierte ich Daten, welche von der Deutschen Bahn geliefert wurden, mit Daten aus OpenStreetMap. In den letzten Wochen meiner Praxisphase

[3] https://www.jetbrains.com/idea/
[4] https://www.jetbrains.com/datagrip/

gab es dann noch verschiedene Datenaufwertungs- und aufbereitungsaufgaben. Auf diese drei Themengebiete werden in diesem Abschnitt nun eingegangen.

2.3.1 Extraktion einer Straßenliste aus OpenStreetMap

Aufgabe

Meine erste Aufgabe bestand darin, eine Liste aller Straßen Deutschlands inclusive dazugehöriger Geodaten zu erstellen. Es sollte eine Java Anwendung erstellt werden, welche via Kommandozeilenargumenten konfiguriert wird, und die entsprechenden Ergebnisse in einer CSV-Datei ablegt. Im Ergebnis sollten pro zusammenhängendem Straßensegment die Daten

- **ID**, eine fortlaufende Nummer
- **Name**, der Name der Straße
- **LineString**, der geographische Straßenverlauf im *Well-Known-Binary (WKB)* Format (siehe Anlage A.1)
- **GeoJSON**, der geographische Straßenverlauf im *GeoJSON* Format (siehe Anlage A.2)

vorhanden sein. Diese Daten sollen anschließend zur geographischen Verortung von Straßen aus Tweets genutzt werden. Eine Beispielausgabe des fertigen Programms findet sich in Anhang A.7.

Lösung

Für meinen Anwendungsfall, die Filterung und Extraktion von Daten, war es notwendig, die OpenStreetMap Daten in einer Datenbank vorliegen zu haben. Die zwei populärsten Werkzeuge sind hierbei **osm2pgsql**[5] und **Osmosis**[6]. Der Hauptgrund hierfür ist dem Datenformat der OpenStreetMap Daten geschuldet. Wie im Anhang A.3 aufgezeigt, beinhalten die OpenStreetMap Daten *Nodes* (Punkte mit Koordinaten) und *Ways* (Linien aus Punkten). Die Importwerkzeuge haben die nützliche Eigenschaft, die Koordinaten aller Punkte eines Weges zu aggregieren. Das ermöglicht die direkte Extraktion der Geometrie eines Ways ohne sich für jeden Way die Koordinaten aus den Daten selbst aggregieren zu müssen. Beide Werkzeuge erzeugen unterschiedliche Datenbankschemata.

[5] https://github.com/openstreetmap/osm2pgsql
[6] https://github.com/openstreetmap/osmosis

Das DFKI besitzt Datenbanken, in der die OpenStreetMap Daten Deutschlands mit osm2pgsql sowie Osmosis importiert wurden. Ich informierte mich darüber, welches Schema am verlustfreisten ist.

> "osm2pgsql is mainly written for rendering data with data. So it only imports tags which are going to be useful for rendering. ... whereas osmosis and osmium more geared towards truthfully representing a full OSM data set." [@GIS12]

Ich entschied mich für die Datenbank im Osmosis-Schema, da diese den wahren Datenbestand am besten repräsentiert.

Eine physikalisch vorhandene Straße wird durch mehrere aneinanderhängende *Ways* repräsentiert. Wenn sich im Straßenverlauf ein Attribut (Ein *Tag*) der Straße ändert, zum Beispiel die zulässige Höchstgeschwindigkeit, muss ein neues Teilstück diese neuen Gegebenheiten abbilden. Ein Beispiel dafür findet sich sich in Anlage A.4. Das Ziel war nun, eine Liste zusammenhängender Ways mit gleichem Name zu aggregieren und diese zu exportieren. Dazu plante ich folgenden Ablauf:

1. Ermittlung aller Straßennamen aus der Datenbank
2. Abfrage aller Ways pro Straßenname
3. Trennung zusammenhängender Ways in Gruppen
4. Export in Zieldatei

Als kompliziertestes Problem stellte sich die Trennung zusammenhängender Way-Gruppen dar. In Abbildung 2 ist ein Kartenausschnitt mit allen Ways mit dem Name *"Berliner Straße"* dargestellt. In diesem sind 5 Way-Gruppen dargestellt. Optisch ist das Problem der Trennung einfach zu lösen, jedoch war mir nicht auf Anhieb klar wie ich das programatisch lösen sollte. Ich habe die Problemstellung dann weiter abstrahiert und kam auf die Idee die Ways und die dazugehörigen Nodes als Graph zu verstehen. Schematisch ist das in Abbildung 3 dargestellt. Ich ging davon aus, dass dieses Graphen-Problem von einer bestehenden Graphen-Bibliothek gelöst werden kann und fand *JGraphT*[7]. Mit JGraphT konnte ich alle Nodes als Knoten und alle Ways als Kanten in einem Graph abbilden. Anschließend konnte der Graph auf zusammenhängende Elemente untersucht und getrennt werden. Ein kurzes Beispiel zur Verwendung von JGraphT zur Trennung eines Graphen in separate verbundene Teile wird in Anhang A.5 gezeigt. Schwierig war hierbei allerdings dass nur Mengen von zusammenhängenden Knoten zurückgegeben wurden, die Kanten jedoch verloren gingen. Ich musste mir also im Voraus die Verbindungen der einzelnen Knoten speichern und diese anschließend erneut verarbeiten. Der vollständige Quellcode der entsprechenden Klasse `GraphSeparator.java` ist in Anhang A.6 angefügt.

[7] http://jgrapht.org

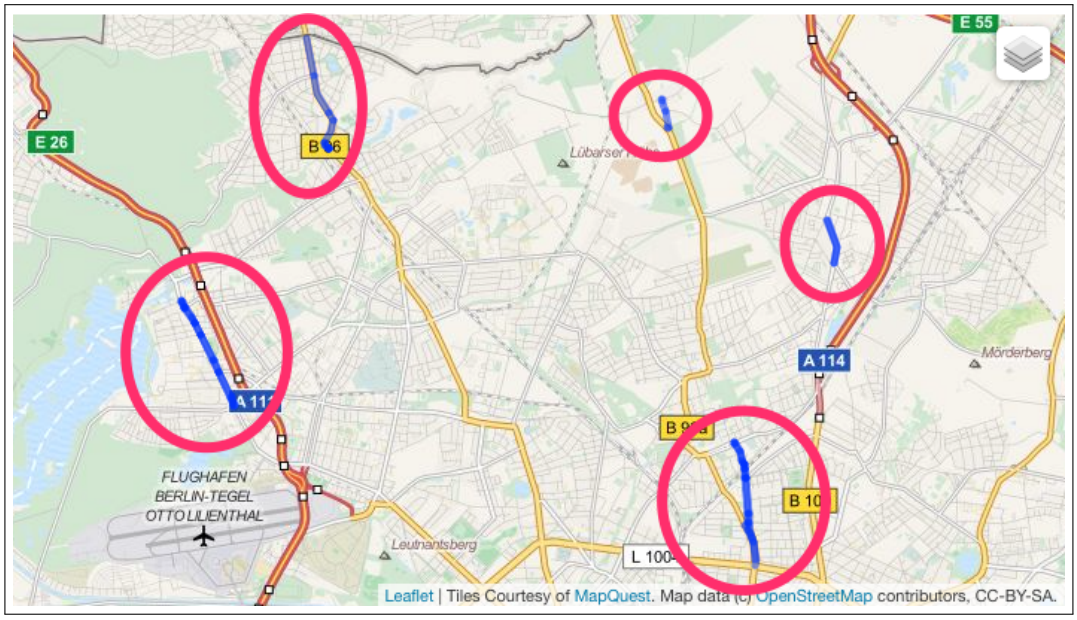

Abb. 2.: Kartenauszug verschiedener Straßengruppen am Beispiel “Berliner Straße”

Programmstruktur

Für viele Problemstellungen innerhalb des zu erstellenden Programms wurden externe Bibliotheken verwendet (Siehe Tabelle 2.1). Die Ahängigkeiten wurden mit *Apache Maven*[8] verwaltet.

Tab. 2.1.: verwendete externe Bibliotheken

Problem	**verwendete Bibliothek**
Datenbankanbindung PostgreSQL	PostgreSQL JDBC Driver JDBC 4.1
Verarbeitung von Kommandozeilenargumenten	Apache Commons CLI
Verarbeitung von JSON Objekten	com.googlecode.json-simple
Aufbau eines Graphen	org.jgrapht.jgrapht-core

Das Programm erhielt den Namen `OsmosisStreetExtractor`. Die Struktur wird in Abbildung 4 gezeigt. Die Klasse `Main` erzeugt hierbei eine Instanz vom Typ `OsmosisDB` welche die Datenbankverbindung verwaltet. Innerhalb von `OsmosisDB` werden nun alle Straßennamen Deutschlands ermittelt und in einer Liste aus `StreetGroup` Instanzen gespeichert. Anschließend verarbeitet `Main` alle `StreetGroup` Objekte und sendet sie zur Trennung in zusammenhängende Segmente an den `GraphSeparator`. Die nun getrennten Ergebnisse pro Straßenname werden in Instanzen vom Typ `ClusterResult` abgelegt, welche dann in die Ausgabedatei geschrieben werden. Die Klasse `PstQueries` hält alle *PreparedStatements* für die Datenbankabfragen.

Eine Beispielausgabe des Programms findet sich in Anhang A.7.

[8] http://maven.apache.org

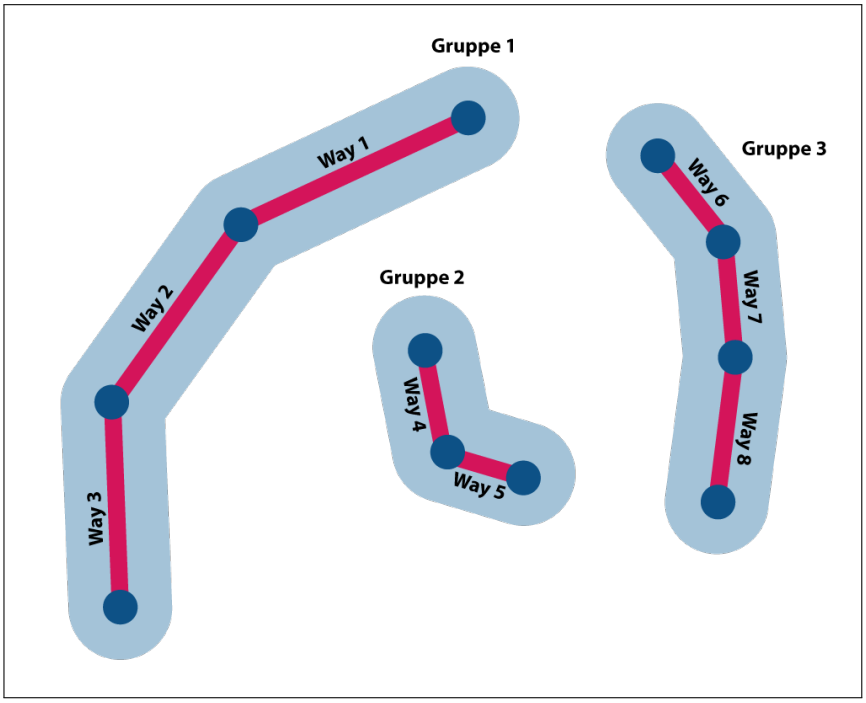

Abb. 3.: Schematische Darstellung des Problems der Weggruppenfindung

Schwierigkeiten

Das Testen auf einer kleinen Datenbank beschleunigte zwar die Entwicklung, aber erst der erste Test auf der Hauptdatenbank zeigte massive Performanceprobleme auf. Ich hatte wollte meine Anwendung möglichst speicherschonend gestalten und fragte die Ways zu den separaten Straßennamen einzeln aus der Datenbank ab. Dies funktionierte auf meiner lokalen und wesentlich kleineren Testdatenbank sehr gut, jedoch auf Datenbank mit den gesamten Daten Deutschlands annähernd gar nicht. Eine Messung zeigte dass die Abfragen auf der Deutschland-Datenbank mit 300ms Abfragezeit der Flaschenhals waren. Ich schrieb letztendlich die Datenbankanbindungsklasse `OsmosisDB` fast komplett um. Statt vieler kleiner Abfragen, welche wenig Daten zurücklieferten, wurden nun wenige Abfragen an die Datenbank gestellt, die jedoch viele Daten lieferten. Die Verarbeitung erfolgte dann komplett im Arbeitsspeicher und war performant genug.

Ebenfalls dauerte es eine Weile bis ich die Trennung der einzelnen Straßengruppen als Graphenproblem verstand.

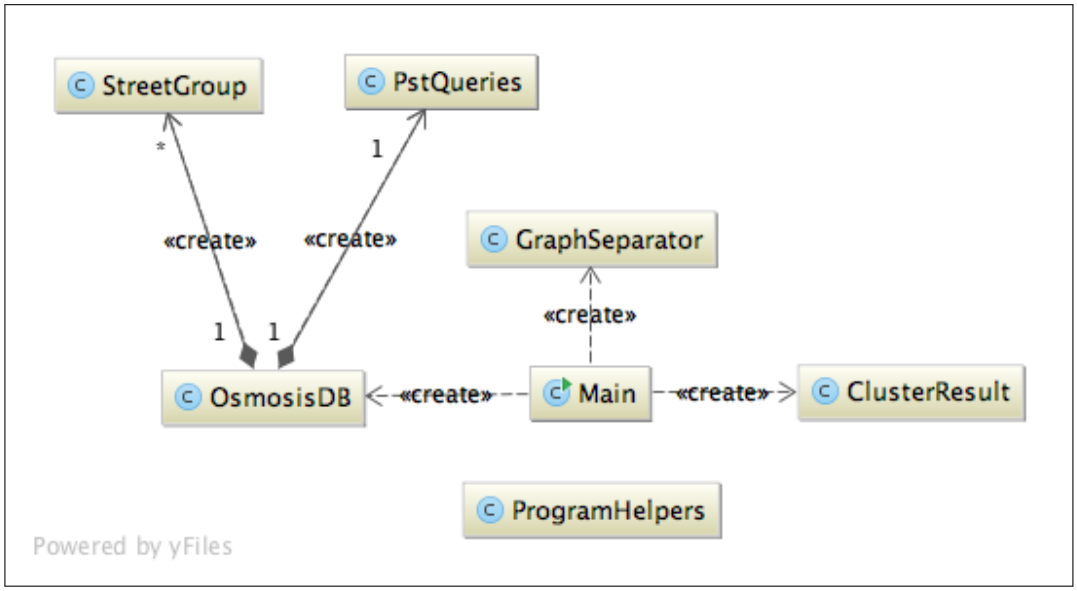

Abb. 4.: Klassendiagramm des fertigen Programms

2.3.2 Verknüpfung von Daten der Deutschen Bahn mit Daten aus OpenStreetMap

Aufgabe

Das DFKI arbeitet mit Daten, welche von der Deutschen Bahn bereitgestellt wurden, die alle größeren Bahnhöfe Deutschlands, sowie die Linien und Abfahrtszeiten aller Zugverbindungen enthalten. Diese liegen im *General Transit Feed Specification (GTFS)* Format[9] vor, einem Datenformat zur Veröffentlichung von Fahrplänen und dazugehörigen Informationen wie zum Beispiel geographischer Positionen der Bahnhöfe. Hierzu ist auch eine kurze Beschreibung in Anhang A.8 verfügbar.

Meine Hauptaufgabe bestand darin, eine Abbildung von Bahnhöfen aus den GTFS-Daten der Deutschen Bahn (im folgenden DB-GTFS Daten genannt) auf Objekte aus den OpenStreetMap Daten zu erstellen. Mit Hilfe dieser Abbildung ist es möglich, zu Bahnhöfen oder Zugstrecken erweiterte Informationen aus OpenStreetMap zu beziehen, wie zum Beispiel das Bundesland oder Straßen in der Nähe des Bahnhofes.

Als Nebenaufgabe sollte geprüft werden, in wiefern sich aus den kombinierten Daten ein geographisch exakter Streckenverlauf eines Zuges für die graphische Darstellung ermitteln lässt.

[9] https://developers.google.com/transit/gtfs/

Lösung

Da GTFS-Daten nur aus Textdateien bestehen, sollten diese zur effizienten Verarbeitung in einer Datenbank vorhanden sein. Das DFKI hatte die DB-GTFS Daten bereits in einer Datenbank vorliegen, jedoch wollte ich eine lokale Kopie in meiner eigenen Entwicklungsumgebung. Zum Import wurde der frei verfügbarer Importer *OpenTransitTools/gtfsdb*[10] verwendet.

Nun verschaffte ich mir einen Überblick über die vorhandenen Daten und klärte die Frage wie ich die Objekte beider Datenbanken miteinander in Verbindung bringen konnte. Ein Bahnhof aus den DB-GTFS Daten wird durch Name und geographischen Koordinaten spezifiziert.

Einen Bahnhof innerhalb von OpenStreetMap auszumachen war ein wenig komplizierter. Eine Recherche im OpenStreetMap Wiki[11] und viele Versuche ergaben dass ein Bahnhof in OpenStreetMap durch einen Node repräsentiert wird, der eine bestimmte Tag-Kombination aufweist. Genauer gesagt ein Node, der

- entweder Im Tag `railway` einen der Werte `"station"`, `"halt"` oder `"stop"`
- oder im Tag `public_transport` den Wert `"stop_position"` und im Tag `train` den Wert `"yes"`

aufweist.

Ich hatte nun also zwei Objektmengen. Auf der einen Seite die Menge aller DB-GTFS Bahnhöfe und auf der anderen Seite die Menge aller OpenStreetMap Bahnhöfe. Nun musste Ich einen Weg finden um zu beschreiben welche DB-GTFS Bahnhöfe zu welchem OpenStreetMap Bahnhöfen gehören.

Im Gespräch mit meinem Betreuer stellte sich heraus, dass es sich hierbei um ein klassisches *Klassifikationsproblem* handelt. Es muss für jede Objektkombination die Frage beantwortet werden, ob diese zusammengehören oder nicht. Um dies festzustellen hatte ich zwei Attribute, zum einen die geographische Position und zum anderen den Name.

Sich alleine auf die Position zu beziehen war nicht möglich, da diese, wie in Abbildung 5 gezeigt, in beiden Datenquellen leicht voneinander abwichen. Ebensowenig konnte eine Bereichssuche genutzt werden, da nicht zugehörige Bahnhöfe teilweise sehr nah beieinander, teilweise jedoch auch sehr fern voneinander weg liegen. und hier kein exakter Grenzwert gefunden werden kann. In Abbildung 6 sind die beiden zusammengehörigen Bahnhofsobjekte “Berlin-Wuhlheide” sowie “S-Wuhlheide” nur

[10] https://github.com/OpenTransitTools/gtfsdb
[11] http://wiki.openstreetmap.org/wiki/Railways

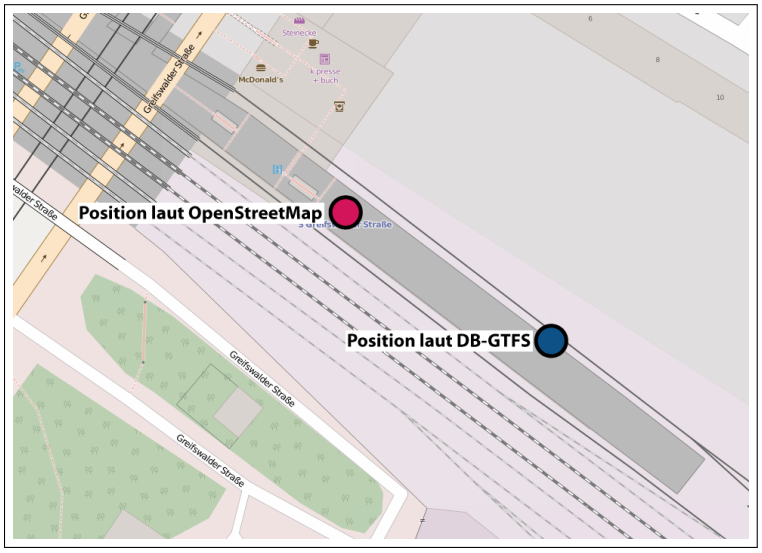

Abb. 5.: Abweichung der geographischen Position in beiden Datenbanken am Beispiel S-Bahnhof Greifswalder Straße Berlin

knapp näher beieinander als das fremde Bahnhofsobjekt “Wuhlheide-Parkeisenbahn”. Hier müsste man die Grenze bei ca. 100m ziehen. In Abbildung 7 benötigt man jedoch ca. 120m um die Zugehörigkeit zu erzeugen. Daraus folgt, dass die Entfernung allein ebenfalls kein Kriterium sein kann. Der Name konnte ebenfalls nicht

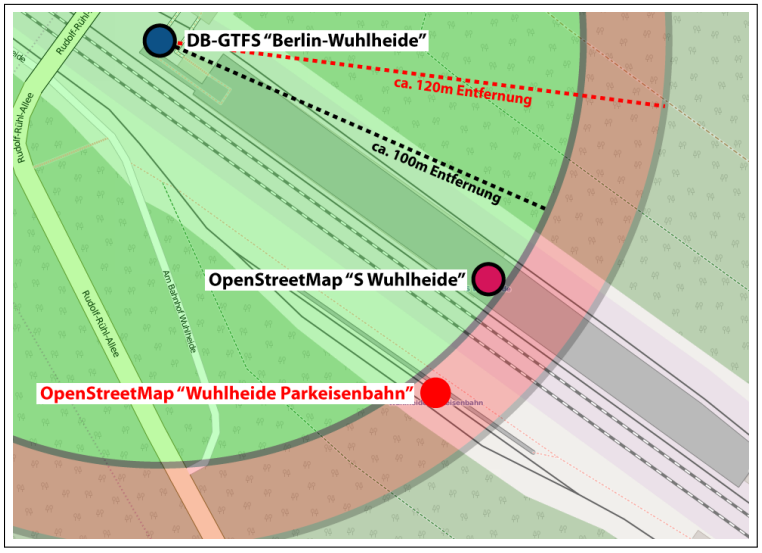

Abb. 6.: Entfernungen zwischen zusammengehörigen und nicht zusammengehörigen Bahnhofsobjekten

alleinstehend als Schlüssel verwendet werden. Zum Beispiel existiert in den DB-GTFS Daten ein Objekt names “Berlin Hbf” und in den OpenStreetMap Daten ein Objekt names “S Hauptbahnhof”. Diese Objekte gehören zusammen, ihre Bezeichnungen sind jedoch stark unterschiedlich.

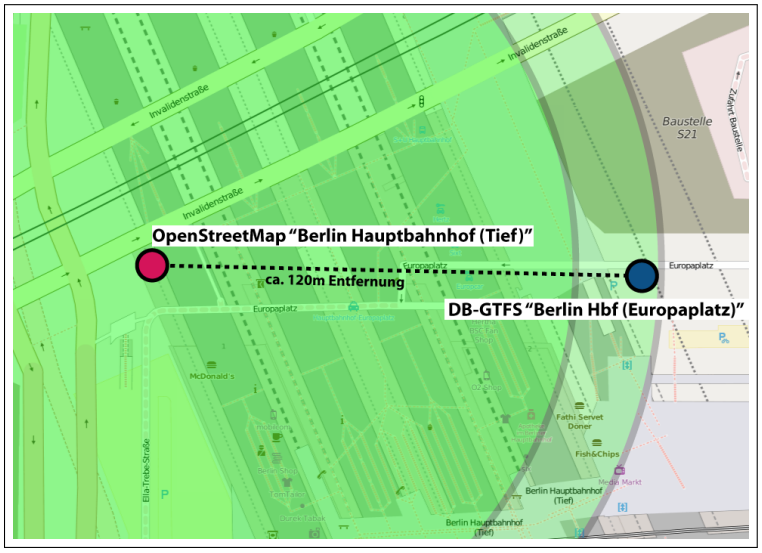

Abb. 7.: Entfernungen zwischen zusammengehörigen Bahnhofsobjekten 120m

Tab. 2.2.: Beispiel aus dem erstellten Trainingsdatensatz zur Klassifikation zusammenhängender Bahnhofsobjekte

gtfs	osm	levenshtein	geoDistance	match
Bernauer Str. (U), Berlin	U Bernauer Straße	0,125	0,0008372348	TRUE
Bernauer Str. (U), Berlin	U Voltastraße	0,6136363636	0,0053660279	FALSE
Bernauer Str. (U), Berlin	U Rosenthaler Platz	0,6242424242	0,0097482731	FALSE
Bernauer Str. (U), Berlin	S Nordbahnhof	0,7965367965	0,0099496517	FALSE
Yorckstr. (S+U), Berlin	U Yorckstraße	0,3164983165	0,0002386729	TRUE
...	...	...	...	...

Mir wurde klar, dass ich die Klassifikation nur durch die Kombination beider Attribute (Name sowie geographische Entfernung) vornehmen konnte. Zum Vergleich der String Ähnlichkeit entschied ich mich für eine normalisierte Variante der *Levenshtein-Distanz*, welche einen Wert im Interval $[0, 1]$ ermittelt, wobei 0 bedeutet beide Strings sind gleich und 1 dass beide Strings vollständig unterschiedlich sind. Details zur genauen Methode finden sich in Anhang A.9.

Da es sich um ein Klassifikationsproblem handelte, unternahm ich Versuche das Problem mit einem *Naive Bayse-Klassifikator* zu lösen. Diesen implementierte ich zusammen mit meinem Betreuuer in der Programmiersprache R. Auf den Klassifikator in R wird an dieser Stelle nicht weiter eingegangen, da der Aufwand, diesen anschließend in das Java-Programm zu integrieren in keinem Verhältnis zum Nutzen stand. Jedoch half mir der, speziell für den Klassifikator erstellte, Trainingsdatensatz (siehe Tabelle 2.2), gute Schwellwerte der beiden Merkmale Namensgleichheit und Entfernung zu finden.

Ich erstellte ein Hilfsprogramm, welches verschiedene Kombinationen aus Schwellwerten testete. Nachdem es versucht hatte eine Zusammengehörigkeit vorherzusagen, wurden die Kennzahlen *Precision (Genauigkeit)*, *Recall (Trefferquote)* und das

F-Maß berechnet. Dies sind Kennzahlen zur qualitativen Bewertung eines Klassifikators. (Näheres hierzu in Anhang A.10).

Die beste Vorhersage, ob zwei Bahnhöfe zusammengehörten, erzielte ich, wenn die Levenshtein-Distanz $\leq 0,825$ und die geographische Distanz $\leq 0,0045$ war. Diese Kombination erzielte auf dem Trainingsdatensatz das beste F-Maß mit einem Wert von $0,9842$.

Nun hatte ich eine Lösung, mit der ich die zu einem DB-GTFS Bahnhofsobjekt zugehörigen OpenStreetMap Bahnhofsobjekte finden konnte. Jedoch gab es innerhalb der DB-GTFS Daten auch teilweise mehrere Bahnhofsobjekte pro Bahnhof. Dies löste ich, in dem ich einen Bahnhof als Sammlung aus verschiedenen DB-GTFS-, sowie OpenStreetMap Objekten verstand. Diese Sammlung repräsentierte ich als `StationCluster`-Objekte (siehe Abbildung 8).

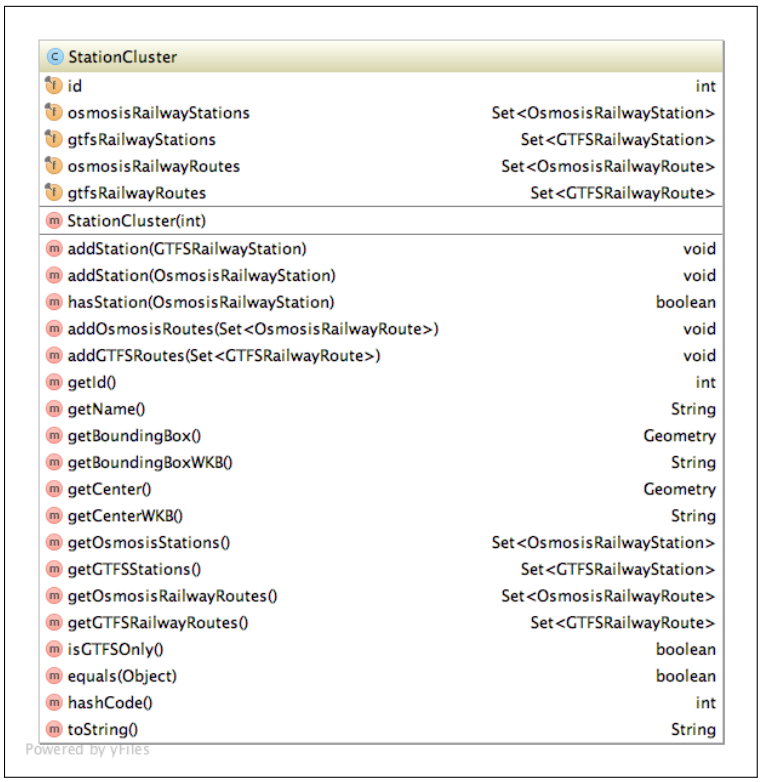

Abb. 8.: Die Klasse StationCluster

Um diese zu befüllen, iterierte ich über alle DB-GTFS Objekte und ermittelte pro Objekt alle zugehörigen OpenStreetMap Objekte. Anschließend prüfte ich, ob bereits ein `StationCluster` existiert, der eins dieser OpenStreetMap Objekte enthält. Wenn ja, dann wurden die DB-GTFS-, sowie die OpenStreetMap Objekte diesem `StationCluster` hinzugefügt. Wenn nicht, dann wurde ein neuer `StationCluster` angelegt.

Die so erzeugten `StationCluster` Objekte waren nun das Kernstück meines Ergebnisses und wurden in die Ausgabedatei exportiert. Des weiteren wurde pro Gruppe ein neuer Name erzeugt und Koordinaten ermittelt, welche den Mittelpunkt aller Objekte sowie das umschließende Rechteck abbilden.

Ein Ergebnisbeispiel findet sich in Anhang A.11.

Programmstruktur

Es wurden erneut externe Bibliotheken (siehe Tabelle 2.3). verwendet.

Tab. 2.3.: verwendete externe Bibliotheken

Problem	**verwendete Bibliothek**
Datenbankanbindung PostgreSQL	PostgreSQL JDBC Driver JDBC 4.1
Verarbeitung von Kommandozeilenargumenten	Apache Commons CLI
Ermittlung von String-Distanzen	info.debatty.java-string-similarity
Ermittlung von Geo-Distanzen	Vivid Solutions Inc. JTS Topology Suite
erweiterte Datenstrukturen, z.B. Tabellen	Google Guava

Das Programm erhielt den Namen `GTFS2OSMRailwayLinker`. Die Struktur des kompletten Programms ist in Abbildung 9 ersichtlich. Die Klasse `Main` erzeugt einen `CSVExporter`, welcher für den Export ins CSV-Format verantwortlich ist. Anschließend werden die Datenbankschnittstellen `GTFSDB` und `OsmosisDB`, sowie der, für die Gruppenbildung zuständige, `Clusterizer` angelegt. Dies ist in Abbildung 10 zu erkennen. Der `Clusterizer` iteriert nun über die DB-GTFS Objekte und legt `StationCluster` an (siehe Abbildung 11). Diese werden letztendlich über den `CSVExporter` exportiert.

Schwierigkeiten

Als Nebenaufgabe sollte geprüft werden, in wiefern sich aus den kombinierten Daten ein geographisch exakter Streckenverlauf eines Zuges für die graphische Darstellung ermitteln lässt. Nach zahlreichen Versuchen stellte sich heraus, dass dies mit den gegebenen Daten nicht möglich ist. Die Versuche nahmen sehr viel Zeit in Anspruch. Zwar sind aus den DB-GTFS Daten die einzelnen Bahnhöfe pro Linie ableitbar, jedoch existiert keine zuverlässige Information über den exakten Streckenverlauf. Das Hauptproblem an dieser Stelle ist, dass das Streckennetz der Deutschen Bahn nur eine Art Straßenkarte ist. Es existieren keine öffentlich zugänglichen Informationen, aus denen der genaue Verlauf, zum Beispiel eines bestimmten Fernzuges, ersichtlich ist.

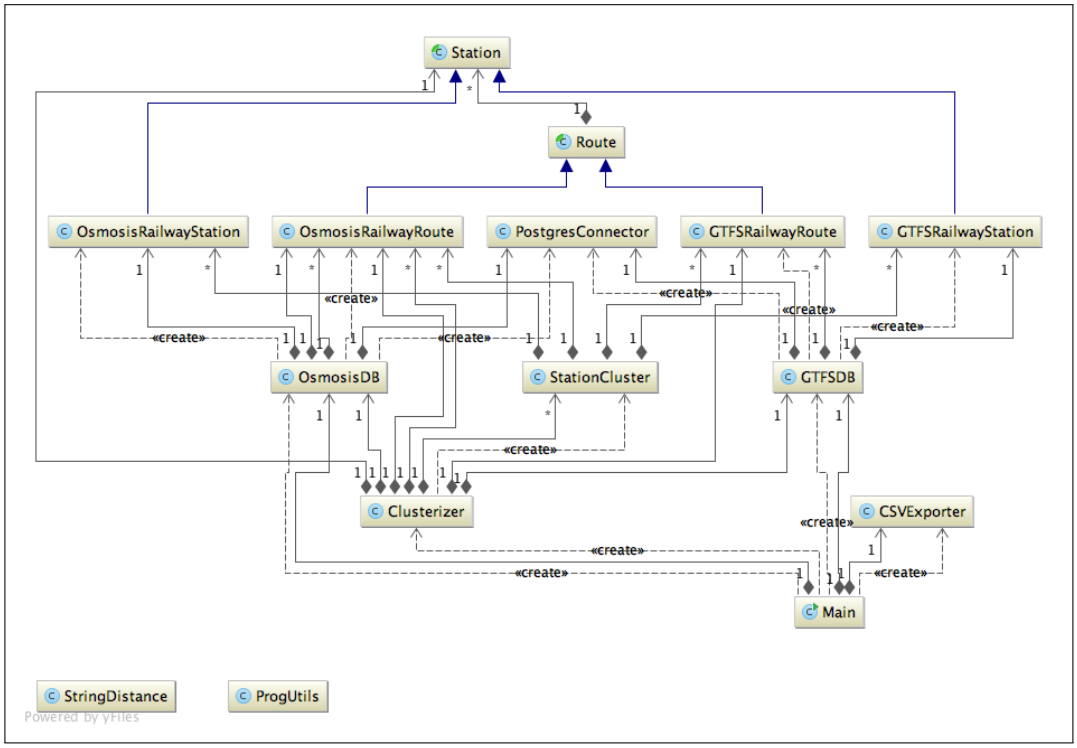

Abb. 9.: Klassendiagramm GTFS2OSMRailwayLinker

2.3.3 Datenaufwertung und Datenaufbereitung

Zum Ende meiner Praxisphase bekam ich noch mehrere kleine Aufgaben. Hauptsächlich ging es hier um Aufwertung von Daten oder dem gezielten Export von Daten aus offenen Datenquellen.

Beispielsweise wurde die Aufgabe gestellt, alle Angaben zur Bevölkerungsgröße aus *Wikidata*[12] zu extrahieren. Dies wurde gelöst, in dem mir eine heruntergeladene Kopie der Daten aus Wikidata zur Verfügung gestellt wurde, und ich diese mit einem Python-Script nach den gesuchten Daten durchsuchte und diese in eine CSV-Datei exportierte.

Eine weitere Aufgabe war es, eine Datenquelle für alle politischen Grenzen der Welt zu suchen und gegebenenfalls eine Datei, welche diese beinhaltet, zu erstellen. Nach einer Recherche stellte ich fest, dass solche Daten zwar kostenfrei für Staatsgrenzen zur Verfügung stehen (zum Beispiel von *GeoNames*[13]), jedoch sind Daten für kleinere administrative Bereiche wie zum Beispiel Bundesländer nicht kostenfrei auffindbar. Meine Idee war es nun, dies mit OpenStreetMap Daten zu lösen. In Abschnitt 2.3.1 auf Seite 5 erwähnte ich kurz das Werkzeug **osm2pgsql**[14]. Eine, mit diesem Werkzeug importierte, OpenStreetMap Datenbank eignet sich sehr

[12] www.wikidata.org
[13] http://www.geonames.org
[14] https://github.com/openstreetmap/osm2pgsql

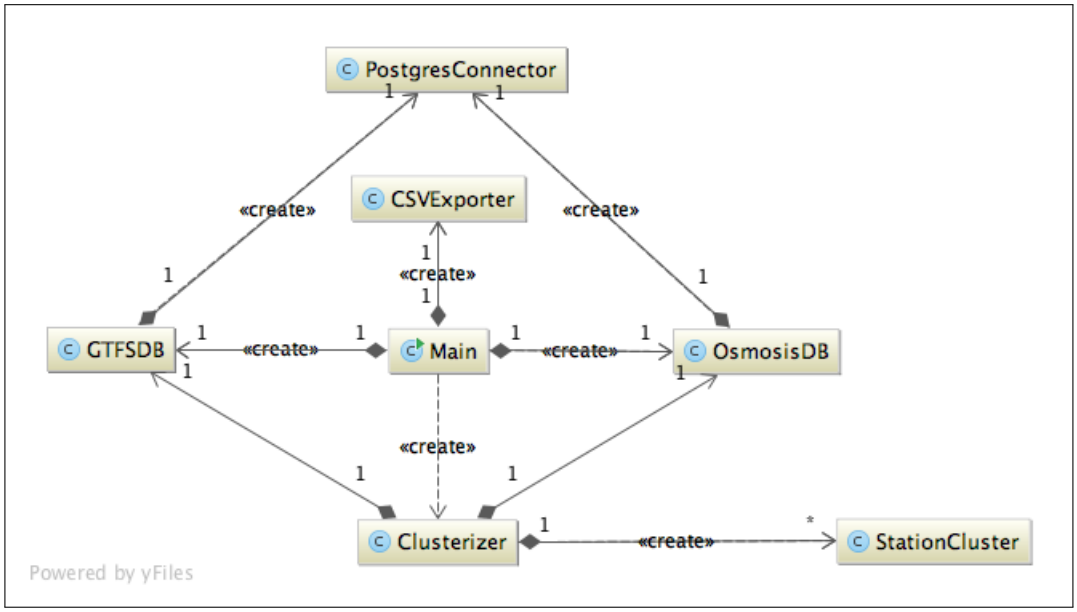

Abb. 10.: Auszug aus dem Klassendiagramm GTFS2OSMRailwayLinker Fokus auf Main

gut zur Extraktion administrativer und politischer Grenzen, da der Hauptzweck der importierten Datenbank das Rendern von Karten ist. Hier wurde innerhalb von osm2pgsql viel Wert auf saubere Erzeugung von Grenzen gelegt. Da ein Import der kompletten

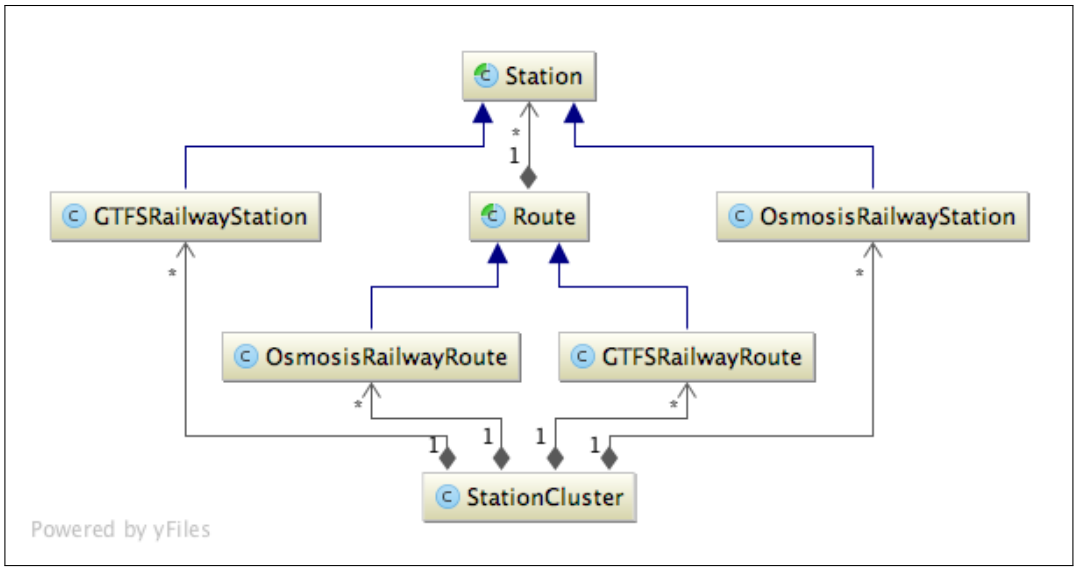

Abb. 11.: Auszug aus dem Klassendiagramm GTFS2OSMRailwayLinker Fokus auf StationCluster

3 Fazit

3.1 Praktikum und Studium

Die, im Studiengang Medieninformatik-Bachelor erworbenen, Grundlagen und Fähigkeiten ermöglichten es mir, meine Aufgaben zu erfüllen. Besonderer Fokus lag hier auf praktischen Erfahrungen mit Java und Datenbanken, sowie grundlegenden Problemlösungsstrategien die es mir ermöglichten ein vorhandenes Problem zu abstrahieren und dadurch zu lösen. Ein Beispiel hierfür ist das Problem Gruppenbildung von Straßen als Graphenproblem zu verstehen, und dieses mit bestehenden Bibliotheken zu lösen.

Wünschenswerte weitere Inhalte im Studiengang Medieninformatik-Bachelor wären meiner Meinung nach Grundlagen zu professionellen Java-Projekten (wie zum Beispiel die Verwendung von *Maven*) und ein Abschnitt Statistik innerhalb der Mathematik-Module gewesen.

Während meines Praktikums beim DFKI wurde mir Zeit gegeben, mich mit den Grundlagen von Klassifikatoren zu befassen. Hier wurde mein Interesse für Text-Klassifikation und Machine Learning geweckt. Ich werde die hier erworbenen Kenntnisse in meiner Bachelorarbeit weiter vertiefen.

3.2 Bewertung des Praktikums

Das DFKI bietet eine angenehme, offene Atmosphäre. Als Praktikant fühlte ich mich nie in der Bedrängnis etwas einfach nur schnell abarbeiten zu müssen, sondern es wurde viel Wert auf Verständnis und dem Erlernen neuer Fähigkeiten gelegt. Ein Beispiel hierfür ist, dass mir die Zeit gegeben wurde, mich eine ganze Woche nur mit dem Thema Klassifikatoren zu befassen. Sofern ein Interesse für Natural Langugage Processing und/oder Machine Learning vorliegt, bietet einem das DFKI sehr viel Fachwissen, Anregungen und Unterstützung. Alle Kollegen waren auch jederzeit bereit, ihr Wissen zu teilen. Mein Gesamteindruck ist sehr positiv. Es war eine sehr interessante Zeit, in der ich sehr viel lernte, und viele neue Fachgebiete kennenlernte.

Anlagen

A

A.1 Well-Known-Binary Format (WKB)

Das *Well-Known-Binary* Format ist die binäre Repräsentation eines geometrischen Objekts des *Simple Feature Models*. Das Simple Feature Model ist eine Untermenge des ISO 19107 Standards, welcher die geometrischen Eigenschaften von Geoobjekten spezifiziert. Ausgehend von einer allgemeinen Oberklasse können geometrische Primitive, wie z.B. ein Punkt, oder komplexe geometrische Objekte, wie z.B. Flächen oder Sammlungen von Objekten, beschrieben werden. (vgl. [Bil10]:358ff.). Die verfügbaren Klassen werden in Abbildung 12 aufgezeigt.

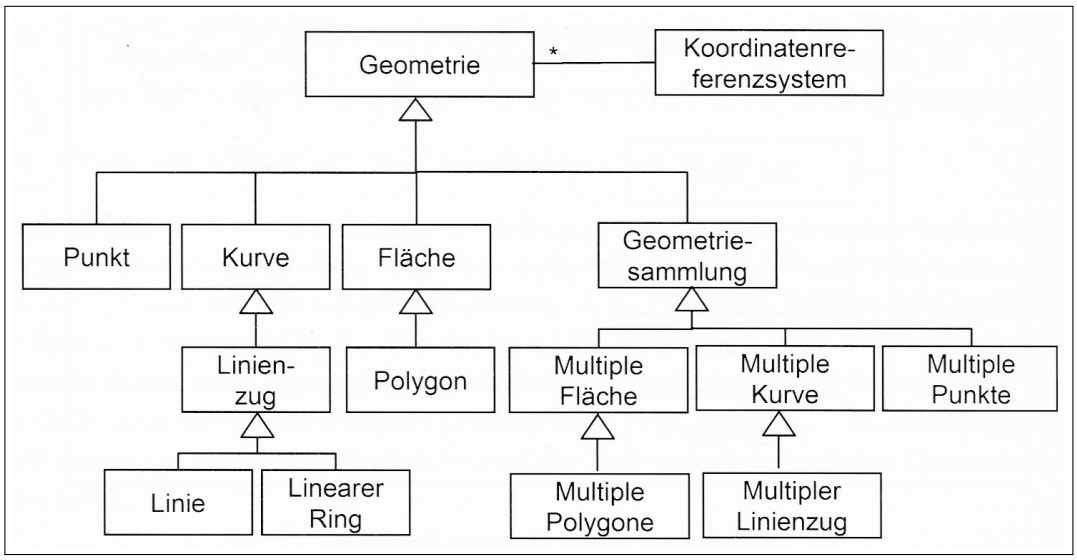

Abb. 12.: Geometrien im Simple Feature Model [Bil10]:360

Das WKB Format wird beispielsweise innerhalb der PostgreSQL Erweiterung PostGIS[1] genutzt, um geometrische Objekte in einer Datenbank abzulegen. Analog dazu existiert das *Well-Known-Text* Format, welches die textuelle Repräsentation geometrische Objekte des Simple Feature Models spezifiziert.

[1] http://postgis.net

Beispiel für das WKB und das WKT Format

Ein Punkt mit den Koordinaten `13.439561 Ost` sowie `52.54002 Nord` entspricht der WKT Repräsentation

```
SRID=4326;POINT(13.439561 52.54002)
```

und der WKB Repräsentation

```
0101000020E6100000B131AF230EE12A40CC9717601F454A40
```

Der Wert `SRID` beinhaltet die ID des zu verwendenden Koordinatenreferenzsystems. Die ID `4326` entspricht dem häufig verwendeten Referenzsystem `WGS84`[2].

A.2 GeoJSON

GeoJSON[3] ist ein Format zur Repräsentation von geometrischen Objekten im JSON Format. Es verwendet ein ähnliches hierarchisches Klassenmodell. (vgl. [@How08]).

Beispiel für das GeoJSON Format

Ein Punkt mit den Koordinaten `13.439561 Ost` sowie `52.54002 Nord` entspricht der GeoJSON Repräsentation

Listing A.1: Beispiel eines geometrischen Punktes in GeoJSON Repräsentation

```
  {
    "type": "Point",
    "coordinates": [
        13.439561,
        52.54002
    ]
}
```

A.3 OpenStreetMap Datenformat

OpenStreetMap bietet seine Daten zum Download in einer Datei, *planet.osm*[4], an. Diese Datei im XML-Format enthält den kompletten Datenbestand des OpenStreetMap

[2] http://spatialreference.org/ref/epsg/4326/
[3] http://geojson.org
[4] http://planet.openstreetmap.org/

Projekts. Da diese Datei sehr groß ist (gepackt ca. 50GB) bieten andere Dienstleister wie zum Beispiel die Geofabrik GmbH Karlsruhe[5] auch kleinere Bereiche des Datenbestandes, zum Beispiel nur Deutschland, an. Zur Speicherung der Daten werden die Elemente *Nodes*, *Ways* und *Relations* verwendet. (vgl. [@Ope15])

- **Nodes**, ein geometrischer Punkt, welcher durch geographische Breite und Länge bestimmt ist
- **Ways**, ein Linienzug zwischen mehreren Punkten. Hiermit werden zum Beispiel Straßen, Flüsse und vieles mehr modelliert
- **Relations**, eine logische Gruppierung mehrerer *Nodes*, *Ways* oder auch *Relations*. Die Mitglieder einer Relation haben einen Bezug zueinander, zum Beispiel ein Wald mit seinen Lichtungen oder ein Bahnhof.

Die Bedeutung der Elemente wird durch Key-Value Paare, sogenannte *Tags*, beschrieben. Hier wird beispielsweise festgelegt, ob ein Way eine Straße abbildet oder einen Fluss.

Beispiel eines Nodes

Listing A.2: Beispiel eines Nodes aus planet.osm

```
<node id="3637807236" visible="true" version="1" changeset="32456135"
     timestamp="2015-07-06T19:04:03Z" user="bigbug21" uid="15748" lat
    ="50.5379840" lon="12.1402231"/>
```

Beispiel eines Ways

Listing A.3: Beispiel eines Ways aus planet.osm

```
  <way id="358952758" visible="true" version="1" changeset="32456135"
       timestamp="2015-07-06T19:04:13Z" user="bigbug21"  uid="15748">
  <nd ref="3637807236"/>
  <nd ref="3637807232"/>
  <nd ref="3637807230"/>
  <nd ref="3637806843"/>
  <nd ref="3637806841"/>
  <tag k="public_transport" v="platform"/>
  <tag k="railway" v="platform"/>
  <tag k="train" v="yes"/>
 </way>
```

Beispiel einer Relation

Listing A.4: Beispiel einer Relation aus planet.osm

[5] http://www.geofabrik.de

```
<relation id="2303826" visible="true" version="9" changeset
    ="34783224" timestamp="2015-10-21T17:05:38Z" user="Kakaner" uid
    ="1851521">
  <member type="way" ref="166800600" role=""/>
  <member type="way" ref="166800590" role=""/>
  <member type="way" ref="166800593" role=""/>
  <member type="way" ref="166800357" role=""/>
  <member type="way" ref="376274301" role=""/>
  <member type="way" ref="166800198" role=""/>
  <member type="way" ref="165821978" role=""/>
  <member type="way" ref="165821752" role=""/>
  <member type="way" ref="165741930" role=""/>
  <member type="way" ref="165741583" role=""/>
  <member type="way" ref="165479804" role=""/>
  <member type="way" ref="165479800" role=""/>
  <member type="way" ref="165479712" role=""/>
  <member type="way" ref="165409360" role=""/>
  <member type="way" ref="165408947" role=""/>
  <member type="way" ref="264419428" role=""/>
  <member type="way" ref="165022595" role=""/>
  <member type="way" ref="165022069" role=""/>
  <member type="way" ref="164988309" role=""/>
  <member type="way" ref="164987948" role=""/>
  <member type="way" ref="159213270" role=""/>
  <member type="way" ref="264419420" role=""/>
  <member type="way" ref="376165609" role=""/>
  <member type="way" ref="158208266" role=""/>
  <member type="way" ref="159211943" role=""/>
  <member type="way" ref="264419427" role=""/>
  <member type="way" ref="356950454" role=""/>
  <member type="way" ref="158207942" role=""/>
  <member type="way" ref="158206932" role=""/>
  <member type="way" ref="250106659" role=""/>
  <member type="way" ref="250106648" role=""/>
  <member type="way" ref="254270906" role=""/>
  <member type="way" ref="158511899" role=""/>
  <member type="way" ref="158511802" role=""/>
  <member type="way" ref="159211993" role=""/>
  <tag k="operator" v="Vogtlandbahn"/>
  <tag k="public_transport:version" v="2"/>
  <tag k="ref" v="VB2"/>
  <tag k="route" v="train"/>
  <tag k="type" v="route"/>
 </relation>
```

A.4 Beispiel einer Straße in OpenStreetMap

Im folgenden Abschnitt soll am Beispiel eines Teilabschnittes einer Straße (siehe Abbildung 13) gezeigt werden, in welcher Form Straßen innerhalb der OpenStreet-

Map Daten vorliegen. Eine Straße ist entweder ein einzelner, oder eine Verkettung aus *Ways* mit gesetztem `highway` Tag. Im Listing A.5 Zeile 13 und 44 ist ein Beispiel dafür ersichtlich. Beispielsweise bedeutet der Wert `"primary"` Bundesstraße oder `"motorway"` Autobahn. Die Unterteilung einer Straße in mehrere Ways ist mitunter

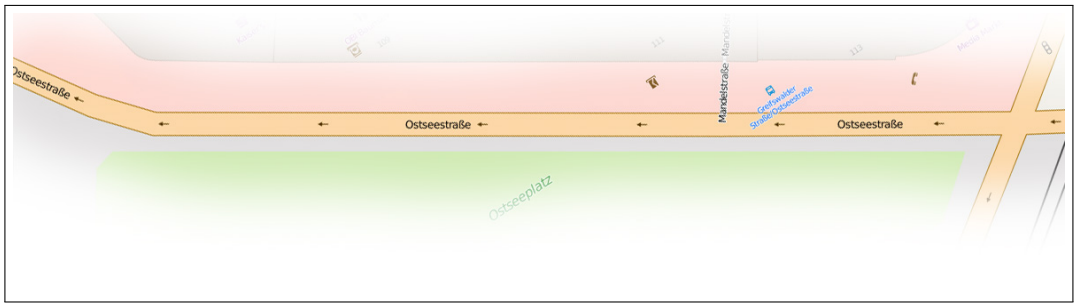

Abb. 13.: Teilausschnitt der Greifswalder Straße

notwendig, da an einem Way-Element alle Daten des aktuellen Straßenabschnitts, wie z.B. die zulässige Höchstgeschwindigkeit, durch *Tags* gespeichert sind. Sofern sich diese Daten im Straßenverlauf ändern, wird dies durch einen separates Way-Element wiedergegeben. Der hier gezeigte Teilabschnitt ist ebenfalls bereits in zwei separate Way-Elemente mit den ID´s 241186022 und 4615358 untergliedert (Abbildung 14). Dies war notwendig, da sich die Anzahl der Fahrspuren geändert hat. Der

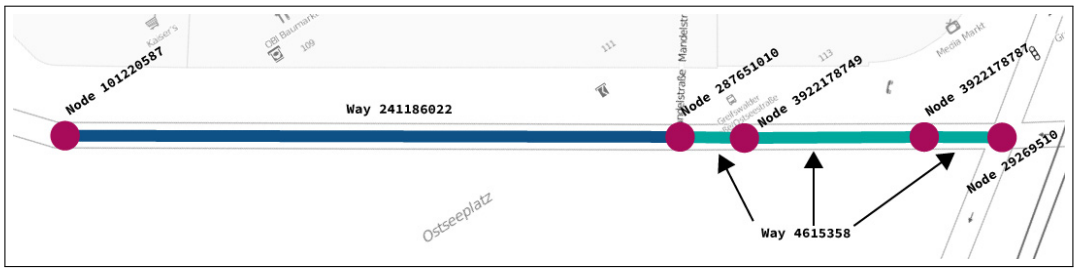

Abb. 14.: Beispiel für multiple Ways einer Straße

Tag mit dem Key “lanes” hat seinen Wert von 2 auf 3 geändert. Die Änderung ist im Listing A.5 auf Zeile 28 und 45 ersichtlich.

Listing A.5: Vollständige Daten des Straßenabschnitts

```
1 <node id="101220587" visible="true" version="11" changeset="18256244"
      timestamp="2013-10-08T22:00:56Z" user="Peter Maiwald" uid="90528"
      lat="52.5464853" lon="13.4424534"/>
2 <node id="287651010" visible="true" version="5" changeset="23075493"
     timestamp="2014-06-22T09:26:32Z" user="atpl_pilot" uid="881429"
     lat="52.5459138" lon="13.4438910"/>
3 <node id="3922178749" visible="true" version="1" changeset="36300926"
      timestamp="2016-01-01T16:28:14Z" user="Balgofil" uid="95702" lat
     ="52.5458508" lon="13.4440518">
4  <tag k="bus" v="yes"/>
5  <tag k="name" v="Greifswalder Straße/Ostseestraße"/>
6  <tag k="public_transport" v="stop_position"/>
```

```
 <tag k="ref:BVG" v="105416"/>
 <tag k="website" v="http://qr.bvg.de/h105416"/>
 <tag k="wheelchair" v="limited"/>
</node>
<node id="3922178787" visible="true" version="1" changeset="36300926"
     timestamp="2016-01-01T16:28:15Z" user="Balgofil" uid="95702" lat
    ="52.5456743" lon="13.4445025">
 <tag k="crossing" v="traffic_signals"/>
 <tag k="highway" v="crossing"/>
</node>
<node id="29269510" visible="true" version="11" changeset="36300926"
    timestamp="2016-01-01T16:28:39Z" user="Balgofil" uid="95702" lat
    ="52.5456175" lon="13.4446475">
 <tag k="TMC:cid_58:tabcd_1:Class" v="Point"/>
 <tag k="TMC:cid_58:tabcd_1:Direction" v="negative"/>
 <tag k="TMC:cid_58:tabcd_1:LCLversion" v="9.00"/>
 <tag k="TMC:cid_58:tabcd_1:LocationCode" v="21554"/>
 <tag k="TMC:cid_58:tabcd_1:NextLocationCode" v="21555"/>
 <tag k="TMC:cid_58:tabcd_1:PrevLocationCode" v="21553"/>
</node>
<way id="241186022" visible="true" version="4" changeset="35323067"
    timestamp="2015-11-15T08:45:00Z" user="anbr" uid="43566">
 <nd ref="287651010"/>
 <nd ref="101220587"/>
 <tag k="cycleway" v="lane"/>
 <tag k="highway" v="primary"/>
 <tag k="lanes" v="3"/>
 <tag k="maxspeed" v="50"/>
 <tag k="name" v="Ostseestraße"/>
 <tag k="oneway" v="yes"/>
 <tag k="postal_code" v="10409"/>
 <tag k="ref" v="L 1004"/>
 <tag k="sidewalk" v="right"/>
 <tag k="turn:lanes" v="left|none|none"/>
 <tag k="wikipedia" v="de:Ostseestraße"/>
</way>
<way id="4615358" visible="true" version="27" changeset="36300926"
    timestamp="2016-01-01T16:28:33Z" user="Balgofil" uid="95702">
 <nd ref="29269510"/>
 <nd ref="3922178787"/>
 <nd ref="3922178749"/>
 <nd ref="287651010"/>
 <tag k="cycleway" v="lane"/>
 <tag k="highway" v="primary"/>
 <tag k="lanes" v="2"/>
 <tag k="maxspeed" v="50"/>
 <tag k="name" v="Ostseestraße"/>
 <tag k="oneway" v="yes"/>
 <tag k="postal_code" v="10409"/>
 <tag k="ref" v="L 1004"/>
 <tag k="sidewalk" v="right"/>
```

```
 <tag k="wikipedia" v="de:Ostseestraße"/>
</way>
```

A.5 Beispiel zur Verwendung von JGraphT zur Trennung eines Graphen in zusammenhängende Elemente

Listing A.6: Beispiel der Trennung eines Graphen in verbundene Teile

```
import org.jgrapht.UndirectedGraph;
import org.jgrapht.alg.ConnectivityInspector;
import org.jgrapht.graph.DefaultEdge;
import org.jgrapht.graph.SimpleGraph;

// ...

// Lege neuen leeren Graphen an, Knoten sind Objekte vom Typ Long
UndirectedGraph<Long, DefaultEdge> graph = new SimpleGraph<>(
    DefaultEdge.class);
// Graph mit 4 Knoten und 2 Kanten befüllen
Long v1 = new Long(1);
Long v2 = new Long(2);
Long v3 = new Long(3);
Long v4 = new Long(4);
graph.addVertex(v1);
graph.addVertex(v2);
graph.addVertex(v3);
graph.addVertex(v4);
graph.addEdge(v1, v2);
graph.addEge(v3, v4);
// Erzeuge neuen ConnectivityInspector
ConnectivityInspector<Long, DefaultEdge> ci = new
    ConnectivityInspector<>(graph);
// Erzeuge eine Liste von Sets mit zusammenhängenden Knoten
// Hier werden jetzt zwei Sets erzeugt. (1,2) und (3,4)
List<Set<Long>> connectedNodes = ci.connectedSets();
```

A.6 GraphSeparator.java

Listing A.7: GraphSeparator.java

```
package de.dfki.OsmosisStreetExtractor.util.geo;

import de.dfki.OsmosisStreetExtractor.util.database.OsmosisDB;
import de.dfki.OsmosisStreetExtractor.util.database.StreetGroup;
import org.jgrapht.UndirectedGraph;
import org.jgrapht.alg.ConnectivityInspector;
```

```java
import org.jgrapht.graph.DefaultEdge;
import org.jgrapht.graph.SimpleGraph;

import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Set;

/**
* Created by Tom Oberhauser
* This class is used to separate a whole wayset (i.e. streets) into separate parts
*/
public class GraphSeparator {

  private final ArrayList<ArrayList<Long>> clusters_by_name;
  private final ArrayList<ArrayList<Long>> clusters_by_ref;
  private final ArrayList<ArrayList<Long>> clusters_by_intref;

  public GraphSeparator(StreetGroup streetGroup, OsmosisDB db) {
    /*
    * graphs for all ways
    */
    UndirectedGraph<Long, DefaultEdge> wayGraphStreetName = new SimpleGraph<>(DefaultEdge.class);
    UndirectedGraph<Long, DefaultEdge> wayGraphStreetRef = new SimpleGraph<>(DefaultEdge.class);
    UndirectedGraph<Long, DefaultEdge> wayGraphStreetIntRef = new SimpleGraph<>(DefaultEdge.class);

    /*
    * K: nodeId, V: WayIds (1 to n)
    */
    HashMap<Long, ArrayList<Long>> name_nodeId_2_wayIds = new HashMap<>();
    HashMap<Long, ArrayList<Long>> ref_nodeId_2_wayIds = new HashMap<>();
    HashMap<Long, ArrayList<Long>> intref_nodeId_2_wayIds = new HashMap<>();

    /*
    * Populate graph for name
    */
    if (streetGroup.hasName()) {
      populateGraph(wayGraphStreetName, name_nodeId_2_wayIds, db, db.getWaysForStreetName(streetGroup.getName()));
    }
    db.removeStreetName(streetGroup.getName()); //street name has been parsed, remove out of database.

    /*
```

```
49      * Populate graph for ref
50      */
51      if (streetGroup.hasRef()) {
52        populateGraph(wayGraphStreetRef, ref_nodeId_2_wayIds, db, db.
              getWaysForStreetRef(streetGroup.getRef()));
53      }
54      db.removeStreetRef(streetGroup.getRef()); //street ref has been
            parsed, remove out of database.
55
56      /*
57      * Populate graph for int_ref
58      */
59      if (streetGroup.hasIntRef()) {
60        populateGraph(wayGraphStreetIntRef, intref_nodeId_2_wayIds, db,
              db.getWaysForStreetIntRef(streetGroup.getIntRef()));
61      }
62      db.removeStreetIntRef(streetGroup.getIntRef()); //street ref has
            been parsed, remove out of database.
63
64      /*
65      * Split graph into parts and generate a List with Sets of NodeIds
66      */
67      ConnectivityInspector<Long, DefaultEdge> ci_streetName = new
            ConnectivityInspector<>(wayGraphStreetName);
68      List<Set<Long>> connectedNodesByName = ci_streetName.
            connectedSets();  //separate whole graph into isolated parts
69
70      ConnectivityInspector<Long, DefaultEdge> ci_streetRef = new
            ConnectivityInspector<>(wayGraphStreetRef);
71      List<Set<Long>> connectedNodesByRef = ci_streetRef.connectedSets
            ();  //separate whole graph into isolated parts
72
73      ConnectivityInspector<Long, DefaultEdge> ci_streetIntRef = new
            ConnectivityInspector<>(wayGraphStreetIntRef);
74      List<Set<Long>> connectedNodesByIntRef = ci_streetIntRef.
            connectedSets();  //separate whole graph into isolated parts
75
76      clusters_by_name = parseGraphClusters(connectedNodesByName,
            name_nodeId_2_wayIds);
77      clusters_by_ref = parseGraphClusters(connectedNodesByRef,
            ref_nodeId_2_wayIds);
78      clusters_by_intref = parseGraphClusters(connectedNodesByIntRef,
            intref_nodeId_2_wayIds);
79  }
80
81  /**
82  * Takes a list of sets of connected nodeIds and generates lists of
        lists of connected wayIds
83  *
84  * @param connectedSets      list of sets of connected nodes
85  * @param node2wayDictionary nodeId -> wayId lookup dictionary
```

```
86   * @return A list of lists which contain all wayIds for one
         connected set / cluster
87   */
88   private ArrayList<ArrayList<Long>> parseGraphClusters(List<Set<Long
         >> connectedSets, HashMap<Long, ArrayList<Long>>
         node2wayDictionary) {
89     ArrayList<ArrayList<Long>> retVal = new ArrayList<>();
90     for (Set<Long> aggregate : connectedSets) {
91       ArrayList<Long> currentAggregateWays = new ArrayList<>();
92       for (Long nodeId : aggregate) {
93         for (Long wayId : node2wayDictionary.get(nodeId)) {
94           currentAggregateWays.add(wayId);
95         }
96       }
97       retVal.add(currentAggregateWays);
98     }
99     return retVal;
100  }
101
102  /**
103  * Takes a list of wayIds and populates the graph and the lookup
         HashMaps
104  *
105  * @param graph          Graph to populate
106  * @param nodeDictionary lookup HashMap to populate (nodeIds ->
         wayIds)
107  * @param db             OsmosisDB object for querying the nodes of
         ways
108  * @param ways           list of way ids
109  */
110  private void populateGraph(UndirectedGraph<Long, DefaultEdge> graph
         , HashMap<Long, ArrayList<Long>> nodeDictionary, OsmosisDB db,
         ArrayList<Long> ways) {
111    if (ways != null) { //maybe name got parsed already
112      for (Long wayId : ways) {
113        ArrayList<Long> nodesArray = db.getNodeIds(wayId); //Get
             array of nodes for current way
114        graph.addVertex(nodesArray.get(0)); //add first vertex
115        nodeDictionary.putIfAbsent(nodesArray.get(0), new ArrayList
             <>()); //generate way dictionary for node if there is none
116        nodeDictionary.get(nodesArray.get(0)).add(wayId); //add wayId
              for node
117        for (int i = 1; i < nodesArray.size(); i++) {
118          Long currentNode = nodesArray.get(i);
119          Long lastNode = nodesArray.get(i - 1);
120          graph.addVertex(currentNode);
121          if (!currentNode.equals(lastNode)) { //loop detection
122            graph.addEdge(lastNode, currentNode);
123          }
124          nodeDictionary.putIfAbsent(currentNode, new ArrayList<>());
               //generate way dictionary for node if there is none
```

```
125             nodeDictionary.get(currentNode).add(wayId); //add wayId for
                    node
126           }
127         }
128       }
129     }
130
131     /**
132     * Returns all clusters by name
133     *
134     * @return list of lists of way ids
135     */
136     public ArrayList<ArrayList<Long>> getClustersByName() {
137       return clusters_by_name;
138     }
139
140     /**
141     * Returns all clusters by ref
142     *
143     * @return list of lists of way ids
144     */
145     public ArrayList<ArrayList<Long>> getClustersByRef() {
146       return clusters_by_ref;
147     }
148
149     /**
150     * Returns all clusters by int_ref
151     *
152     * @return list of lists of way ids
153     */
154     public ArrayList<ArrayList<Long>> getClustersByIntref() {
155       return clusters_by_intref;
156     }
157 }
```

A.7 Beispielausgabe der Straßenliste des OsmosisStreetExtractor

Es folgt eine Beispielausgabe des `OsmosisStreetExtractor`. Der Inhalt des Feldes `linestring` wurde aus Gründen der Lesbarkeit gekürzt.

A.8 General Transit Feed Specification (GTFS)

Hier findet sich, auszugsweise, eine kurze Übersicht des *General Transit Feed Specification (GTFS)* Formates. GTFS Daten bestehen aus mehreren gepackten Textdateien.

Eine Übersicht über die benötigten Dateien innerhalb eines GTFS Pakets und deren Inhalt ist in Tabelle A.1 und Abbildung 15 dargestellt. (vgl. [@Goo16])

Tab. A.1.: Benötigte Dateien innerhalb eines GTFS-Pakets

Datei	Inhalt
agency.txt	Verkehrsunternehmen
stops.txt	Bahnhöfe, Haltestellen
routes.txt	Linien, z.B. eine S-Bahn Strecke
trips.txt	eine konkrete Fahrt auf einer Linie
stop_times.txt	Abfahrtszeiten pro Fahrt und Bahnhof
calendar.txt	Tage, an denen ein Dienst verfügbar ist

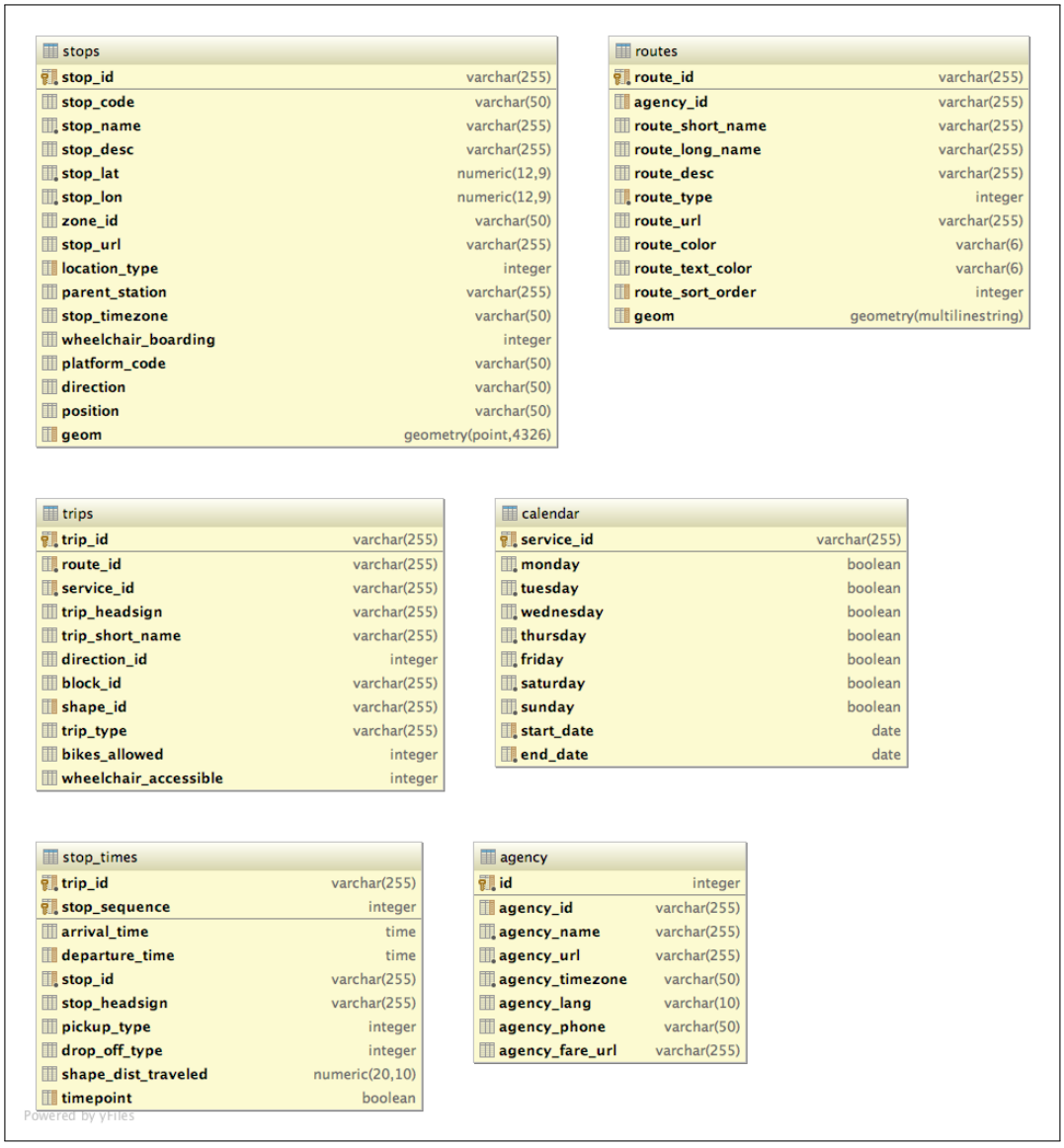

Abb. 15.: Felder innerhalb von GTFS Daten

A.9 Levenshtein Distanz zur Ermittlung der Ähnlichkeit zweier Bahnhofsnamen

Die *Levenshtein Distanz* zwischen zwei Zeichenketten gibt die Anzahl der Editieroperationen zurück, die notwendig sind um die eine Zeichenkette in die andere zu transformieren. Der Größe des ermittelten Wertes kann, je nach Länge der Zeichenketten variieren. Die verwendete Bibliothek *java-string-similarity*[6] bietet die Möglichkeit, eine normalisierte Levenshtein-Distanz zu ermitteln. Dazu wird die ermittelte Levenshtein-Distanz anschließend durch die Länge der längsten Zeichenkette dividiert. Diese Operation ergibt ein Interval $[0, 1]$.

Nun musste ein Algorithmus entwickelt werden, der die normalisierte Levenshtein-Distanz ein bisschen besser auf das Themengebiet Bahnhofsbezeichnungen vorbereitet. Ein Beispiel hierfür sind die Zeichenketten `"Hauptbahnhof Berlin"` und `"Berlin Hauptbahnhof"`. Offensichtlich bezeichnen diese den selben Bahnhof, jedoch entspricht die Levenshtein-Distanz zwischen den Zeichenketten 14 und die normalisierte Levenshtein-Distanz $14/19 \approx 0,737$.

Ich entschied mich dazu, die Zeichenketten vor dem Vergleich zu *tokenizen*, das bedeutet die Zeichenkette anhand bestimmter Trennungszeichen in eine Liste aus Teilzeichenketten aufzuteilen. Dazu wurde die Klasse `java.util.StringTokenizer` verwendet (Siehe Listing A.8 Zeile 74-81). Anschließend werden die Tokens der beiden zu vergleichenden Zeichenketten alle miteinander verglichen und die jeweils niedrigsten Levenshtein-Distanzen, gewichtet anhand der Länge des kürzeren Tokens aufsummiert (Siehe Listing A.8). Dies sorgt zum Beispiel dafür, dass die zu Beginn erwähnten Zeichenketten `"Hauptbahnhof Berlin"` und `"Berlin Hauptbahnhof"` eine Distanz von 0 aufweisen.

Listing A.8: Klasse StringDistance zur Ermittlung der Gleichheit zweier Bahnhofsnamen

```
1  package utils;
2
3  import com.google.common.collect.HashBasedTable;
4  import com.google.common.collect.Table;
5  import info.debatty.java.stringsimilarity.NormalizedLevenshtein;
6  import info.debatty.java.stringsimilarity.interfaces.
       NormalizedStringDistance;
7
8  import java.util.ArrayList;
9  import java.util.List;
10 import java.util.StringTokenizer;
11
12 /**
13  * Created by Tom Oberhauser
```

[6] https://github.com/tdebatty/java-string-similarity

```
14  * Methods for String comparison
15  */
16 public class StringDistance {
17   /**
18    * Compares two Strings. It returns a weighted sum of the best
         matching tokens.
19    * (eg. "Berlin Hauptbahnhof" and "(Hauptbahnhof) Berlin" would
         have a distance of 0)
20    *
21    * @param s1 left String
22    * @param s2 right String
23    * @return likelihood [0;1] where 0 is equal and 1 is different
24    */
25   public static double minDistance(final String s1, final String s2)
        {
26     double retVal = 0;
27     int n = 0;
28     List<String> words_left = tokenize(s1);
29     List<String> words_right = tokenize(s2);
30     Table<String, String, Double> matcherTable = HashBasedTable.
         create();
31     /*
32     compare every left with every right word and fill matcher tables
33      */
34     for (String wordLeft : words_left) {
35       for (String wordRight : words_right) {
36         double lev = levenshtein(wordLeft, wordRight);
37         matcherTable.put(wordLeft, wordRight, lev);
38       }
39     }
40     /*
41     find smallest values and remove corresponding rows and cols until
           the table is empty
42      */
43     while (!matcherTable.isEmpty()) {
44       String minRow = null;
45       String minCol = null;
46       Double minVal = Double.MAX_VALUE;
47       for (String row : matcherTable.rowKeySet()) {
48         for (String col : matcherTable.columnKeySet()) {
49           Double currentCell = matcherTable.get(row, col);
50           if (currentCell < minVal) {
51             minRow = row;
52             minCol = col;
53             minVal = currentCell;
54           }
55         }
56       }
57       if (minRow != null && minCol != null) {
58         int minLength = (minRow.length() <= minCol.length()) ? minRow
             .length() : minCol.length();
```

```
59          n += minLength;
60          retVal += (minVal * (double) minLength);
61          matcherTable.row(minRow).clear();
62          matcherTable.column(minCol).clear();
63        }
64      }
65      return retVal / n;
66    }
67
68    /**
69     * Tokenizes a String
70     *
71     * @param s String
72     * @return List of tokens without delimiters
73     */
74    private static List<String> tokenize(final String s) {
75      List<String> list = new ArrayList<>();
76      StringTokenizer tokenizer = new StringTokenizer(s, "()[].-,;␣",
            false);
77      while (tokenizer.hasMoreTokens()) {
78        list.add(tokenizer.nextToken());
79      }
80      return list;
81    }
82
83    /**
84     * Calculates the normalized levenshtein distance of two Strings
85     *
86     * @param s1 left String
87     * @param s2 right String
88     * @return likelihood [0;1] where 0 is equal and 1 is different
89     */
90    private static double levenshtein(final String s1, final String s2)
          {
91      NormalizedStringDistance nlv = new NormalizedLevenshtein();
92      return nlv.distance(s1, s2);
93    }
94  }
```

A.10 Kennzahlen zur qualitativen Bewertung eines binären Klassifikators

Ein Klassifikator soll etwas bestimmtes aus gegebenen Merkmalen vorhersagen. Genauergesagt entspricht ein Klassifikator einer Abbildung eines Merkmalsraumes auf eine Menge von Klassen. Um Die Qualität dieser Vorhersage bzw. Abbildung zu bewerten, existieren Kennzahlen. Im folgenden wird auf die Kennzahlen *Precision*

Tab. A.2.: Wahrheitsmatrix

	Vorhersage +	**Vorhersage -**
Wahrheit +	True Positive (TP)	False Negative (FN)
Wahrheit -	False Positive (FP)	True Negative (TN)

(Genauigkeit), *Recall (Trefferquote)* und *F-Maß* zur qualitativen Bewertung eines binären Klassifikators eingegangen.

Zunächst muss ein Datensatz mit bekannten Ergebnissen vorliegen. Dieser wird auch Gold-Standard genannt. Anhand des Gold-Standards können nun Vorhersagen mit bekannten Ergebnissen durchgeführt, und eine *Wahrheitsmatrix* befüllt werden. In Tabelle A.2 ist eine Wahrheitsmatrix für die Beispielklassen + und – abgebildet.

Anhand der Messwerte TP, FP, FN, TN lassen sich nun Kennzahlen errechnen.

Precision (Genauigkeit)

Der *Precision-Wert* gibt das Verhältnis zwischen allen positiven Vorhersagen und den korrekten positiven Vorhersagen an. Vereinfacht gesagt, wie viele von den positiv vorhergesagten Ergebnissen waren korrekt.

$$Precision = \frac{TP}{TP + FP}$$

Recall (Trefferquote)

Der *Recall-Wert* gibt das Verhältnis zwischen allen korrekt als positiv vorhergesagten Objekte an der Gesamtheit der tatsächlich positiven Objekte an. Vereinfacht gesagt, wie viele positive Ergebnisse wurden wirklich gefunden.

$$Recall = \frac{TP}{TP + FN}$$

F-Maß

Die Kennzahlen *Precision* und *Recall* alleine ergeben noch keine gute Abbildung der Vorhersagequalität. Beispielsweise lässt sich der Precision-Wert anheben, in dem der Klassifikator nur ein einzelnes Ergebnis als + korrekt klassifiziert. Alle als + vorhergesagten Werte währen dann in diesem Fall korrekt. Aus diesem Grund gibt

es das *F-Maß*, welche beide Werte miteinander vereint und als Qualitätskriterium verwendet werden kann.

$$F = 2 * \frac{Precision * Recall}{Precision + Recall}$$

A.11 Beispielausgabe des GTFS2OSMRailwayLinker

Der Inhalt des Feldes `boundingBox` wurde aus Gründen der Lesbarkeit gekürzt.

Listing A.9: Ausgabe des GTFS2OSMRailwayLinker

```
1 id,name,center,boundingBox,osm_node_ids,gtfs_stop_ids
2 1456,"Berlin Greifswalder Str",0000000001402
    AE0E29F9CE8DC404A45231B229B35,000000000300...06C0E47
    ,{3659830181,3876863066,1127967028},{8089011}
```

Literaturverzeichnis

[Bil10] Ralf Bill. *Grundlagen der Geo-Informationssysteme*. 5., völlig neu bearb. Aufl. Berlin [u.a.]: Wichmann, 2010 (zitiert auf Seite 19).

Online-Quellen

[@GIS12] GISpunkt HSR Wiki. *Osm2pgsql - GISpunkt HSR*. 2012. URL: http://giswiki.hsr.ch/Osm2pgsql (besucht am 24. Mai 2016) (zitiert auf Seite 6).

[@Goo16] Google. *General Transit Feed Specification Reference*. 2016. URL: https://developers.google.com/transit/gtfs/reference (besucht am 25. Mai 2016) (zitiert auf Seite 30).

[@How08] Howard Butler (Hobu Inc.), Martin Daly (Cadcorp), Allan Doyle (MIT), Sean Gillies (UNC-Chapel Hill), Tim Schaub (OpenGeo), Christopher Schmidt (MetaCarta). *The GeoJSON Format Specification*. 2008. URL: http://geojson.org/geojson-spec.html (besucht am 23. Mai 2016) (zitiert auf Seite 20).

[@Ing16] Ingo Schwarzer. *Smart Data For Mobility (SD4M) – Projekt-Präsentation*. 2016. URL: http://www.sd4m.net/sites/default/files/publications/%20SD4M-Pr%C3%A4sentation.pdf (besucht am 10. Mai 2016) (zitiert auf Seite 4).

[@Ope15] OpenStreetMap Wiki. *DE:Elemente - OpenStreetMap Wiki*. 2015. URL: http://wiki.openstreetmap.org/wiki/DE:Elemente (besucht am 24. Mai 2016) (zitiert auf Seite 21).

Declaration

You can put your declaration here, to declare that you have completed your work solely and only with the help of the references you mentioned.

Berlin, 03.06.2016

Tom Oberhauser