Lucas Bärenwald
6177115
Informatik (BA)
lucas@baerenwald.de

Bachelorarbeit

Automatische Visualisierung von Gebäuden auf der Basis von sprachlichem Input

Lucas Bärenwald

Abgabedatum: 26.09.2019

Institut für Informatik
Text Technology Lab
Prof. Dr. Alexander Mehler

Erklärung

Hiermit bestätige ich, dass ich die vorliegende Arbeit selbstständig verfasst habe und keine anderen Quellen oder Hilfsmittel als die in dieser Arbeit angegebenen verwendet habe.

_____

Ort, Datum

_____

Unterschrift

Abstract

Das Ziel dieser Arbeit ist, einen Text automatisch darauf zu untersuchen, ob er Gebäude beschreibt, und diese gegebenenfalls zu visualisieren. Zu diesem Zweck wurde ein Prototyp entwickelt, der mithilfe von NLP-Software auf Basis einer UIMA-Pipeline einen Text auf Gebäudedaten untersucht und diese anschließend als 3D-Modelle auf einer Karte visualisiert. Um die Güte des Projekts zu bestimmen wurde eine Evaluation durchgeführt, in der die Aufgabe darin bestand, Paragraphen ihren zugehörigen 3D-Modellen zuzuordnen. Die Ergebnisse wiesen eine Erkennungsrate von 88.67% auf. Jedoch wurden auch Schwächen im Standardisierungsverfahren der Parameter und in der einseitigen Art zu Visualisieren aufgezeigt. Zum Schluss wird vorgestellt, wie diese Schwachstellen mithilfe eines ontologischen Modells behoben werden können und wie mit dem Projekt weiterverfahren werden kann.

Danksagung

An dieser Stelle möchte ich mich bei all denjenigen bedanken, die mich während der Anfertigung dieser Bachelorarbeit unterstützt und motiviert haben. Mein besonderer Dank gilt Herrn Giuseppe Abrami und Herrn Prof. Dr. Alexander Mehler von der Goethe Universität Frankfurt für ihre Betreuung und Unterstützung. Ebenfalls bedanke ich mich bei allen, die an der Evaluation meines Projekts teilgenommen haben. Außerdem möchte ich mich bei Ina Bärenwald und Pedro Alves Zipf für das Korrekturlesen bedanken. Zuletzt danke ich Delfine Zumbusch und meiner restlichen Familie sowie meinen Freunden und Kommilitonen, die immer ein offenes Ohr, aufmunternde Worte, einen guten Rat oder ein angenehmes Schweigen für mich übrig hatten.

Inhaltsverzeichnis

Glossar

AMOD Dependency, die eine Modifikation durch ein Adjektiv beschreibt [31]. 23

Annotation Metadaten, die von Annotatoren an das zu analysierende Dokument angehängt werden. 11–14, 16, 19, 20, 22, 24–27, 33

Annotator NLP Werkzeuge, die spezifische Merkmale eines Textes erkennen, verarbeiten und als Metadaten an das Dokument anhängen. 9–20, 22–27, 30, 33, 34

Carto Frei zugänglicher Dienst, der das Abspeichern und Anzeigen von geografischen Daten ermöglicht. [5]. 15, 17, 18

CONJ Dependency, die eine Verbindung zweier Wörter darstellt, die mit „und“ oder „oder“ verknüpft werden [31]. 23

Dependency Abhängigkeit zwischen zwei Wörtern. Gemeinsam beschreiben sie die grammatikalische Struktur eines Satzes [8]. 13, 20, 22–24

GeoJSON Ein besonderes Format in JSON, das dazu entwickelt wurde um verschiedene geografische Daten zu repräsentieren[3]. 9, 15, 18, 25–27, 30

JSON JavaScript Object Notation ist die Sprache in der JavaScript Objekte formuliert werden. Sie ist zum Datenaustausch entwickelt worden, da sie gut von Menschen und Maschinen lesbar ist[24]. 26

Lemma Die Grundform oder auch Nennform eines Wortes. 12, 14, 20, 22–24, 26

Mapbox Anbieter von Online-Karten [21]. 15

Named Entity Ein Eigenname einer Sache, wie z.B. Personen oder Firmennamen [22]. 13, 20, 26

NLP Natural Language Processing „beschreibt Techniken und Methoden zur maschinellen Verarbeitung natürlicher Sprache“ [23]. 10, 12, 16, 35

NSUBJ Dependency, die, ausgehend vom Subjekt, die syntaktische Hauptaussage des Satzes beschreibt [31]. 23

Overpass API API des OpenStreetMap Dienstes, der die Abfrage von geografischen Daten ermöglicht [28]. 15, 26

POS Part-of-Speech beschreibt die Wortart eines Wortes. 13, 20, 24

SPARQL SPARQL Protocol and RDF Query Language ist eine graphenbasierte Abfragesprache für das Resource Description Framework (Datenmodell, welches auf gerichteten Graphen basiert). 14, 24, 26

Token Die kleinste Einheit eines Dokuments. Im Falle von Texten beschreiben Token die einzelnen Wörter [6]. 12–16, 20, 22–26

1. Einleitung

1.1. Motivation

Im Feld der Computerlinguistik beschäftigen sich Menschen schon seit den 1950er Jahren mit der maschinellen Verarbeitung von natürlichen Sprachen [4]. Dabei besteht ein großer Teil der Forschung aus dem „Natural Language Processing“. Es behandelt die Analyse von natürlicher Sprache, wobei nicht nur der Inhalt von Texten, sondern auch deren Struktur Forschungsgegenstand sind [23].

Im Falle eines Textes, der den Aufbau einer Stadt, die Häuser einer Straße oder auch nur ein einzelnes Bauwerk beschreibt, bestünde das Resultat einer Analyse, neben den Eigenschaften (Höhe, Position, Form, etc.) eines oder mehrerer Gebäude, aus Daten zur Satzstruktur, Eigenschaften verschiedener Wörter und weiterer grammatikalischer Besonderheiten. Diese repräsentieren zwar den Inhalt und den Aufbau des Textes, eine genaue Vorstellung der Bauwerke bekommen allerdings nur diejenigen, die die nötigen fachlichen Kenntnisse besitzen, um aus den Informationen ein Modell zu erstellen.

Eine Software-Lösung, die Texte auf Kenndaten von Bauwerken untersucht und diese anschließend automatisch in einer passenden Umgebung anzeigt, würde daher nicht nur fachkundigen Menschen helfen, ein Modell aufgrund eines Textes zu erstellen, sondern auch Laien die Möglichkeit bieten, einen Text zu schreiben und sich das Resultat automatisch generieren zu lassen.

Das in dieser Arbeit vorgestellte Text-To-Building-Projekt wurde zu diesem Zweck entwickelt. Es untersucht einen übergebenen Text automatisch auf Gebäudedaten und stellt aufgrund der ermittelten Informationen eine Visualisierung bereit.

1.2. Related Work

Die Projekte OSM-3D [14], OSM2World [26], ViziCities [32], OpenScienceMap [25] und OSMBuildings (siehe Abschnitt 2.2.2) sind alle Open-Source Dienste, die Web-basierend Gebäude in 3D auf einer Karte visualisieren. Hierfür werden die geografischen Daten des OpenStreetMap-Dienstes (siehe Abschnitt 2.2.1) verwendet. Da schon vorgefertigte Gebäudedaten genutzt werden, bieten diese sich nicht für die Nutzung im Text-To-Building-Projekt an. ViziCities und OSMBuildings haben jedoch jeweils eine JavaScript API mit der man die Dienste auch mit eigenen GeoJSON-Daten benutzen kann. Da das Frontend des Text-To-Building-Projekts nicht nur Gebäude, sondern auch den zugehörigen Text anzeigen soll, wurde eine eigene Implementierung auf Basis der OSMBuildings Bibliothek programmiert.

Im Bereich der Sprachanalyse wurde kein Projekt gefunden, welches spezifisch Gebäudedaten annotiert, aufgrund dessen der Text-To-Building-Annotator im Rahmen dieser Arbeit entwickelt wurde.

1.3. Aufgabenstellung

Das Ziel dieser Arbeit ist, eine Software zu entwickeln, die einen gegebenen Text auf Gebäudemerkmale untersucht und diese automatisch in Form von 3D-Modellen auf einer Karte anzeigt. Dies ermöglicht Leuten, die keine Programmier- oder NLP-bezogene Fachkenntnis besitzen, anhand eines Textes ein Modell eines Bauwerkes zu erzeugen und darzustellen.

1.4. Struktur der Arbeit

Die Arbeit besteht aus sechs Kapiteln: Das erste Kapitel enthält eine Vorstellung der Thematik und eine Einordnung ins Fachgebiet. Im zweiten Kapitel werden alle technologischen Voraussetzungen vorgestellt und Begrifflichkeiten erklärt, auf denen die Umsetzung und mögliche Anwendung bzw. Erweiterung des Text-To-Building-Projekts aufbaut. Mit dem dritten Kapitel beginnt die eigentliche Bearbeitung der Aufgabenstellung. Es beinhaltet Informationen dazu, wie die Text-To-Building-Software funktioniert. Unterteilt ist das Kapitel in zwei Abschnitte: Der erste Abschnitt zeigt, wie das Frontend mithilfe der OSMBuildings Bibliothek (siehe Abschnitt 2.2.2) und der GeoViz WebApp (siehe Abschnitt 2.2.3) umgesetzt wurde. Im zweiten Abschnitt wird das Backend vorgestellt, das hauptsächlich aus einer UIMA-Pipeline besteht, die aus weiterverwendeten und neu implementierten Annotatoren aufgebaut ist. Das vierte Kapitel behandelt die Evaluation des Projekts und zeigt anhand ihrer Ergebnisse, wie das Text-To-Building-Projekt das Ziel der Arbeit erreicht und welche Schwachstellen es aufweist. Im fünften Kapitel werden anhand der aufgezeigten Schwachstellen mögliche Verbesserungen mithilfe eines ontologischen Modells präsentiert und es werden Anwendungsbeispiele, wie der TextImager, genannt. Das sechste und letzte Kapitel fasst die Arbeit zusammen und zieht ein abschließendes Fazit.

2. Technologische Voraussetzungen

In diesem Kapitel werden alle benötigten Voraussetzungen und Projekte erläutert auf denen das Text-To-Building-Projekt aufbaut. Zuerst werden alle für die sprachliche Vorverarbeitung benötigten Komponenten und Dienste. Dann folgt eine Erläuterung aller Projekte, die die Basis der Visualisierung bilden. Der dritte Abschnitt beinhaltet Veranschaulichungen von Projekten und Konzepten, mit denen das Text-To-Building-Projekt verwendet oder weiterentwickelt werden kann.

2.1. Sprachliche Vorverarbeitung

2.1.1. Unstructured Information Management Architecture

Das Unstructured Information Management Architecture (UIMA) Framework ermöglicht es, Anwendungen, die unstrukturierte Daten (z.B. Texte, Audio-, Videodateien) verarbeiten, zu erstellen und miteinander zu verbinden [19]. Um die Informationen des übergebenen Textes auf relevante Daten zu untersuchen müssen sie zunächst in eine für die Verarbeitung geeignete Struktur überführt werden. Für diesen Zweck nutzt das Text-To-Building-Projekt folgende Komponenten aus der Apache UIMA Implementierung des UIMA Frameworks:

Common Analysis System

Das Common Analysis System (CAS) ist eine Datenstruktur, über die der Datenaustausch verschiedener Komponenten einer UIMA Anwendung erfolgt. Durch Indizierung wird einerseits der Zugriff auf bereits bestehende Annotationen, andererseits das Hinterlegen neuer Annotationen gewährleistet [15]. Dadurch können verschiedene Annotatoren gemeinsam auf dem selben Dokument arbeiten.

Typsystem

Jede UIMA Anwendung nutzt ein Typsystem. In diesem sind alle Datentypen aufgeführt, die während des Textanalyseprozesses genutzt werden. So können Daten, unabhängig von der Programmiersprache, modelliert und an andere Annotatoren angepasst werden [15]. Dies führt zu einem fehlerfreien Umgang mit den Annotationen aller Komponenten innerhalb einer UIMA Pipeline [34].

Pipeline

Mithilfe des CAS und des Typsystems einer UIMA Anwendung wird die Möglichkeit geschaffen, mehrere Annotatoren in einer Pipeline zu verbinden. Dadurch haben die Annotatoren Zugriff auf die vorherigen Annotationen und können diese für ihre Verarbeitung nutzen.

2.1.2. DKPro™ Core Framework

Das DKPro™ Core Framework ist eine Sammlung von NLP Software, welche auf dem Apache UIMA Framework aufbaut [13]. Einige Komponenten daraus ermöglichen die Vorverarbeitung des Textes, die für den Text-To-Building-Annotator notwendig ist.

LanguageToolSegmenter

Der LanguageToolSegmenter teilt den übergebenen Text in Sätze und Token ein, die für die Funktionalität der darauf folgenden Komponenten benötigt werden [12].

Abbildung 2.1.: Language Tool Segmenter

ParagraphSplitter

Der ParagraphSplitter unterteilt das übergebene Dokument in Paragraphen. Das geschieht durch einen regulären Ausdruck, durch den doppelte Zeilenumbrüche erkannt werden. Die Weboberfläche, die dazu dient die Visualisierung des Text-To-Building-Projekts zu evaluieren, benötigt die Paragraph Annotationen, um eine genaue Verbindung zwischen einem Gebäude und dem Paragraphen herzustellen, der es beschreibt [11].

Abbildung 2.2.: Paragraph Splitter

LanguageToolLemmatizer

Der LanguageToolLemmatizer ermittelt aufgrund von übergebenen Sätzen und Token die Lemmata der Token durch Abgleichen der Wörter mit einem Lexikon [10]. Diese werden im Text-To-Building-Annotator dazu benutzt, um bei der Suche nach einem Stichwort alle möglichen Formen des Wortes abzudecken, indem nicht mit den Token sondern mit den zugehörigen Lemmata verglichen wird.

Abbildung 2.3.: Language Tool Lemmatizer

StanfordPosTagger

Der StanfordPosTagger annotiert mithilfe der übergebenen Token und Sätze die zu den Token passenden POS-Tags [20]. Sie werden einerseits für folgende Annotatoren benötigt, andererseits helfen sie die Suche nach Stichworten einzugrenzen. So kann ein fälschlicherweise entdecktes Wort, falls es nicht der richtigen Wortart entspricht, wieder ausgeschlossen werden.

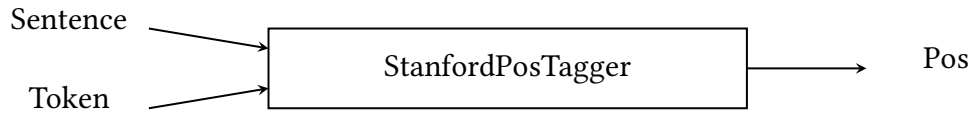

Abbildung 2.4.: Stanford Pos Tagger

StanfordNamedEntityRecognizer

Mithilfe der vorher annotierten Token und Sätze kann der StanfordNamedEntityRecognizer Named Entities erkennen [22]. Dies ist notwendig, falls in einem Text Gebäude vorkommen, die schon bekannt sind. Durch die Named Entity-Annotationen kann gezielt nach passenden geografischen Gebäudedaten gesucht werden.

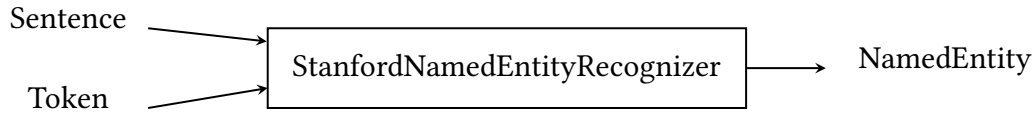

Abbildung 2.5.: Stanford Named Entity Recognizer

CoreNLPDependencyParser

Der CoreNLPDependencyParser nutzt die vorher erkannten Token, Sätze und POS-Tags, um die Dependencies jedes Satzes zu annotieren [9]. Aufgrund derer kann eine Verbindung zwischen verschiedenen Token hergestellt werden. So kann einem Token, welches ein Attribut beschreibt, ein Wert zugewiesen werden, oder einem Gebäudetoken seine Attributtoken.

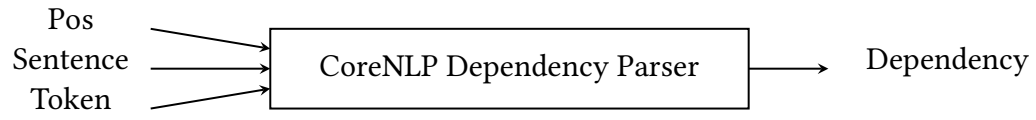

Abbildung 2.6.: CoreNLP Dependency Parser

2.1.3. CoReference-Annotator

Der CoReference-Annotator ist ein von Alexander Henlein mittels maschinellem Lernen entwickelter UIMA Annotator, der erkennt, ob sich ein Ausdruck am Anfang eines Satzes auf ein Wort aus einem vorherigen Satz bezieht [18]. Da der Dependency Parser aus dem DKPro™ Core Framework nur die grammatikalische Struktur innerhalb eines Satzes erkennt, ist der CoReference-Annotator essentiell für die Analyse von Beschreibungen, die sich über mehrere Sätze erstrecken. Durch seine Annotationen werden auch Verbindungen zwischen Token, die sich in verschiedenen Sätzen befinden, erkannt.

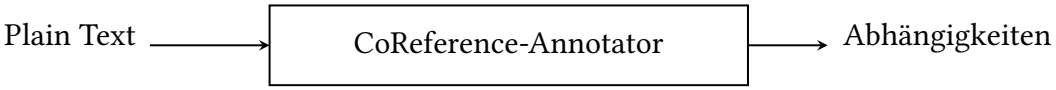

Abbildung 2.7.: CoReference-Annotator

2.1.4. Color-Annotator

Der von Giuseppe Abrami entwickelte Color-Annotator vergleicht die Lemmata des Textes mit einer Liste von Farbnamen aus der WikiData-Datenbank (siehe Abschnitt 2.1.5) und speichert die zugehörigen RGB-Werte für gefundene Wörter ab [1]. So lassen sich die Gebäude und deren Dächer in spezifischen Farben visualisieren, falls sie angegeben sind.

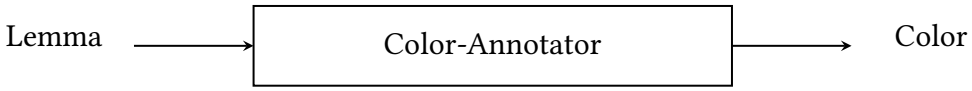

Abbildung 2.8.: Color-Annotator

2.1.5. WikiData

Die Wikidata ist eine frei zugängliche, kollaborative Wissensdatenbank, die die Daten aus anderen Wikimedia Projekten in strukturierter Form bereitstellt [33]. Sie stellt einen Service zur Abfrage der Daten bereit auf den per SPARQL Sprache zugegriffen werden kann [30]. Dadurch können Wortlisten erstellt werden, die viele Wörter einer bestimmten Gruppierung beinhalten. So kann z.B. die Suche nach Gebäuden in einem Text erheblich erweitert werden, wenn nicht nur nach wenigen, manuell beschriebenen Wörtern, sondern nach einer großen Anzahl von Wörtern aus der Wikidata gesucht wird.

2.1.6. Tiefen- und Breitensuche

Die Tiefen- und die Breitensuche sind Suchalgorithmen, die auf Graphen agieren. Die Tiefensuche iteriert dabei über Kanten, bis vom aktuellen Knoten kein Knoten besucht werden kann, der noch nicht besucht wurde. Anschließend setzt sich der Algorithmus vom vorherigen Knoten aus fort. Der Algorithmus bricht ab, wenn alle erreichbaren Knoten einmal besucht wurden. Somit werden per Tiefensuche zuerst die Knoten bearbeitet, die möglichst weit

entfernt vom Startknoten liegen. Die Breitensuche hingegen arbeitet nahe gelegene Knoten eher ab, als weiter entfernte Knoten. Dies wird durch eine Liste realisiert, in der die noch zu bearbeitenden Knoten gespeichert werden. Wird ein Knoten besucht, so werden seine Nachbarknoten, die noch nicht besucht wurden, an die Liste angehängt und der aktuelle Knoten gelöscht. Ist kein Knoten in der Liste bricht der Suchalgorithmus ab [29].

Der Text-To-Building-Annotator nutzt die Breitensuche um bestimmte Zugehörigkeit zwischen Token zu finden. Da die Tiefensuche zu Beginn möglichst lange Wege geht ist die Breitensuche besser für den Annotator geeignet. Diese stellt sicher, dass zuerst möglichst kurze Verbindungen zwischen Wörtern gefunden, und verringert damit die Chance, dass Zugehörigkeiten falsch erkannt werden.

2.2. Basis der Visualisierung

2.2.1. OpenStreetMap

OpenStreetMap ist ein Kartendienst, dessen geografische Daten kollaborativ durch Nutzer erstellt und erweitert werden [16]. Diese Daten sind frei verwendbar und werden über die Overpass API zur Verfügung gestellt [28]. Das Text-To-Building-Projekt nutzt diese, um bereits bekannte Gebäude zu visualisieren, sollten sie im Text vorkommen.

2.2.2. OSMBuildings

Um die ermittelten Gebäudedaten zu visualisieren wird die OSMBuildings Bibliothek verwendet, die für die Programmiersprache JavaScript entworfen wurde. OSM Buildings generiert 3D Gebäude auf einer Karte, die aufgrund von Mapbox- oder Carto-Daten erstellt wird [27]. Die Gebäudedaten müssen dabei im GeoJSON-Format vorliegen und die Attribute Höhe, Höhe über dem Boden, Position, Anzahl der Geschosse, Farbe, Form sowie Dachform, Dachhöhe, Dachfarbe und Dachausrichtung können verarbeitet werden.

2.2.3. GeoViz WebApp

Die GeoViz WebApp ist eine Zusammenstellung von verschiedenen Kartenapplikationen, die in ein HTML-Framework eingebunden sind. Dieses bietet die Möglichkeit einen Text und die Visualisierung seiner Informationen in einer kompakten Nutzerschnittstelle, die an das Frontend des TextImagers (siehe Abschnitt 2.3.1) angelehnt ist, anzuzeigen. Dafür ist die WebApp in drei Teile unterteilt: ein Navigationspanel, ein Feld für das Dokument und ein Feld für die Karte. Sie wurde 2018 unter der Leitung von Tolga Uslu und Giuseppe Abrami im Text Technology Lab der Goethe Universität Frankfurt entwickelt. Das Frontend des Text-To-Building-Projekts ist als eine Erweiterung der GeoViz WebApp implementiert.

2.3. Anwendung und Erweiterung

2.3.1. TextImager

TextImager ist ein Framework, in dem NLP-Software und Visualisierungstools miteinander verbunden werden, um die Analyse von Texten und Textkorpora zu ermöglichen. Es wurde im Text Technology Lab der Goethe Universität Frankfurt basierend auf dem UIMA-Framework (siehe Abschnitt 2.1.1) entwickelt. Dabei lag der Fokus auf einer benutzerfreundlichen Schnittstelle, um das Nutzen von NLP Tools für Menschen ohne Programmierkenntnisse zu ermöglichen [17]. Da der Text-To-Building-Annotator auf dem UIMA-Framework basiert, besteht die Möglichkeit, ihn als Erweiterung in das TextImager-Framework einzubauen.

2.3.2. UIMA Database Interface

Das UIMA Database Interface ist eine Schnittstelle, die 2018 von Alexander Mehler und Giuseppe Abrami entwickelt wurde. Sie ermöglicht die Nutzung von UIMA-Dokumenten (siehe Abschnitt 2.1.1), die auf Datenbanken verwaltet werden. Das Interface bietet einen Mechanismus zum Abfragen der Dokumente und ist in Systeme einbindbar, in denen der Zugriff auf Dateien durch Berechtigungen verwaltet wird [2]. Sollte eine gemeinsame Nutzung der Annotationen des Text-To-Building-Annotators in Betracht gezogen werden, stellt das UIMA Database Interface eine geeignete Lösung dar, um dies umzusetzen.

2.3.3. Ontologien

Eine Ontologie in der Informatik ist eine Beschreibung von Begriffen und deren Beziehungen zueinander. Diese Struktur spiegelt Sachverhalte und Verhaltensweisen von Dingen in einem gewissen Anwendungsbereich wieder. Im Bereich des NLP können Ontologien als Hilfestellung für die richtige Interpretation von Textinhalt genutzt werden. Eine Maschine alleine erkennt nur die Abhängigkeiten zwischen zwei Token. Durch eine Ontologie kann zusätzlich entschieden werden, ob diese Abhängigkeit auch auf semantischer Ebene besteht [7]. Im Falle des Text-To-Building-Annotators kann eine Ontologie, die die Zusammenhänge einzelner Gebäudekomponenten beschreibt, die korrekte Annotation von Gebäuden unterstützen.

Die in diesem Kapitel vorgestellten Technologien decken alle benötigten Grundlagen für diese Arbeit ab. Darauf aufbauend wird im nächsten Kapitel geschildert, wie das Text-To-Building-Projekt Gebäude automatisch anhand eines beschreibenden Textes visualisiert.

3. Text-To-Building

In diesem Kapitel wird das Text-To-Building-Projekt vorgestellt. Im ersten Teilkapitel wird auf die Implementierung des Frontends eingegangen. Danach beschreibt das zweite Teilkapitel den Aufbau des Text-To-Building-Annotators und wie die Gebäudedaten durch sprachliche Analyse ermittelt werden.

3.1. Visualisierung der Gebäudedaten

Die Visualisierung der Gebäude setzt sich aus der Funktionalität der OSMBuildings Bibliothek, den Kartendaten des Carto-Dienstes und der GeoViz WebApp zusammen, die den Rahmen des Frontends bildet. Im Folgenden wird darauf eingegangen, wie diese Komponenten genutzt werden, und welche weiteren Funktionalitäten implementiert werden mussten, um die Evaluation zu ermöglichen.

Wie in Abbildung 3.1 zu sehen ist, bietet die GeoViz WebApp ein Feld für das Dokument und ein Feld für die Karte, auf der die Gebäude zu sehen sind. Da das Navigationspanel für das Frontend des Text-To-Building-Projekts nicht notwendig ist, verzichtet die Abbildung zu Gunsten der Übersichtlichkeit darauf. Die WebApp bezieht den Text aus dem Cas-Dokument, das in der sprachlichen Vorverarbeitung erstellt wird (siehe Kapitel 3.2). Dieses wird im XML-Format vom Text-To-Building-Annotator übergeben.

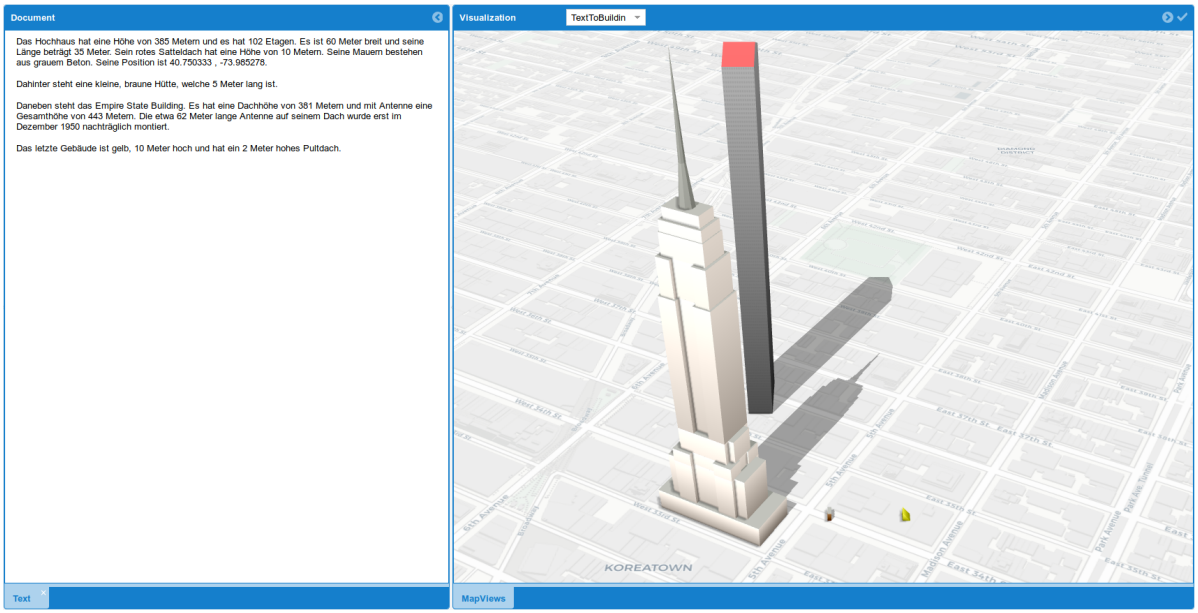

Abbildung 3.1.: Text-To-Building-Frontend

Die Visualisierung des Text-To-Building-Projekts, innerhalb der Anzeige der GeoViz Web-App, wird durch die OSMBuildings Bibliothek realisiert. Dafür wird zuerst ein OSMBuildings-Objekt mit Parametern, die die Ausrichtung, die Position und den Zoomfaktor der Karte bestimmen, erzeugt. Anschließend bildet das OSMBuildings-Objekt eine Karte ab, die auf dem "light_all"-Stil der Basiskarten des Carto-Dienstes basiert. Die erstellte Karte bietet automatisch die Möglichkeit, damit zu interagieren (Drehen, Zoomen und Bewegen). Danach folgt das Zeichnen der Gebäude. Jedes Gebäude wird mit der "addGeoJSON"-Funktion der OSMBuildings Bibliothek auf die Karte projiziert und intern mit einer Identifikationsnummer versehen, die für den Evaluationsprozess benötigt wird. Die GeoJSON-Daten liegen in einer JavaScript Datei als Variable definiert vor, die vom Text-To-Building-Annotator am Ende der Sprachanalyse erzeugt wird (siehe Abschnitt 3.2.4).

Zusätzlich zur bisher vorgestellten Arbeitsweise wurden weitere Funktionalitäten implementiert, die für die Evaluation (siehe Kapitel 4) benötigt werden. Bevor die WebApp geladen wird erscheint eine Nachricht an den Nutzer, in der die Aufgabe der Evaluation erläutert wird (siehe Abbildung A.1).

Der Evaluationsprozess beinhaltet eine Zuordnung eines Gebäudes zu einem Paragraphen, die durch gemeinsames markieren erfolgt (siehe Abschnitt 4.1). Dafür wird nach dem Laden von Gebäuden und Text eine Datenstruktur erstellt, die alle notwendigen Daten für eine Evaluationssitzung speichert. Sie beinhaltet: die Zeitstempel für den Start- und Endzeitpunkt, einen Zähler für die Anzahl der markierten Gebäude, die Identifikationsnummer des momentan markierten Gebäudes, ein Array mit verschiedenen RGB-Werten für die Markierungsfarben, ein Array, in das die Daten jeder Markierung eingetragen wird, und einen Schalter, der regelt, ob als nächstes ein Gebäude oder ein Paragraph markiert werden muss. Anschließend wird der Zeitstempel für den Startpunkt gesetzt.

Bei einem Mausklick auf ein Gebäude wird zunächst überprüft, ob ein Gebäude markiert werden kann. Falls der Schalter den Wert *"Kein Gebäude wurde markiert"* beinhaltet, wird die aktuelle Gebäude-ID der Datenstruktur mit der Identifikationsnummer des ausgewählten Gebäudes überschrieben und ein neuer Markierungseintrag für das Gebäude angelegt. Danach wird das Gebäude eingefärbt (siehe Abbildung A.5) und der Schalter auf *"Ein neues Gebäude wurde markiert"* gesetzt. Die Farbe ergibt sich aus der Anzahl der markierten Gebäude und des Arrays, das die Markierungsfarben speichert.

Ein Mausklick auf einen Paragraphen löst die zweite Markierungsfunktion aus. Sie prüft zuerst, ob der Schalter auf *"Ein neues Gebäude wurde markiert"* steht. Falls er dies tut, erscheint der ausgewählte Paragraph in der selben Farbe, mit der das zuvor markierte Gebäude eingefärbt wurde (siehe Abbildung A.6). Zusätzlich wird dem zugehörigen Eintrag des Markierungsarrays die Identifikationsnummer des Paragraphen hinzugefügt. Diese entspricht, der Identifikationsnummer des Gebäudes, das er beschreibt. Zuletzt wird der Zähler für die Anzahl der markierten Gebäude um Eins erhöht und der Schalter wechselt zu *"Kein Gebäude wurde markiert"*.

In der rechten oberen Ecke der WebApp befindet sich ein Haken (siehe Abbildung A.2). Wenn man auf diesen klickt wird die Evaluationssitzung abgeschlossen. Der Zeitstempel für den

Endzeitpunkt wird gesetzt und ein Dokument erstellt, das die für die Evaluation relevanten Daten enthält. Die Informationen der einzelnen Markierungen, der Start- sowie Endzeitpunkt und die Gesamtbearbeitungszeit werden abgespeichert. Zum Schluss wird das Dokument per E-Mail versendet, um die Auswertung der Daten zu ermöglichen, und die Abschlussnachricht wird dem Nutzer angezeigt (siehe Abbildung A.3).

3.2. Extrahieren der Gebäude Informationen

Das Backend des Text-To-Building-Projekts setzt sich aus den Komponenten, die in Abschnitt 2.1 behandelt werden, und dem Text-To-Building-Annotator zusammen. Dieser ist wiederum in den Height-, Depth-, Width-, Position-, Roof- und Building-Annotator unterteilt. Wie diese Komponenten zusammen agieren, um relevante Daten zu ermitteln und in geeigneter Form zu speichern, wird in diesem Abschnitt dargestellt.

3.2.1. UIMA Typsystem

Die Basis des Text-To-Building-Annotators bildet das Typsystem, das man in Abbildung 3.2 sehen kann. Die Typen Height, Width und Depth haben jeweils nur ein relevantes Attribut: den Wert. Diese Datentypen sind für die Annotation von Attributen, die die Höhe, Länge oder Breite beschreiben. Der Position-Datentyp besitzt die Attribute Longitude und Latitude, wodurch Positionen auf der Karte annotiert werden können und aufgrund des Color-Datentyps werden die RGB-Werte von Farben im Text annotiert. Für die Annotation von Dächern im Text verwendet man den Roof-Datentyp. Er nutzt den Color- und den Height-Typ um die Farbe und Höhe des Dachs zu speichern und das Shape-Attribut beinhaltet die Form des Dachs. Die Gebäudedaten, die später visualisiert werden, werden mithilfe des Building-Typs annotiert. Zusätzlich zu den Attributen, die auf den vorher genannten Datentypen basieren, müssen auch noch die Höhe über dem Boden, der Name, die Anzahl der Etagen, die Rotation auf der Karte und die Form des Hauses annotiert werden. Hierfür hat der Building-Typ jeweils ein Attribut. Abschließend haben alle Datentypen ein „*begin*“- und ein „*end*“-Attribut, die Anfang und Ende des zugehörigen Wortes markieren.

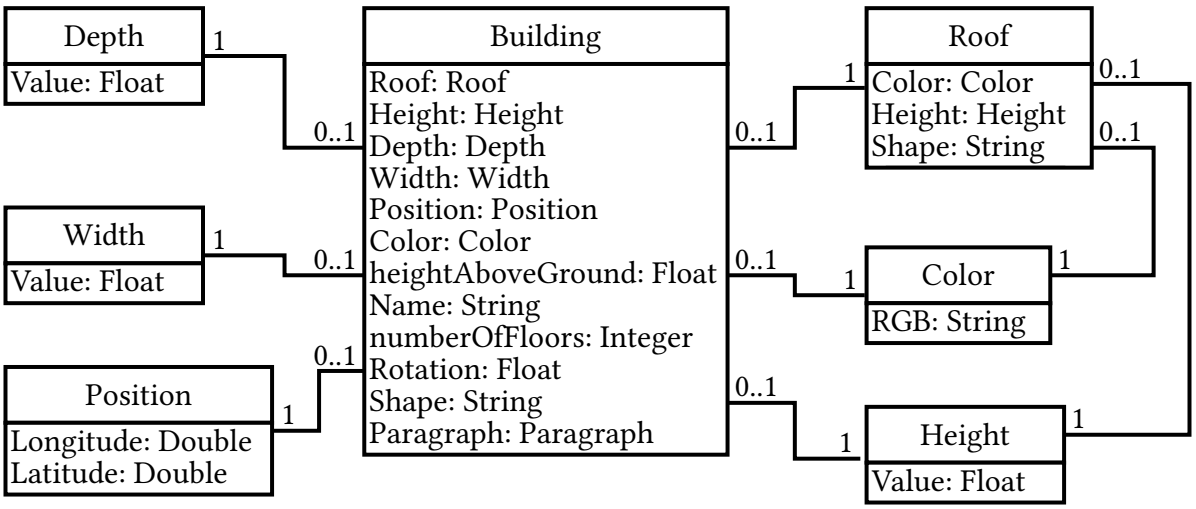

Abbildung 3.2.: UIMA Typsystem

3.2.2. Annotationsbasis

Um die Funktionalität des Text-To-Building-Annotators zu gewährleisten benötigt er bestimmte Annotationen. Diese werden durch eine UIMA-Pipeline erzeugt, die in Abbildung 3.3 zu sehen ist. Sie zeigt die Nutzung der einzelnen Komponenten und wie sie aneinander geschaltet sind. Die Pfeile nach außen zeigen die Ausgabe jedes Annotators.

Nachdem für den zu analysierenden Text ein neues Cas-Dokument erstellt wird, in dem alle Annotation hinterlegt werden, erfolgt die Verarbeitung durch die Komponenten des DKPro™ Core Frameworks (siehe Abschnitt 2.1.2). Der Text wird durch den LanguageToolSegmenter in Sätze und Token unterteilt und der ParagraphSplitter markiert die Paragraphen des Textes. Danach wird jedem Token durch den LanguageToolLemmatizer ein Lemma und durch den StanfordPosTagger ein POS-Tag zugewiesen. Im Anschluss annotiert der StanfordNamedEntityRecognizer alle Named Entities, sollten sie im Text vorkommen und zum Abschluss analysiert der CoreNLPDependencyParser die grammatikalische Struktur des Textes, die er in Dependencies abspeichert.

Auf die Tools des DKPro™ Core Frameworks folgt der CoReference-Annotator. Er erkennt anhand des Textes, ob sich mehrere Wörter in verschiedenen Sätzen auf ein Subjekt beziehen, und generiert daraus die zugehörigen CoReference-Annotationen (siehe Abschnitt 2.1.3).

Am Ende der Pipeline agiert der Color-Annotator, der jedes Lemma auf einen Farbnamen überprüft. Falls er Farben im Text findet, werden diese mitsamt ihrer RGB-Werte annotiert.

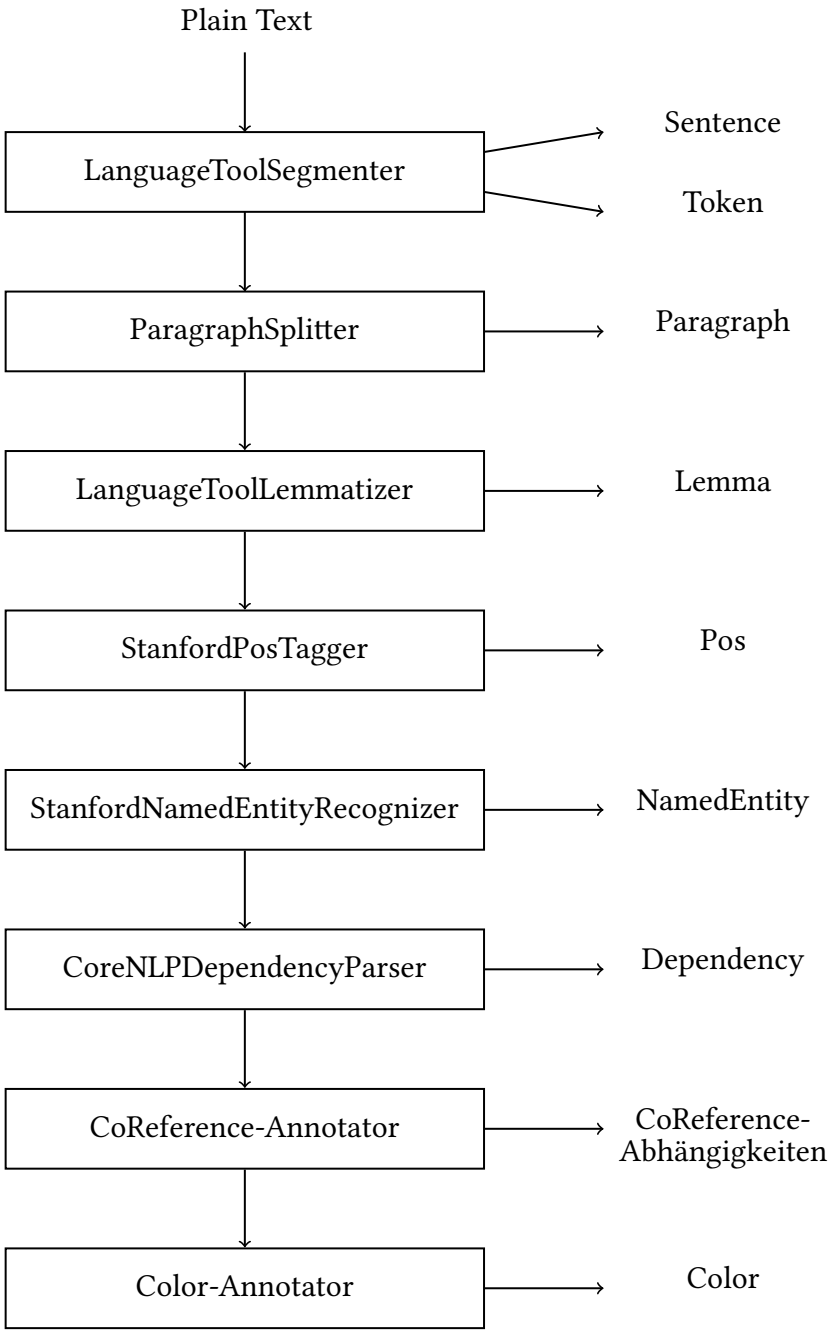

Abbildung 3.3.: UIMA-Pipeline, auf deren Annotationen der Text-To-Building-Annotator aufbaut

3.2.3. Attributannotatoren

Den Anfang des Text-To-Building-Annotators machen die Attributannotatoren. Für die Datentypen Height, Width, Depth und Position wurde jeweils ein Annotator implementiert, der die Werte für das entsprechende Attribut annotiert, falls ein passendes Wort im Text vorkommt. Die Annotatoren benötigen zu diesem Zweck die Token-Annotationen, deren Lemmata und die Dependency-Annotationen.

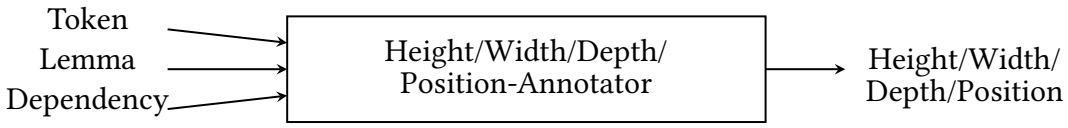

Abbildung 3.4.: Height/Width/Depth/Position-Annotator

Die Annotatoren für den Height-, den Width- und den Depth-Datentyp unterscheiden sich in den Wörtern, nach denen im Text gesucht wird. Die restliche Funktionalität ist dieselbe. Zuerst wird durch die Liste aller Token iteriert, die im aktuellen Cas-Dokument gespeichert sind. Wenn das Lemma des Tokens das Wort „Höhe/Breite/Länge“ enthält oder dem Wort „hoch/breit/lang“ entspricht, wird ein Height/Width/Depth-Objekt erstellt. Anschließend wird für dieses Objekt ein zugehöriger Wert gesucht, wofür die Funktion „*searchForNumber*“ implementiert wurde.

Sie agiert wie eine Breitensuche (siehe Abschnitt 2.1.6) auf einem Graphen, wobei die Token die Knoten und die Dependencies die Kanten des Graphen repräsentieren (siehe Abbildung 3.5). Falls ein Token, dessen Text einer Zahl entspricht, gefunden wird, überprüft die Funktion die unmittelbare Umgebung des Tokens. Diese wird nach einem Token abgesucht, das „Meter“ oder „Metern“ beinhaltet, um auszuschließen, dass Zahlen ohne Einheit (z.B. Jahreszahlen) angenommen werden. Wenn ein solches Token gefunden wird, bricht die Suche ab. Abschließend wird dem Height/Width/Depth-Objekt der Wert des Zahlentextes zugeordnet (siehe Implementierung 1).

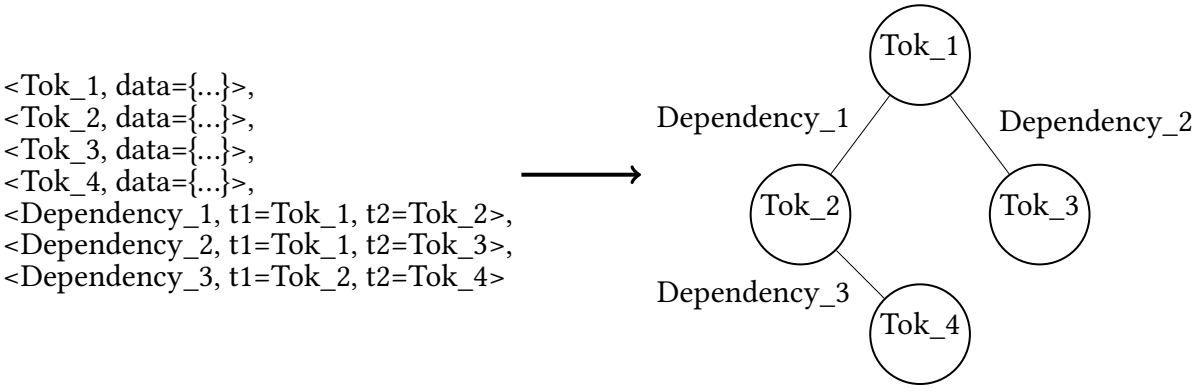

Abbildung 3.5.: Graph, der sich aus Token und Dependencies zusammensetzt

Implementierung 1 überprüfe Token auf Zahlenwert

Input: T: aktuell zu bearbeitendes Token
Cas: Cas-Dokument, auf dem gearbeitet wird

```
if T.Text ist eine Zahl then
    for { d : Dependency-Annotationen aus Cas } do
        if d enthält T und „Meter“ /„Metern“ then
            H.Value := T.Zahlenwert
        end if
    end for
end if
```

Da die Position eines Gebäudes durch mehrere Zahlen beschrieben wird, funktioniert der zuvor vorgestellte Ansatz für den Position-Annotator nicht. Hierfür wird auf die grammatikalische Struktur eines Beispielsatzes zurückgegriffen, die in der Abbildung 3.6 zu sehen ist.

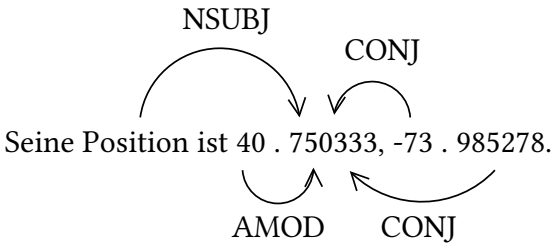

Abbildung 3.6.: Grammatikalische Struktur zum Erkennen der Position

Am Anfang des Bearbeitungsprozesses erfolgt eine Iteration über alle Token des Cas-Dokuments. Hierbei wird überprüft, ob das Lemma des aktuellen Tokens das Wort „Position“ enthält. Wenn ein passendes Token gefunden wird, beginnt die Suche nach dem Längen- und Breitengrad. Die Nachkommastellen des Breitengrades erkennt der Annotator anhand der NSUBJ-Dependency, die vom Position-Token wegführt. Von dort aus führt eine AMOD-Dependency zum Ganzzahl-Anteil des Breitengrades und zwei CONJ-Dependencies zu den Teilen des Längengrades. Insofern alle Teile des Breiten- und Längengrades gefunden wurden, annotiert der Position-Annotator die Werte in einem Position-Objekt auf dem Cas-Dokument.

3.2.4. Roof-Annotator und Building-Annotator

Dieser Abschnitt zeigt, wie der Roof-Annotator und der Building-Annotator die sprachliche Verarbeitung abschließen.

Ehe die beiden Annotatoren Dächer und Gebäude finden können, besteht die Notwendigkeit, Wörter zu definieren, nach denen gesucht werden kann. Um die Ausbeute der Suche zu maximieren, werden vor dem Aufruf der Annotatoren zwei SPARQL-Anfragen an die Wikidata Datenbank (siehe Abschnitt 2.1.5) geschickt (siehe Abbildung 3.7). Es werden Namen von Einträgen, die eine Untergruppe der Klasse „*Dach*“ oder „*Gebäude*“ sind, abgefragt, um möglichst viele verschiedene Bezeichnungen dafür zu erhalten. Daraufhin werden die Ergebnisse für die spätere Nutzung in zwei unterschiedlichen Listen abgespeichert.

```
SELECT ?label
WHERE {
   ?item rdfs:label „Gebäude“@de.
   ?build wdt:P279+ ?item.
   ?build rdfs:label ?label.
   FILTER ( lang ( ?label ) = „de“ ).
}
```

Abbildung 3.7.: SPARQL-Abfrage zum Erstellen einer Liste mit Gebäudenamen

Mit der vorher erstellten Dach-Liste arbeitet der Roof-Annotator weiter. Aufgrund der Token, deren Lemmata und deren POS-Tags sucht er nach Wörtern, die Dächer beschreiben. Für die gefundenen Dächer werden per Breitensuche über die vorhandenen Dependencies zugehörige Height-Annotationen und Color-Annotationen ermittelt. Diese speichert er gemeinsam mit der Form des Daches in einer Roof-Annotation ab (siehe Abbildung 3.8).

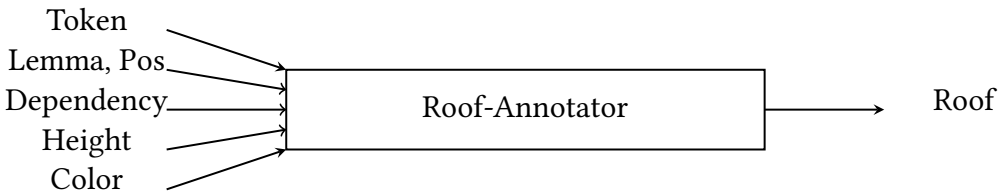

Abbildung 3.8.: Roof-Annotator

Der erste Schritt besteht darin, für jedes Token des Cas-Dokuments zu überprüfen, ob einer der Namen der Dach-Liste das Lemma des Tokens enthält und ob der POS-Tag des Tokens aussagt, dass es sich um ein Nomen handelt. Da nur Substantive relevant für die Annotation sind, wird durch den Vergleich mit dem POS-Tag die Fehlerquote des Annotators reduziert.

Wird ein Token gefunden, das die Bedingungen erfüllt, erstellt der Roof-Annotator ein Roof-Objekt und setzt seine Attribute auf Standardwerte. So sind die Daten für die Visualisierung vollständig, auch wenn Attribute nicht angegeben oder gefunden werden. Die Standardwerte entsprechen einem Flachdach in einem dunklen Rot, das 10 Meter hoch ist. Die Höhe ist nur relevant, wenn eine andere Form aber keine Höhe angegeben ist.

Als nächstes erfolgt die Annotation der Dachform. Dafür wird der Name des Daches mit einer Liste abgeglichen, die alle Dachformen enthält, die von der OSMBuildings Bibliothek unterstützt werden. Abbildbar sind: ein Flachdach, ein Satteldach, ein Pyramidendach, ein Pultdach und ein Kuppeldach. Nicht unterstütze Dachformen vermerkt der Annotator als Flachdach.

Im Anschluss wird von dem Dach-Token ausgehend per Breitensuche, wie in Abschnitt 3.2.3 anhand der Abbildung 3.5 beschrieben, nach zugehörigen Attributen gesucht. Hierfür wurde die Funktion „*handleRoofAttribute*“ implementiert (siehe Implementierung 2). Für jedes Token, das abgesucht wird, folgt ein Aufruf der Funktion.

Für das Token wird geprüft, ob im Cas-Dokument eine zugehörige Height- oder Color-Annotation vorliegt. Falls eine passende Annotation existiert und das zugehörige Attribut des Dach-Objekts undefiniert ist, wird es mit einem Verweis auf die gefundene Height- oder Color-Annotation überschrieben. Sind alle Token abgearbeitet, speichert der Roof-Annotator das Roof-Objekt als Annotation auf dem Cas-Dokument ab.

Implementierung 2 „handleRoofAttribute“

```
Input: R: Dach, zu dem die Attribute gehören
  T: Token, das überprüft wird
  Cas: Cas-Dokument auf dem gearbeitet wird

  for { Attribut : { Height, Color } } do
      for { a : Attribut-Annotationen aus Cas } do
          if a.Text == T.Text und R.Attribut ist undefiniert then
              R.Attribut := a.Value return
          end if
      end for
  end for
```

Nachdem alle Roof-Annotationen getätigt wurden, löst der Building-Annotator den Roof-Annotator ab. Mittels aller vorher erstellten Annotationen erkennt er bereits bekannte und unbekannte Gebäude. Jedes dieser Gebäude wird anschließend in einer Building-Annotation hinterlegt (siehe Abbildung 3.9) oder, im Falle von bereits bekannten Gebäuden, direkt in einem GeoJSON-Objekt gespeichert. Da alle Gebäudedaten gemeinsam in einem GeoJSON-Objekt zusammengeführt werden sollen, wird vor Beginn der Verarbeitung ein solches erzeugt.

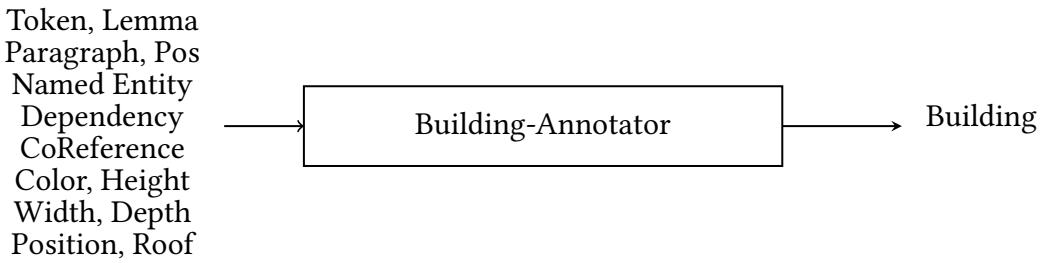

Abbildung 3.9.: Building-Annotator

Zuerst werden bereits bekannte Gebäude erkannt und verarbeitet. Zu diesem Zweck iteriert der Building-Annotator über alle Named Entity-Annotationen und stellt für jede eine Anfrage an die Overpass API. Darüber werden geografische Daten über das Gebäude erhalten, sollte OpenStreetMap über diese verfügen. Die erhaltenen Daten werden im JSON-Format übermittelt, aufgrund dessen der Building-Annotator diese in das GeoJSON-Format überführt, bevor er sie dem anfangs erstellten GeoJSON-Objekt hinzufügt.

Der nächste Schritt besteht darin, alle Token zu finden, die ein Gebäude entweder beschreiben oder sich auf das beschreibende Token beziehen. Dazu wird innerhalb jedes Paragraphen nach Gebäudenamen gesucht. Dies geschieht per Iteration über alle Token-Annotationen des Cas-Dokuments. Falls das Lemma des aktuellen Tokens in einem der Namen der Liste, die durch die SPARQL-Abfrage in Abbildung 3.7 erstellt wurde, vorkommt und es sich dabei um ein Nomen handelt, wird ein neues Building-Objekt erstellt. Dessen Name wird mit dem Text des Tokens überschrieben und das Paragraph-Attribut mit der Identifikationsnummer des aktuell bearbeiteten Paragraphen überschrieben. Im Anschluss wird mithilfe der CoReference-Annotationen ermittelt, welche Token innerhalb des Paragraphen dasselbe Gebäude beschreiben und diese in einer Liste gemeinsam hinterlegt. So werden beispielsweise im vierten Paragraphen des Evaluationstextes (siehe Abschnitt 4.1) die Token der Wörter „*Leuchtturm*“, „*seine*“, „*Er*“ und „*Er*“ gefunden und gespeichert.

Mithilfe der zuvor erstellten Liste bestimmt der Building-Annotator die Attribute, die das Gebäude weiter beschreiben. Ausgehend von jedem Token der Liste, wird, wie in Abschnitt 3.2.3, per Breitensuche nach Attributen gesucht. Um Token auf ein mögliches Attribut zu untersuchen, wurde die Funktion „*handleBuildingAttribute*“ implementiert (siehe Implementierung 3). Für jedes Token, das von der Suche gefunden wird, erfolgt ein Aufruf dieser Funktion.

Aufgrund des übergebenen Tokens überprüft die Funktion, ob ein zugehöriges Attribut der Typen Height, Width, Depth, Position, Color oder Roof existiert. Sollte dies der Fall sein, wird das passende Attribut des Building-Objekts mit einem Verweis auf die gefundene Attribut-Annotation überschrieben und die Funktion bricht ab. Wenn keine passende Annotation gefunden wird, folgen zwei weitere Untersuchungen. Wenn das Lemma des übergebenen Tokens den Wörtern „Wand“, „Mauer“ oder „Fassade“ entspricht, wird in der unmittelbaren Nähe davon eine Color-Annotation gesucht. Findet der Annotator solch eine Annotation, wird das Color-Attribut des Building-Objekts mit einem Verweis auf diese überschrieben und die Funktion bricht ab. Zuletzt wird geprüft ob das Lemma den Wörtern „Etage“ oder „Geschoss“ entspricht. Auf diese Weise kann die Anzahl der Etagen eines Gebäudes erkannt

werden. Existiert in unmittelbarer Nähe ein Zahlenwert, wird das „*numberOfFloors*“-Attribut des Building-Objekts damit überschrieben und die Funktion endet.

Implementierung 3 „handleBuildingAttribute“

```
Input: B: Gebäude, zu dem die Attribute gehören
  T: Token, das überprüft wird
  Cas: Cas-Dokument auf dem gearbeitet wird

  for { Attribut : { Height, Width, Depth, Position, Color, Roof } } do
      for { a : Attribut-Annotationen aus Cas } do
          if a.Text == T.Text und B.Attribut ist undefiniert then
              B.Attribut := a.Value return
          end if
      end for
  end for
  if T.Lemma == „Wand“ oder T.Lemma == „Mauer“ oder T.Lemma == „Fassade“ then
      Suche von T aus nach einer Color-Annotation
      if Color-Annotation c gefunden then
          B.Color := c return
      end if
  end if
  if T.Lemma == „Etage“ oder T.Lemma == „Geschoss“ then
      Suche von T aus nach einer Zahl
      if Zahl z gefunden then
          B.AnzahlGeschosse := z return
      end if
  end if
```

Zum Abschluss der Verarbeitung wird jedes Building-Objekt von der Funktion „*standardizeBuilding*“ standardisiert, d.h. der Annotator setzt die Attribute, die nicht gefunden bzw. angegeben wurden, auf vorher definierte Standarwerte, damit keine Fehler aufgrund von unvollständigen Parametern im Frontend entstehen. Die Standarwerte entsprechen einem grauen Haus mit grauem Flachdach, das 10 Meter in der Breite, Länge und Höhe misst und neben dem Empire State Building in New York City steht.

Nachdem der Building-Annotator alle Gebäude annotiert hat, müssen die für die Visualisierung notwendigen Daten in eine Form gebracht werden, die das Frontend benutzen kann. Dies erreicht er, indem aus jeder Building-Annotation ein GeoJSON-Objekt erstellt wird. Alle Attribute der Building-Annotation werden dabei in eine Form gebracht, die die OSMBuildings Bibliothek verarbeiten kann (siehe Abschnitt 2.2.2). Anschließend wird damit das GeoJSON-Objekt erweitert, das zu Beginn der Verarbeitung durch den Building-Annotator erstellt wurde. Dieses wird zuletzt, als Variable in einer JavaScript-Datei definiert, an einem Ort abgespeichert, auf den das Frontend danach zugreifen kann. Um auch das Cas-Dokument,

aus dem das Frontend den Text bezieht, bereitzustellen, wird dieses in eine XML-Datei konvertiert.

Zusammenfassend wird in diesem Kapitel das gesamte Text-To-Building-Projekt vorgestellt. Der erste Teil beschreibt, wie das Frontend Gebäude neben dem zugehörigen Text anzeigen kann, der zweite Teil erläutert, wie die benötigten Daten für diese Visualisierung zustande kommen. Um die Güte des Text-To-Building-Projekts bewerten zu können wurde eine Evaluation durchgeführt, deren Aufbau und Ergebnisse zeigt das nächste Kapitel.

4. Evaluation

Dieses Kapitel behandelt die Evaluation des Text-To-Building-Projekts. Am Anfang wird der Aufbau der Evaluationsaufgabe geschildert. Danach schließt sich die Erläuterung der Durchführung und die Vorstellung der Ergebnisse an.

4.1. Aufbau

Die Evaluation soll die Güte des Text-To-Building-Projekts zeigen. Dafür wurde eine Aufgabe erstellt, die mehrere Probanden absolvierten. Während jedes Evaluationsdurchgangs wurden Daten erhoben, mithilfe derer repräsentative Ergebnisse ermittelt werden konnten.

Die Aufgabe bestand darin, jedem Paragraphen das Gebäude zuzuweisen, das von ihm beschrieben wurde. Der Aufbau ist in Abbildung A.4 zu sehen. Die Zuweisung erfolgte über die Nutzerschnittstelle, die im Kapitel 3.1 vorgestellt wurde. Wenn der Proband alle Markierungen gesetzt hatte (Abbildung A.7 zeigt eine mögliche Belegung) sollte die Evaluationssitzung per Klick auf den dafür vorgesehenen Haken abgeschlossen werden.

Der Text der Evaluationsaufgabe lautete wie folgt:

> *Das Hochhaus hat eine Höhe von 255 Metern und es hat 102 Etagen. Es ist 60 Meter breit und seine Länge beträgt 35 Meter. Sein rotes Satteldach 10 Meter hoch. Seine Mauern bestehen aus grauem Beton. Seine Position ist 40.750333 , -73.985278.* (1)
>
> *Daneben steht eine kleine, braune Hütte, welche 5 Meter lang und 3 Meter breit ist.* (2)
>
> *In der Nähe steht die Konditorei die 5 Meter hoch und 3 Meter lang ist. Ihre Position ist 40.750388, -73.986651. Ihr blaues Pultdach ist 2 Meter hoch.* (3)
>
> *Der Leuchtturm ist 130 Meter hoch und seine Mauern sind weiß. Er ist 20 Meter lang und 20 Meter breit. Er trägt ein Pyramidendach.* (4)
>
> *Eine große lila Apotheke steht am Ende der Straße. Sie ist 15 Meter lang und 10 Meter breit. Sie besitzt eine Höhe von 35 Metern und ein rotes Flachdach. Ihre Position ist 40.749688, -73.986651.* (5)
>
> *In der Umgebung steht der gelbe Teeladen, welcher 4 Meter hoch ist. Er ist 5 Meter lang und 2 Meter breit. Sein schwarzes Satteldach ist 2 Meter hoch.* (6)

In Abbildung 4.1 ist eine genaue Ansicht der Gebäude zu sehen, die für die Evaluation angezeigt wurden. Sie sind mit der Nummer des zugehörigen Paragraphen versehen.

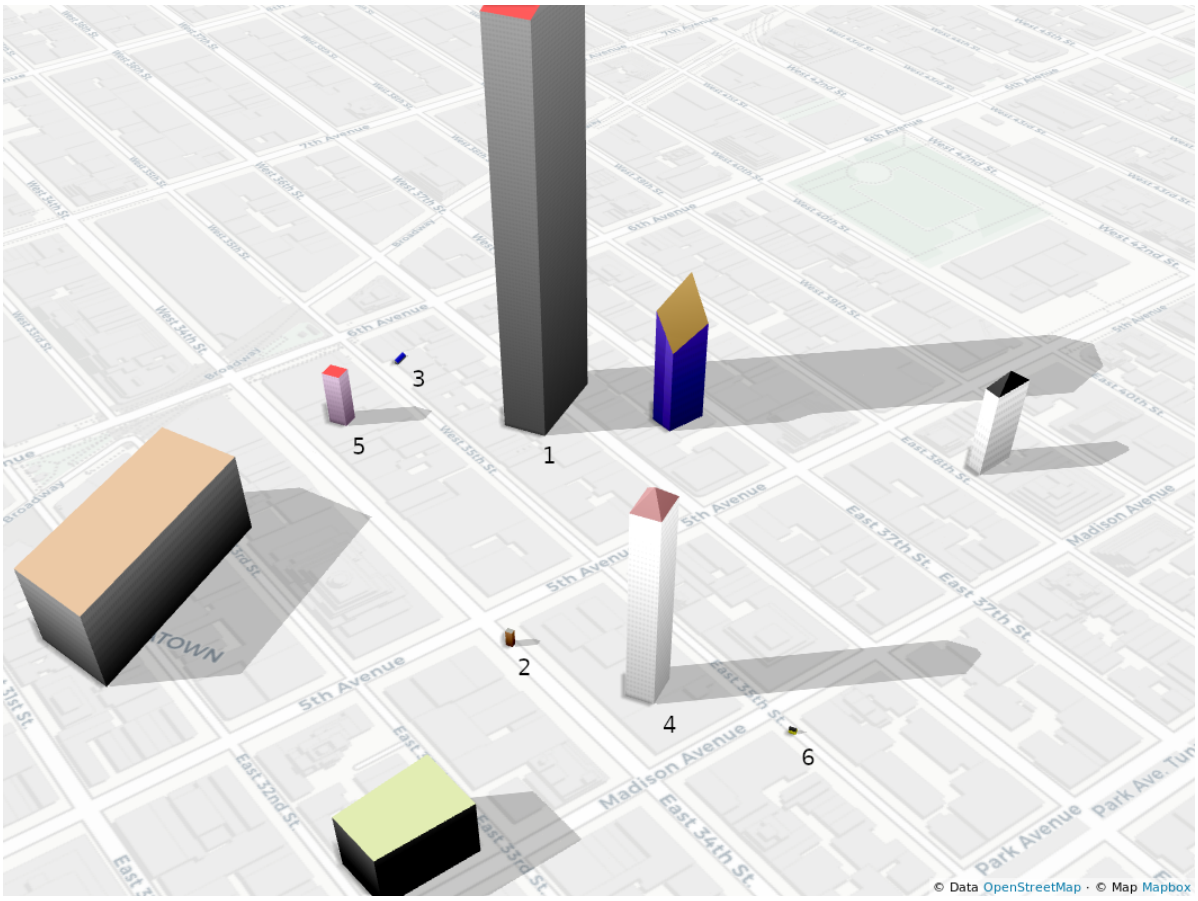

Abbildung 4.1.: Gebäude, die Teil der Evaluationsaufgabe sind

Die Gebäudedaten der markierten Gebäude wurden mithilfe des Text-To-Building-Annotators ermittelt. Die zusätzlichen Gebäude wurden manuell in das GeoJSON-Objekt eingefügt, um zu verhindern, dass Gebäude per Ausschlussverfahren einem Paragraphen zugeordnet werden. Dies erhöht die Genauigkeit der Ergebnisse, die durch die Evaluation gesammelt werden.

4.2. Ergebnisse

Die Umsetzung der Evaluation erfolgte durch eine Internetseite, die das Frontend zur Verfügung stellt. Der Hyperlink der Seite wurde über soziale Medien verteilt. So haben 25 Probanden erfolgreich an der Evaluation teilgenommen.

Es wurden zwei relevante Faktoren betrachtet. Während jeder Evaluationssitzung wurde gespeichert, wie lange der Evaluationsprozess gedauert hat und welche Gebäude richtig zugeordnet waren. Im Durchschnitt hat die Bearbeitung der Aufgabe 190.48 Sekunden gedauert,

wobei der schnellste Proband 90.73 Sekunden und der langsamste 343,92 Sekunden benötigte. Die Erfolgsrate ein Gebäude richtig zu erkennen betrug 88.67%. In der Abbildung 4.2 sind die Erfolgsraten der einzelnen Gebäude und durch welche Paragraphen sie beschrieben werden, zu sehen. Das Hochhaus und die Apotheke wurden zu 96% richtig erkannt, der Leuchtturm zu 92%. Die Konditorei und der Teeladen haben eine Erfolgsrate von 84% und die Hütte wurde mit 80% am seltensten richtig markiert.

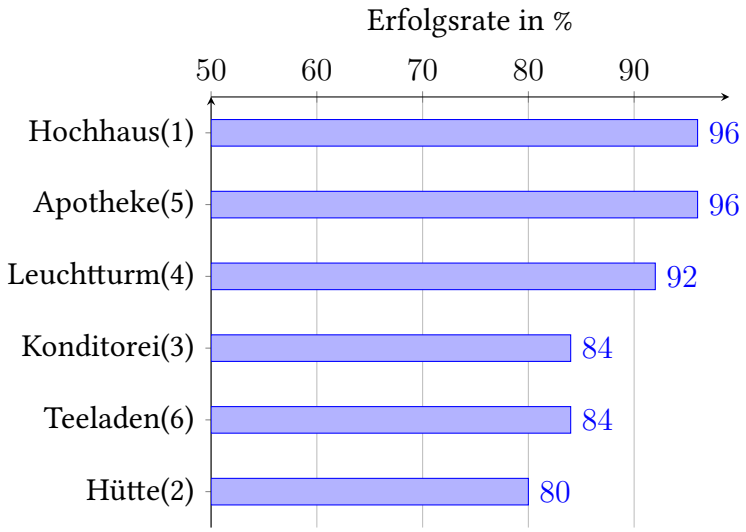

Abbildung 4.2.: Erfolgsrate in Abhängigkeit des Paragraphen

Es wurde zusätzlich die Anzahl der Wörter innerhalb eines Paragraphen in die Auswertung miteinbezogen. So ergab sich, dass die Erfolgsrate steigt, je mehr Wörter ein Paragraph umfasste bzw. je umfassender das Gebäude beschrieben wurde (siehe Abbildung 4.3). Eine Ausnahme macht der Paragraph, der den Leuchtturm beschreibt. Trotz der, relativ zu den Paragraphen mit höherer Erfolgsrate, niedrigen Anzahl an Wörtern wurde der Leuchtturm in 92% der Fälle erkannt.

Das Ziel des Text-To-Building-Projekts ist es, Gebäude anhand eines beschreibenden Textes zu visualisieren. Die Ergebnisse der Evaluation zeigen, dass dieses Ziel erreicht wurde. Bei Angabe von ausreichenden Informationen zu den Gebäude ergibt sich eine hohe Erkennungsrate. Trotzdem sind Schwachstellen erkennbar. So ist die Visualisierung von Gebäuden mit gering ausfallender Beschreibung deutlich weniger wiedererkennbar, als bei genauerer Beschreibung durch mehr Attribute. Dies zeugt von einem unausgereiften Standardisierungsverfahren der Gebäudeparameter. Auch hat die Visualisierung aller Gebäude Ähnlichkeit mit einem typischen Hochhaus. So entspricht die „*Hütte*“ aus Paragraph 2 nicht einer authentischen Darstellung einer Hütte.

Basierend auf den Erkenntnissen aus diesem Kapitel werden im folgenden Kapitel Möglichkeiten zur Verbesserung, Erweiterung und Anwendung des Text-To-Building-Projekts vorgestellt.

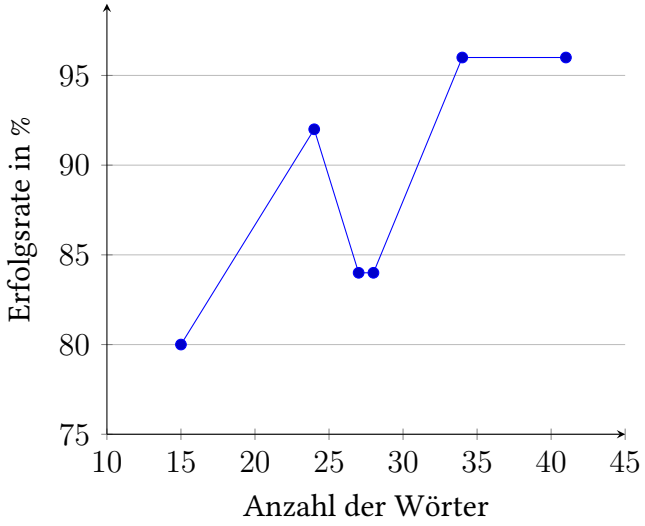

Abbildung 4.3.: Erfolgsrate in Abhängigkeit von der Wortanzahl

5. Futurework

Dieses Kapitel beschreibt, wie mit dem Text-To-Building-Projekt weiter verfahren werden kann und sollte. Es werden mehrere Ansätze erörtert, wie das Projekt entweder verwendet oder erweitert bzw. verbessert werden kann.

5.1. TextImager

Bislang existiert das Text-To-Building-Projekt nur als eigenständige Softwarezusammensetzung, bestehend aus Frontend und Backend. Aufgrund der Basis der beiden Komponenten, bietet sich eine Integration des Text-To-Building-Projekts in den TextImager an. Wie in Abschnitt 2.3.1 beschrieben, basiert der Mechanismus zur sprachlichen Analyse des TextImagers, genau wie der Text-To-Building-Annotator, auf dem UIMA Framework. Die einzelnen Komponenten des Text-To-Building-Annotators können somit reibungslos integriert werden. Hierbei ist die Bereitstellung von den benötigten Annotationen vonnöten, um die Funktionalität der Komponenten des Text-To-Building-Annotators gewährleisten zu können. Zusätzlich dazu gleicht die Struktur der GeoViz Webapp, in die das Frontend des Text-To-Building-Projekts eingebaut ist, der Oberfläche des TextImagers. Dadurch kann auch das Frontend ohne großen Aufwand zum TextImager hinzugefügt werden.

5.2. UIMA Database Interface

Die Bearbeitung von Texten und die Sammlung von Daten durch den Text-To-Building-Annotator beschränkt sich momentan auf die Nutzung einer Person. Projekte wie WikiData (Abschnitt 2.1.5) und OpenStreetMap (Abschnitt 2.2.1) zeigen, welch eine Sammlung an Daten entstehen kann, wenn viele Menschen sich daran beteiligen. Beispielsweise könnte anhand von historischen Texten ein Modell einer Stadt, die heutzutage nicht mehr existiert, durch die gemeinsame Nutzung des Text-To-Building-Annotators entstehen. Die Funktionalität einer kollaborativen Datenspeicherung kann durch das UIMA Database Interface integriert werden. Es ermöglicht UIMA-Anwendungen Dokumente auf Datenbanken zu nutzen und zu verwalten (siehe Abschnitt 2.3.2). Da der Text-To-Building-Annotator auf dem UIMA-Framework (siehe Abschnitt 2.1.1) basiert, ist eine Erweiterung des Annotators um das UIMA Database Interface mühelos umsetzbar.

5.3. Unterstützung der sprachlichen Analyse durch Ontologien

Sowohl die sinkende Wiedererkennungsrate, bei unausführlicher Beschreibung der Gebäude, als auch die unauthentische Visualisierung von Gebäuden, die einem Hochhaus nicht ähneln,

lassen sich auf eine unzureichende Verarbeitung des Textinhalts durch den Text-To-Building-Annotator zurückführen. Durch ein ontologisches Modell (siehe Abschnitt 2.3.3) lässt sich seine rein syntaktische Sprachanalyse um eine semantische Überprüfung der gefundenen Wortverbindungen erweitern.

Eine passende Ontologie könnte wie folgt aufgebaut sein: Der Begriff „*Bauwerk*“ und weitere Begriffe, die verschiedene Bauwerke beschreiben, bilden eine Hierarchie. Alle Begriffe sind Unterklassen von „*Bauwerk*“ und jeder Begriff, der ein Gebäude spezifischer beschreibt, als ein anderer, ist eine Unterklasse des anderen. Zusätzlich zu dieser Hierarchie hat jeder Gebäudebegriff Relationen zu Begriffen, die die zur Visualisierung notwendigen Parameter beschreiben. Sie unterteilen sich in „*zwingend benötigt*“ und „*optional*“ ein. Ausgehend von einem gefundenen Gebäudenamen ist es anhand des ontologischen Modells möglich, gezielt nach allen benötigten und optionalen Parametern zu suchen.

Wird im Text ein Begriff gefunden, den die Ontologie abdeckt, können die benötigten Daten gezielt gesucht und fehlende Parameter dem Begriff entsprechend standardisiert werden. Dadurch wird umgangen, dass z.B. eine Hütte standardmäßig mit einer Höhe von 10 Metern visualisiert wird.

Weiterführend ermöglicht eine Ontologie, dass verschiedene Teile eines Gebäudes erkannt und richtig zugeordnet werden. Begriffe wie „*Dach*“, „*Rumpf*“ oder „*Flügel*“ können in das Modell miteinbezogen werden und mit ihren benötigten Parametern in Relation stehen. Ein Gebäude kann genauer dargestellt werden, wenn es in kleinere, signifikante Teile unterteilt wird. Darüber hinaus eröffnet die Aufteilung von Gebäuden die Möglichkeit, größere Gebäudekomplexe zu erkennen und zu visualisieren.

Ein Vorteil von dieser Herangehensweise ist, dass die Ontologie erweiterbar ist. Je größer das Modell ist, desto größer ist die Vielfalt der Visualisierung. Die Funktionalität des Annotators wächst also mit der Vollständigkeit des ontologischen Modells.

6. Fazit

Das Ziel dieser Arbeit ist es, ein Programm zu entwickeln, das eine Gebäudebeschreibung in Form eines Textes erhält, diese mittels NLP-Software automatisch analysiert und schließlich die ausformulierten Gebäude auf einer Karte präsentiert. Dies ermöglicht Nutzern ohne Kenntnis von Programmiersprachen, Software zur 3D-Modellierung oder anderer NLP-Begrifflichkeiten, ein Modell von einem oder mehreren Gebäuden zu erstellen.

Nachdem zuerst die technologischen Voraussetzungen geklärt werden, folgt eine Vorstellung des Text-To-Building-Projekts und wie es funktioniert. Danach werden die Evaluationsergebnisse dargelegt. Anhand eines Beispieltext und zugehörigen Modellen von Gebäuden wurde eruiert, dass die Erfolgsrate ein Gebäude zu erkennen, wenn es aufgrund eines vorliegenden Textes visualisiert wurde, bei 88,67% liegt. Die Erfolgsrate sinkt, je unvollständiger das Gebäude beschrieben wurde. Erkennbare Schwächen sind hierbei ein unausgereiftes Standardisierungsverfahren und zu unspezifische Visualisierungsarten für verschiedene Arten von Gebäuden. Im Anschluss folgt ein Ausblick darauf, wie mit dem Text-To-Building-Projekt in Zukunft weiter verfahren werden kann. Es wird gezeigt warum eine Integration in den TextImager (Abschnitt 2.3.1) möglich und sinnvoll ist, wie das Projekt durch das UIMA Database Interface (Abschnitt 2.3.2) erweitert werden kann und wie Front- und Backend von einer Implementierung eines ontologischen Modells profitieren würden.

Abschließend lässt sich folgendes Fazit ziehen: Das Ziel, eine automatische Visualisierung von Gebäuden auf der Basis von sprachlichem Input zu entwickeln, wurde erreicht. Jedoch bieten die einseitige Art der Visualisierung und die simpel gehaltene Modellierung des Building-Datentyps Raum für Verbesserungen. Die Umsetzung des Text-To-Building-Projekts ist keineswegs perfekt, aber sie ist eine solide Grundlage für weitergehende Forschung im Gebiet der automatischen Gebäudevisualisierung.

Literatur

[1] Giuseppe Abrami. *Color-Annotator*. 2018.

[2] Giuseppe Abrami und Alexander Mehler. „A UIMA Database Interface for Managing NLP-related Text Annotations“. In: *Proceedings of the Eleventh International Conference on Language Resources and Evaluation (LREC-2018)*. 2018.

[3] Howard Butler u. a. „Geojson specification“. In: *Geojson. org* (2008).

[4] Kai-Uwe Carstensen u. a. *Computerlinguistik und Sprachtechnologie: Eine Einführung*. Springer-Verlag, 2009.

[5] *CARTO Introduction*. URL: https://cartodb.readthedocs.io/en/latest/intro.html# (besucht am 17. 09. 2019).

[6] Hinrich Schütze Christopher D. Manning Prabhakar Raghavan. *An Introduction to Information Retrieval*. Apr. 2009, S. 22. URL: https://nlp.stanford.edu/IR-book/pdf/irbookonlinereading.pdf (besucht am 16. 08. 2019).

[7] Philipp Cimiano, Christina Unger und John McCrae. „Ontology-based interpretation of natural language“. In: *Synthesis Lectures on Human Language Technologies* 7.2 (2014), S. 1–178.

[8] *The Stanford Natural Language Processing Group*. URL: https://nlp.stanford.edu/software/nndep.html (besucht am 16. 08. 2019).

[9] *DKPro Core™ CoreNlpDependencyParser*. 2018. URL: https://dkpro.github.io/dkpro-core/releases/1.9.3/docs/component-reference.html#engine-CoreNlpDependencyParser (besucht am 11. 08. 2019).

[10] *DKPro Core™ LanguageToolLemmatizer*. 2018. URL: https://dkpro.github.io/dkpro-core/releases/1.9.3/docs/component-reference.html#engine-LanguageToolLemmatizer (besucht am 11. 08. 2019).

[11] *DKPro Core™ ParagraphSplitter*. 2018. URL: https://dkpro.github.io/dkpro-core/releases/1.9.3/docs/component-reference.html#engine-ParagraphSplitter (besucht am 03. 09. 2019).

[12] *DKPro Core™ LanguageToolSegmenter*. 2018. URL: https://dkpro.github.io/dkpro-core/releases/1.9.3/docs/component-reference.html#engine-LanguageToolSegmenter (besucht am 11. 08. 2019).

[13] *DKPro Core™ User Guide*. 2018. URL: https://dkpro.github.io/dkpro-core/releases/1.9.3/docs/user-guide.html (besucht am 11. 08. 2019).

[14] Marcus Goetz und Alexander Zipf. „OpenStreetMap in 3D–detailed insights on the current situation in Germany“. In: *Proceedings of the AGILE 2012 International Conference on Geographic Information Science, Avignon, France*. Bd. 2427. 2012, S. 2427.

[15] Thilo Gotz und Oliver Suhre. „Design and implementation of the UIMA Common Analysis System“. In: *IBM Systems Journal* 43.3 (2004), S. 476–489.

[16] Mordechai Haklay und Patrick Weber. „Openstreetmap: User-generated street maps“. In: *IEEE Pervasive Computing* 7.4 (2008), S. 12–18.

[17] Wahed Hemati, Tolga Uslu und Alexander Mehler. „TextImager: a Distributed UIMA-based System for NLP“. In: *Proceedings of COLING 2016, the 26th International Conference on Computational Linguistics: System Demonstrations*. Osaka, Japan: The COLING 2016 Organizing Committee, Dez. 2016, S. 59–63. URL: https://www.aclweb.org/anthology/C16-2013.

[18] Alexander Henlein. „Entity-Modell-basierte Anaphernresolution mittels Deep Learning (LSTMs)“. Magisterarb. Goethe Universität Frankfurt am Main, Sep. 2018.

[19] *Unstructured Information Management Architecture SDK*. URL: https://www.ibm.com/developerworks/data/downloads/uima/index.html (besucht am 16. 08. 2019).

[20] Christopher Manning u. a. „The Stanford CoreNLP natural language processing toolkit“. In: *Proceedings of 52nd annual meeting of the association for computational linguistics: system demonstrations*. 2014, S. 55–60.

[21] *Mapbox - Wikipedia*. URL: https://en.wikipedia.org/wiki/Mapbox (besucht am 17. 09. 2019).

[22] *Stanford Named Entity Recognizer (NER)*. URL: https://nlp.stanford.edu/software/CRF-NER.html (besucht am 16. 08. 2019).

[23] Stefan Luber. *Was ist Natural Language Processing?* 2016. URL: https://www.bigdata-insider.de/was-ist-natural-language-processing-a-590102/ (besucht am 14. 08. 2019).

[24] Nurzhan Nurseitov, Michael Paulson, Randall Reynolds und Clemente Izurieta. „Comparison of JSON and XML data interchange formats: a case study.“ In: *Caine* 9 (2009), S. 157–162.

[25] *OpenScienceMap*. URL: http://www.opensciencemap.org/s3db/#&scale=16&rot=-37&tilt=65&lat=40.708&lon=-74.011 (besucht am 23. 09. 2019).

[26] *OSM2World*. URL: https://wiki.openstreetmap.org/wiki/OSM2World (besucht am 23. 09. 2019).

[27] *Copyright · OSMBuildings*. 2019. URL: https://osmbuildings.org/copyright/ (besucht am 11. 08. 2019).

[28] *Overpass API - OpenStreetMap Wiki*. 2019. URL: https://wiki.openstreetmap.org/wiki/Overpass_API (besucht am 21. 08. 2019).

[29] Stuart Russell und Peter Norvig. *Künstliche Intelligenz*. Pearson Deutschland GmbH, 2012.

[30] *A gentle introduction to the Wikidata Query Service*. 2019. URL: https://www.wikidata.org/wiki/Wikidata:SPARQL_query_service/A_gentle_introduction_to_the_Wikidata_Query_Service (besucht am 10. 09. 2019).

[31] *Stanford typed dependencies manual*. URL: https://nlp.stanford.edu/software/dependencies_manual.pdf (besucht am 19. 09. 2019).

[32] *ViziCities*. URL: https://github.com/robhawkes/vizicities (besucht am 23. 09. 2019).

[33] *Wikidata*. 2019. URL: https://www.wikidata.org/wiki/Wikidata:Main_Page (besucht am 23. 08. 2019).

[34] Graham Wilcock. „Introduction to linguistic annotation and text analytics“. In: *Synthesis Lectures on Human Language Technologies* 2.1 (2009), S. 107.

Abbildungsverzeichnis

A. Abbildungen

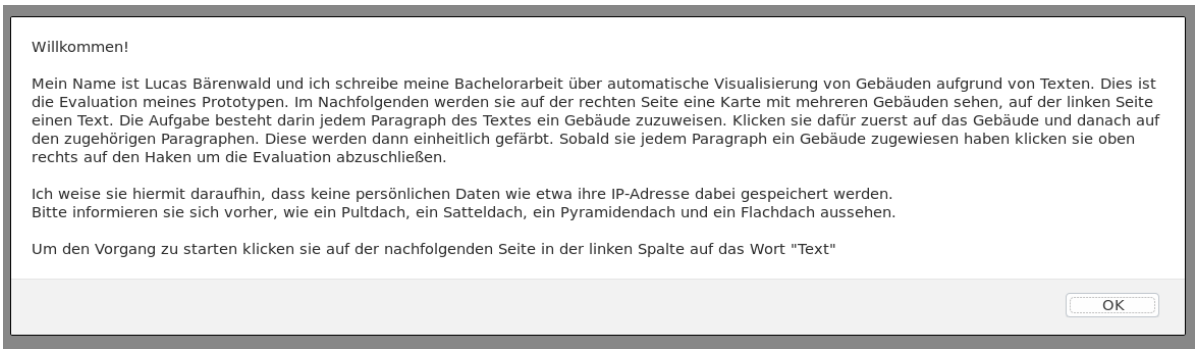

Abbildung A.1.: Erläuterung der Aufgabe

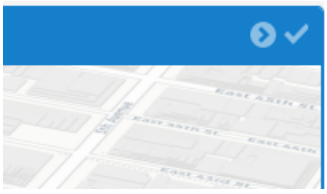

Abbildung A.2.: Abschlusshaken

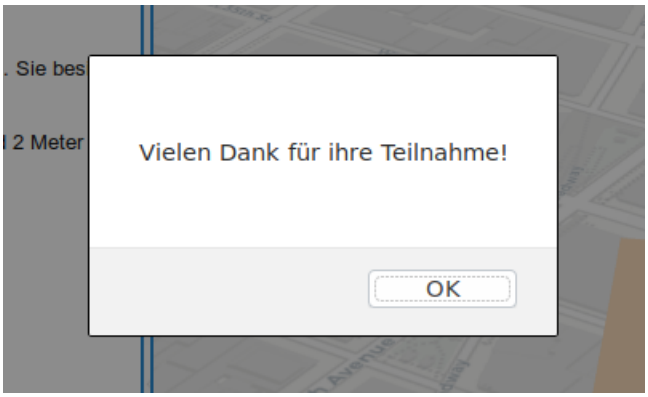

Abbildung A.3.: Abschlusstext

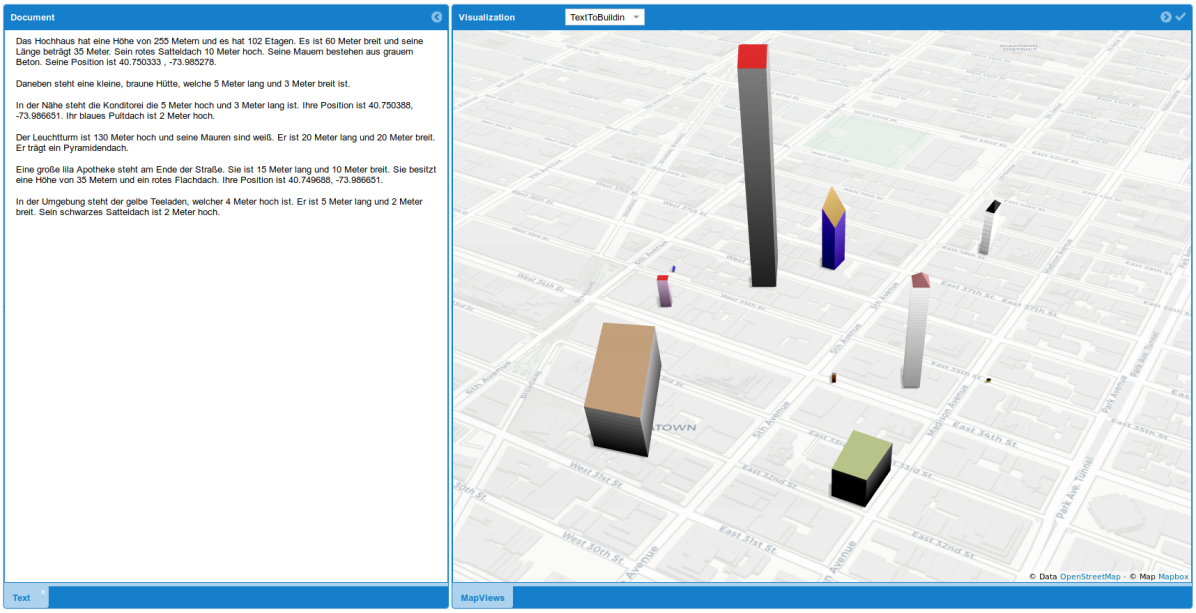

Abbildung A.4.: Aufbau der Evaluation

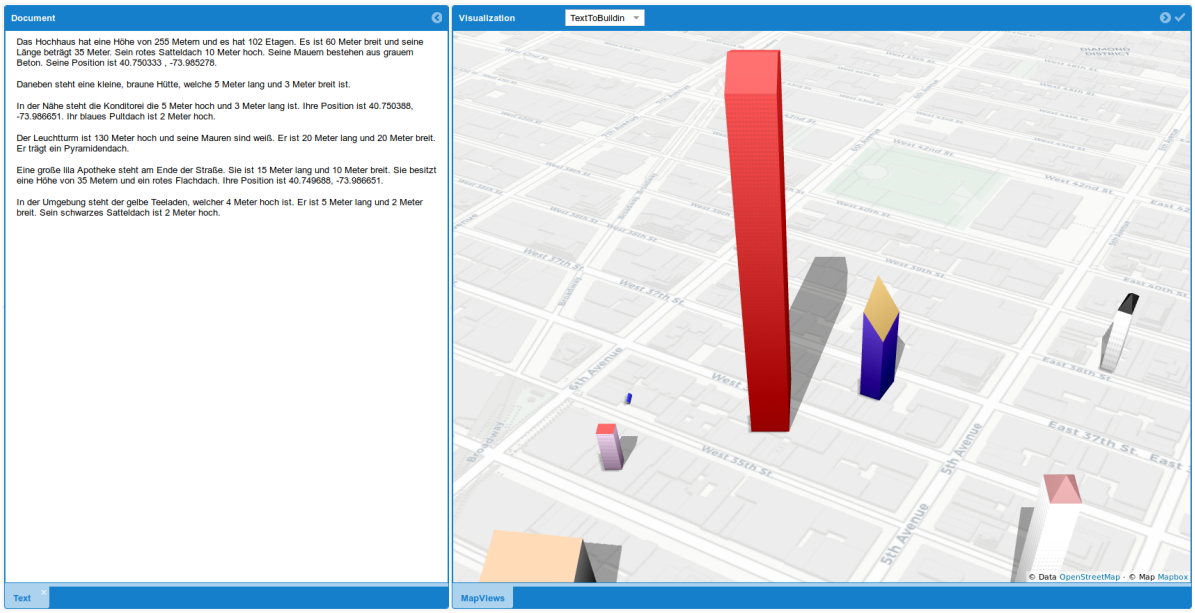

Abbildung A.5.: Markieren eines Gebäudes

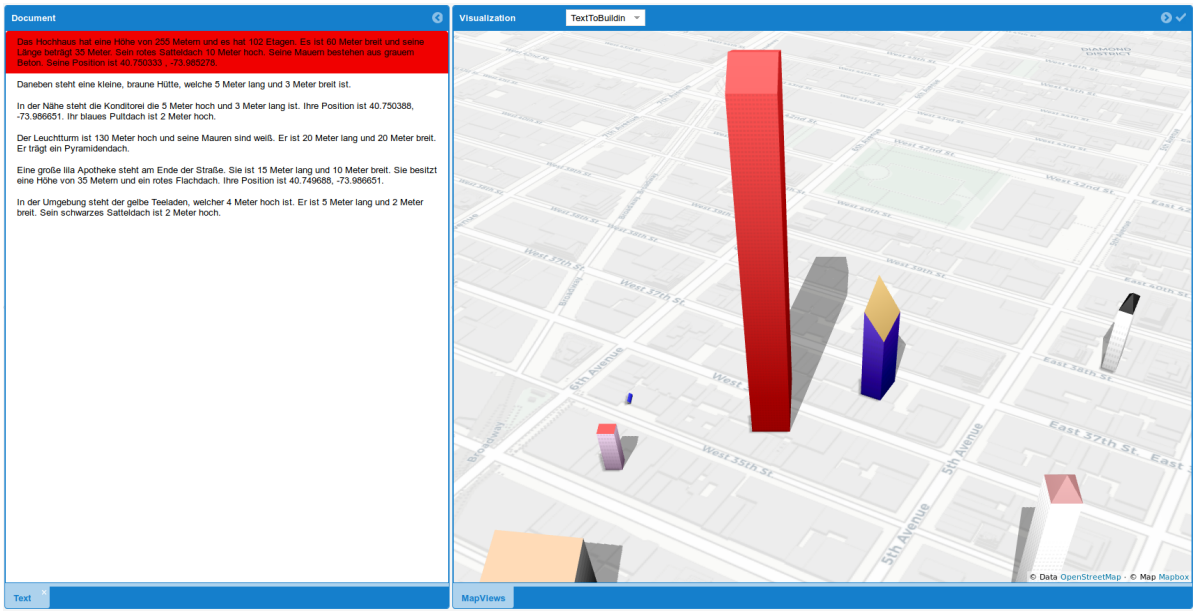

Abbildung A.6.: Markieren eines Paragraphen

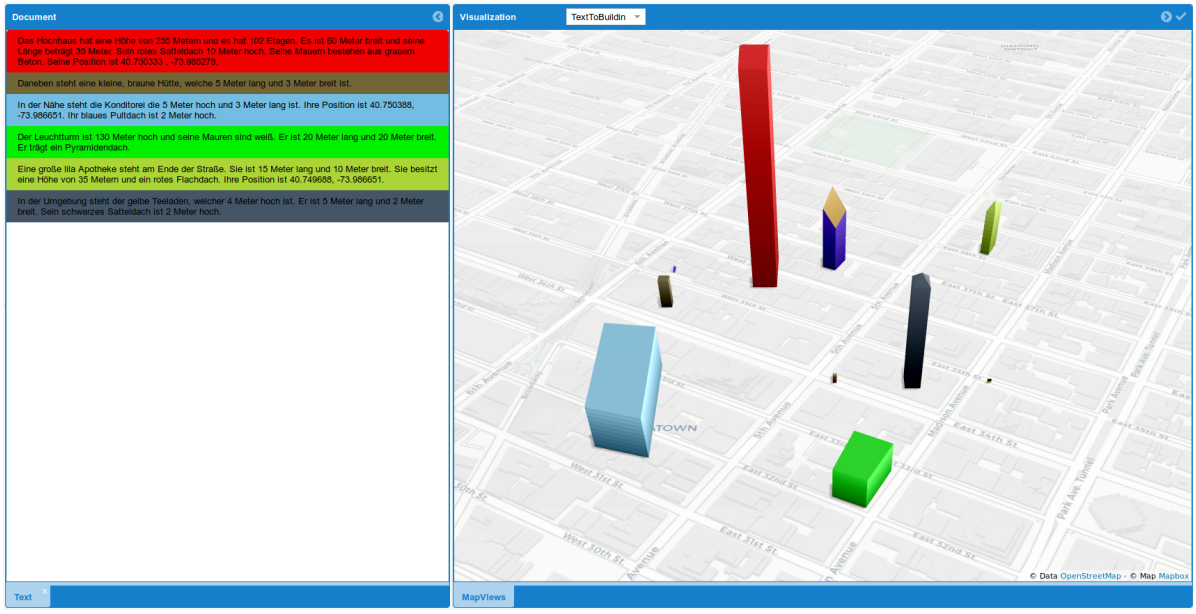

Abbildung A.7.: Markierungen Beispiel