

Bachelor-Forschungsprojekt Informatik: Relevante OSM-Tags vorschlagen

Marco Hildebrand, 3137242, st3137242@stud.uni-stuttgart.de

Lukas Baur, 3131138, st141998@stud.uni-stuttgart.de

Felix Bühler, 2973410, st117123@stud.uni-stuttgart.de

October 24, 2018

Abstract

Die vom *Institut für Formale Methoden der Informatik Stuttgart* entwickelte textbasierte Suchmaschine *OSCAR*, die OpenStreetMap-Daten auf Eingabe von OSM-Tags durchsucht, liefert unbefriedigende Ergebnisse auf anderweitige textuelle Eingaben. Im Rahmen unseres Bachelor-Forschungsprojekt Informatik sollte diese Lücke geschlossen werden, indem eine Anfrage an das von uns entwickelte System eine Menge an damit verwandten, relevanten Tags zurückgibt.

1 Einleitendes

1.1 Projektrahmen

Die Arbeit wurde im Rahmen des *Bachelor-Forschungsprojekts Informatik* in der Zeit vom April bis Oktober 2018 angefertigt. Diese Ausarbeitung stellt die inhaltliche Dokumentation des entwickelten Moduls dar.

1.2 Initiale Problemstellung

Grundlage für unsere Arbeit war die Suchmaschine *OSCAR*, die vom *Institut für Formale Methoden der Universität Stuttgart* entwickelt wurde.

OSCAR durchsucht auf Eingabe eines *OpenStreetMap-Tags* die zugehörige Datenbank nach passenden Einträgen und bereitet das Suchresultat grafisch auf. Ein *Tag* ist in OpenStreetMap wie folgt definiert:

$$\textit{key}=\textit{value}$$

Ein *key* wird benutzt, um ein Themenbereich zu charakterisieren, es repräsentiert einen Typ oder beschreibt ein Feature. Außerdem werden Tags vereinzelt als Namespaces verwendet [1].

Der *value*-Teil stellt ein Wert des Features da. Typische Werte sind Eigenschaften oder Zahlen [1]. Beispiele für Tags sind *building=yes*, *building=house* oder *highway=service* [2][3].

Da die Eingabe auf Tags beschränkt ist, benötigt ein User zur Suche einen passenden Tag. Diese Lücke soll mithilfe dieses Projekts geschlossen werden. Das zu entwickelnde System soll auf Eingabe eines natürlichen Wortes der englischen Sprache möglichst eng verwandte, relevante OpenStreetMap-Tags vorschlagen.

1.3 Abgrenzungen

Unsere Arbeit konzentriert sich auf die Suche der relevanten Tags zu einem eingegebenen Wort. Formaler ausgedrückt besteht unsere Eingabe aus genau einem Wort der englischen Sprache, das nicht in der zugrundeliegenden Stop-Word-Liste enthalten ist.

2 Projekt-Durchführung

2.1 Planungsaspekte

Zu Beginn unserer Arbeit grenzten wir unser Projekt thematisch ein und überlegten uns eine grobe Vorstrukturierung. Dazu gliederten wir unser Projekt in **drei** wesentliche Bausteine:

Im zeitlich ersten Arbeitsblock sollten wir uns mit der Darstellung, der Qualität und der Möglichkeit des Zugriffs der Daten vertraut machen. Im Folgenden überlegten wir uns eine aufbereitete brauchbare Daten-Zwischenform, auf deren Grundlage die spätere Suche durchgeführt werden soll. Der dritte Arbeitsbaustein galt der eigentlichen Such-Implementierung.

Die bearbeiteten Arbeitspakete werden im folgenden inhaltlich beschrieben. Die Pakete sind intern zeitlich sequentiell beschrieben, überlappen sich allerdings in Ihrer Abarbeitung. Der Grund hierfür sind Abhängigkeiten, wie zum Beispiel, dass die Datenaufbereitung an die Repräsentation des Suchalgorithmus angepasst werden muss, zuvor aber Daten als Grundlage der Suche beschafft sein müssen.

2.2 Datenbeschaffung

2.2.1 Download des OSM Wikis

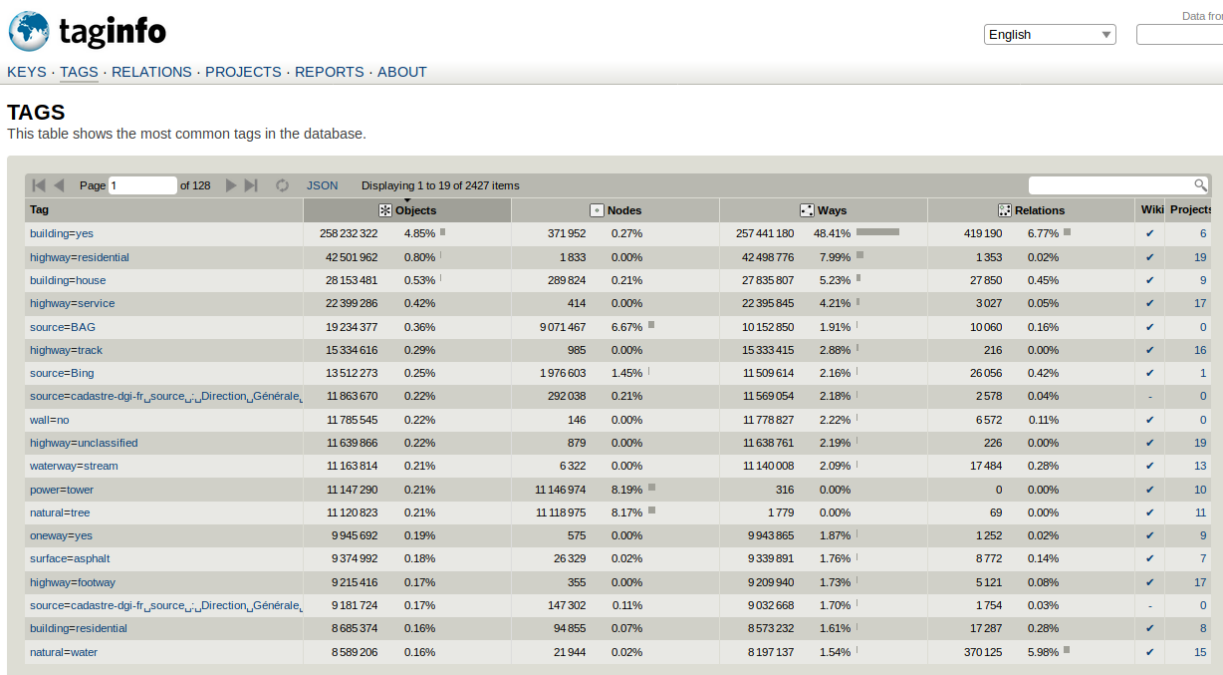
Unsere anfängliche Recherche begannen wir mit der Website von OpenStreetMap [4], insbesondere mit dem zugehörigem Wiki [5]. Das OSM-Wiki verfügt über eine ausführliche Dokumentation vieler gängiger OSM-Tags. Unser Ziel war es, auf alle vorhandenen Daten-Tupel, bestehend aus einem gültigen Tag und einer zugehörigen Tag-Beschreibung, lokalen Zugriff zu haben.

Leider bestand nur die Möglichkeit eine veraltete Version des OSM-Wiki von 2013 herunter zu laden, die zudem jedoch frei von erkennbaren Strukturen war, und uns somit keinen Ansatz lieferte.

2.2.2 Der Versuch mit tagInfo

Zwischenzeitlich versuchten wir alternativ mithilfe der Website *taginfo* [6] an die gesuchten Daten zu gelangen. taginfo wurde in Zusammenarbeit von Jochen und Christian Topf entwickelt und sammelt auf Grundlage der OSM-Daten aktuell rund 2.500 Tags inklusive deren statistischen Charakteristika und teilweise Beschreibungen[7]. Zusätzlich besteht die Möglichkeit, deren komplette Datenbank herunterzuladen.

Leider stellten wir fest, dass die Beschreibungs-Einträge der Datenbank zu lückenhaft und damit



Tag	Objects	Nodes	Ways	Relations	Wiki	Project
building=yes	258 232 322 4.85%	371 952 0.27%	257 441 180 48.41%	419 190 6.77%	✓	6
highway=residential	42 501 962 0.80%	1 833 0.00%	42 498 776 7.99%	1 353 0.02%	✓	19
building=house	28 153 481 0.53%	289 824 0.21%	27 835 807 5.23%	27 850 0.45%	✓	9
highway=service	22 399 286 0.42%	414 0.00%	22 395 845 4.21%	3 027 0.05%	✓	17
source=BAG	19 234 377 0.36%	9 071 467 6.67%	10 152 850 1.91%	10 060 0.16%	✓	0
highway=track	15 334 616 0.29%	985 0.00%	15 333 415 2.88%	216 0.00%	✓	16
source=Bing	13 512 273 0.25%	1 976 603 1.45%	11 509 614 2.16%	26 056 0.42%	✓	1
source=cadastre-dgi-fr_source=Direction_Générale	11 863 670 0.22%	292 038 0.21%	11 569 054 2.18%	2 578 0.04%	-	0
wall=no	11 785 545 0.22%	146 0.00%	11 778 827 2.22%	6 572 0.11%	✓	0
highway=unclassified	11 639 866 0.22%	879 0.00%	11 638 761 2.19%	226 0.00%	✓	19
waterway=stream	11 163 814 0.21%	6 322 0.00%	11 140 008 2.09%	17 484 0.28%	✓	13
power=tower	11 147 290 0.21%	11 146 974 8.19%	316 0.00%	0 0.00%	✓	10
natural=tree	11 120 823 0.21%	11 118 975 8.17%	1 779 0.00%	69 0.00%	✓	11
oneway=yes	9 945 692 0.19%	575 0.00%	9 943 865 1.87%	1 252 0.02%	✓	9
surface=asphalt	9 374 992 0.18%	26 329 0.02%	9 339 891 1.76%	8 772 0.14%	✓	7
highway=footway	9 215 416 0.17%	355 0.00%	9 209 940 1.73%	5 121 0.08%	✓	17
source=cadastre-dgi-fr_source=Direction_Générale	9 181 724 0.17%	147 302 0.11%	9 032 668 1.70%	1 754 0.03%	-	0
building=residential	8 685 374 0.16%	94 855 0.07%	8 573 232 1.61%	17 287 0.28%	✓	8
natural=water	8 589 206 0.16%	21 944 0.02%	8 197 137 1.54%	370 125 5.98%	✓	15

Figure 1: Beispielhafte Datenbankeinträge der Datenbank von taginfo

für unsere Zwecke nicht geeignet sind. Als Zwischenlösung kombinierten wir unseren bisherigen Ansätze, indem wir die Tag-Einträge der heruntergeladenen taginfo-Datenbank als Grundlage für ein Crawlen der OSM-Wiki-Seite verwenden wollten: Wir wollten ausnutzen, dass ein Link zu einer tag-beschreibenden Wiki-Seite die folgende Form aufweist:

wiki.openstreetmap.org/wiki/Tag%3Key%3Dvalue

Wobei die Variablen *key* und *value* gemäß obiger Erklärung zu füllen sind. Unvorteilhafterweise stellte sich das Downloaden der Seiten schwieriger als gedacht heraus, da es zu manchen Tags noch nicht einmal eine Wiki-Seite gibt.

2.2.3 Finallösung mit OSM-Wiki-Sitemap

Schlussendlich verfolgten wir den finalen Ansatz, die aktuelle Sitemap der Wiki-Seite[8] herunterzuladen und anschließend die Links, welche die im vorhergehenden Abschnitt beschriebene Struktur ausweisen, herausgeschrieben.

Als Resultat hatten wir nun alle möglichen OSM-Wiki-Links lokal im *txt*-Format zur Verfügung. Es konnten die Daten für den nächsten Arbeitsschritt, dem Aufbereiten der Einträge, weiterverwendet werden.

2.3 Crawling und Datenaufbereitung

Eine Such-Engine auf Grundlage der HTML-Files aufzusetzen, schlossen wir aus mehreren Gründen aus. Zuerst, findet sich in den HTML-Files viele unnötige Informationen wieder, z.B. Bild-Links, *JavaScript*-Teile oder gar unwichtige Metainformationen der Seite. Das alles verlangsamt zum einen die spätere Suche, verfälscht zum anderen aber auch den späteren Such- und Indizierungsprozess. Unser Ziel war es, die Daten in eine solche Form umzuwandeln, dass unsere Such-Engine effizient darauf arbeiten kann.

2.3.1 Initiale Idee

Zu Beginn verfolgen wir die Idee, wiederkehrende Strukturen zu erkennen und in unser Suchranking miteinbeziehen. Existiert beispielsweise innerhalb der OSM-Wiki-Seite zu dem Tag *highway=residential* ein Paragraphen *related Tags* mit dem Eintrag *highway=tertiary*, so ist es für den Nutzer möglicherweise interessant, auf Eingabe von *highway* beide Ergebnisse vorzufinden, auch wenn die Beschreibung von *highway=tertiary* nicht allein zum Suchwort gepasst hätte. Es würde in folge dessen ein Beziehungs-Netzwerk aufgebaut werden.

2.3.2 Probleme des initialen Ansatzes

Unglücklicherweise stellten sich heraus, dass die Artikel des Wikis keiner spezifischen oder homogenen Form folgten. Zum einen sind die Strukturen zwischen den Artikeln sehr verschieden (zudem auch verschieden ausführlich), zum anderen finden sich inhaltlich ähnliche Absätze unter verschiedenen Überschriften, so beziehen sich “See Also”, “See also”, “see also”, “related tags”, “Related Tags” oder “Similar Tag” eigentlich semantisch auf dasselbe. Das Ziel, weitere Informationen aus Tabellen zu entnehmen konnte aufgrund der heterogenen Struktur der Seiten ebenfalls nicht erreicht werden.

2.3.3 Implementierte Variante

Um die spätere Suchanfrage dennoch befriedigend beantworten zu können, einigten wir uns auf eine Struktur, die die Seite in drei Teile zerteilt. Später werden diese dann unabhängig voneinander durchsucht und gewichtet in die Ergebnisliste eingebracht. Die Struktur der aufbereiteten Daten lässt sich der unteren Grafik Figure 2 entnehmen.

Der Crawler arbeitete die zuvor erstellte *txt*-Linkliste sequentiell ab, und konvertierte die Eingabe direkt in das in Figure 2 dargestellte Format.

Dafür verwendeten wir den Aufruf `scrapy crawl osmWiki -t json -o keys.json` unseres eigenen Crawlers.

Das Ergebnis dieser Arbeitsphase war eine vollständige Liste von *.json*-Objekten, die die textuellen Informationen der HTML-Seiten enthalten, Metainformationen wie Skripte und HTML-Tags jedoch nicht. Zudem sind die Informationen gruppiert und wir erhalten Zugriff auf die Auftrittshäufigkeit eines jeden Tags.

```

{
  "title": "<Tag-Name>",
  "url": "<URL zum OSM-Wiki>",
  "related_terms": "<Liste mit zugehörigen Tags, die als Tabelle zu finden waren>",
  "tables": [ <Tabellen, mit jeweils Zeilen/Spaltenattributen> ],
  "main_description": "<die textuelle Beschreibung des Tags bis zum ersten Absatz>",
  "content": "<Objekt, dass wiederum Objekte für jeden Absatz und dessen Inhalt speichert>",
  "side": {
    "description": "<textuelle Kurzbeschreibung>",
    "<optional weitere Paare>",
    "tag_info": {<Statistiken als Objekt>}
  }
}

```

Figure 2: Schematische Darstellung der *.json*-Struktur. Die Einträge mit spitzen Klammern markieren entsprechend Meta-Anweisungen

2.4 Suchanfrage beantworten

Die finale Phase verknüpft alle bereits getätigten Arbeiten zu einer gesamten Anwendung, die aus mehreren funktionalen Einheiten besteht. Aus der Figure 3 lassen sich die wesentlichen Komponenten erkennen.

Diese Dokumentation nutzt im Folgenden die Pipeline-artige Struktur der Implementierung aus, um die Implementierung von links her sequentiell zu erklären.

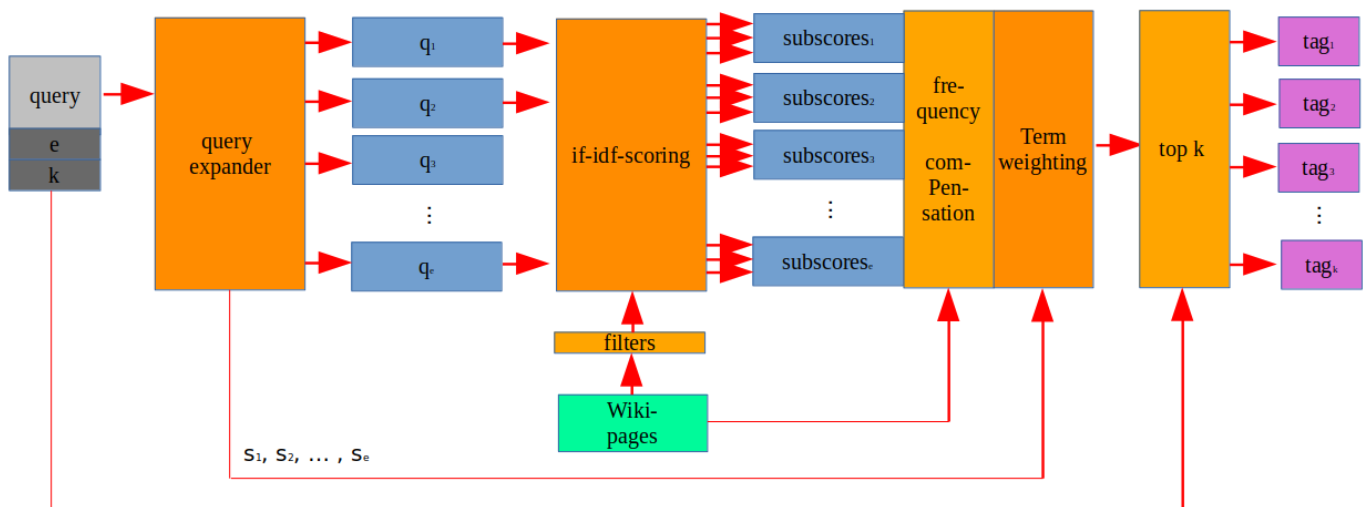


Figure 3: Gesamtstruktur der Such-Implementierung

2.4.1 Eingabeparameter

Die produzierende Ausgabe ist, neben den Wiki-Seite die den eigentlichen Suchraums überhaupt definiert, von drei Freiheitsgrade bestimmt.

Die *query*-Eingabe besteht aus einem Wort der englischen Sprache, wie in subsection 1.3 zu Beginn definiert. Die Ausgabe der Algorithmus soll $k \in \mathbb{N}_{>0}$ gültige Tags zurückliefern, die möglichst passend zur Eingabe *query* sind. Wie wir in subsubsection 2.4.2 zeigen werden, erweitern wir die Eingabe als ersten Schritt semantisch. Der Grad Expansion wird durch den Parameter $e \in \mathbb{N}_{>0}$ beeinflusst.

2.4.2 Query Expansion

Idee

Die dahinterstehende Idee des ersten Pipeline-Knotens ist die Folgende: Da die Tag-Auswahl relativ begrenzt und zudem auf syntaktisch identische Treffer begrenzt ist, wollen wir die Eingabe erweitern. Dies möchten wir mit einem konstruiertem Beispiel verdeutlichen: Angenommen der User sucht nach *Schnellrestaurant*, auf keiner OSM-Seite gebe es kein Vorkommen dieses Wortes, so würde die Suche keine Ergebnisse liefern. Durch Query-Expansion würden wir die Eingabe um die Wörter *Fastfood-Restaurant*, *Schnellrestaurants* und *Restaurant* erweitern. Als Resultat können wir so *amenity=fast_food* erwarten. Ähnliche Beispiele könnten sich auch für die Englische Sprache konstruieren lassen.

Realisierung

Wir verwendeten für die Umsetzung das Framework *Google word2vec*[9]. Es erlaubt auf Eingabe eines Wortes die Suche nach möglichst nahegelegenen Wörtern mit derselben Semantik. Der große Vorteil dadurch ist zusätzlich, dass wir parallel nach Singular und Plural der Eingabe suchen können, da deren Semantik praktisch identisch ist.

Das Resultat der Anwendung des Frameworks sind $e-1$ semantisch ähnliche Wörter, die zu dem Vektor der Eingabequery in nächster Umgebung lagen. Wir nennen diese im folgenden q_2, q_3, \dots, q_e . Implizit wird q_1 als die Anfangsquery betrachtet.

Neben den semantisch ähnlichen Queries speichern wir ebenfalls die vektorwertige (semantischen) Distanz zum Eingabevektor ab. Diese Ähnlichkeitswerte nennen wir entsprechend s_1, s_2, \dots, s_e mit der Eigenschaft $0 < s_i \leq 1$. Je höher dieser Wert, desto näher ist der Begriff der Ursprungsquery semantisch. Es dürfte offensichtlich sein, dass s_1 demnach den Wert 1 besitzt.

2.4.3 Scoring

Die *tf-idf-Komponente* ist dafür zuständig, möglichst relevante Tags zu den s_i zu finden. Dafür nutzen wir die aufbereiteten *.json*-Files des OSM-Wikis - wie in Figure 2 erklärt - als Grundlage für unsere Suche. Die Bewertung besteht aus zwei Phasen, die sequentiell ausgeführt werden. In der ersten Phase werden zu einem gegebenen Suchwort jeweils 3 Scores berechnet. Jeder der drei Scores beschreibt hierbei den *tf-idf*-Wert innerhalb eines Tag-Dokuments, repräsentativ für einen Dokumentteil. Anschließend werden die Zwischenscores gewichtet verrechnet. Zu Beginn erstellen wir drei verschiedene Indizes, die von dieselben Dimension sind. Der erste indiziert alle *main-description*-Teile der Wiki-Seiten, der zweite alle *side-description*-Teile und der letzte die restlichen Inhalte der *.json*-Einträge. Für jedes der q_i starten wir eine Suche in jedem Indizes, wir erhalten folglich die Zwischenergebnisvektoren $\vec{sub}_{i,0}, \vec{sub}_{i,1}, \vec{sub}_{i,2}$, stellvertretend für die drei Teile.

Die Idee war nun, eine sinnvolle Gewichtung der Treffer zu finden. Einen guten Treffer in der kurzen Main-Description soll in etwa so viel wert sein wie eine sehr passender Treffer in der ausführlichen Content-Sektion. Somit soll es möglich sein, von der absoluten Textlänge zu abstrahieren und gleichzeitig sehr kurze Seiten, die dadurch einen sehr hohen tf-idf-Score haben, zu dämpfen.

In der Praxis stellte sich eine gleichmäßige Gewichtung von je ein Drittel als zweckmäßig heraus. Die neuen, kombinierten tf-idf-Scores lassen sich demnach mit $c_1 = c_2 = c_3 = \frac{1}{3}$ wie folgt berechnen:

$$score_i = \sum_{j=0}^2 c_j * \vec{sub}_{i,j}$$

2.4.4 Frequency Compensation

Grundidee des *Frequency Compensation* Moduls ist eine Bevorzugung oft genutzter Tags, gegenüber sehr seltenen. Hierzu beziehen wir die Statistik der in subsection 2.2.2 beschriebenen *tagInfo*-Daten.

Da ein beispielweise zehnfach so oft getaggttes Wort, im Allgemeinen offensichtlich nicht auch zehnfach so relevant ist, verwendeten wir an dieser Stelle anstatt der reinen Multiplikation die Logarithmusfunktion. Die neuen Werte lassen sich mit

$$score'_i = score_i * \log(freq)$$

berechnen. \vec{freq} bezeichnet hierzu den Vektor, der die absoluten Häufigkeiten der zugehörigen Tags beinhaltet. Selbige Konstruktion schließt offensichtlich den Fall aus, den Logarithmus von 0 zu berechnen.

2.4.5 Term weighting

Bis hierher wurden nun also zu einer Eingabe $e - 1$ weitere Wörter abgeleitet, und jeweils gewichtete und anschließend geglättete tf-idf-Vektoren entsprechend berechnet. Als letzten Vorbereitungsschritt sollen die e Vektoren zusammengelegt werden, gewichtet nach der ursprünglichen Relevanz zum Suchwort.

Die Bedeutung der Berechnung sollte intuitiv sein: Jeder der e Vektoren speichert je einen Wert pro Tag ab, der die entsprechenden Relevanz abbildet. Die s_i , die ein Maß der semantischen Ähnlichkeit der Ursprungseingabe q_1 zu allen anderen q_i sind, Werden nun als Gewichtung verwendet. Dadurch werden semantisch ähnliche Suchtreffer höher gewichtet als semantisch suboptimale. Ein Beispiel sollte dies klar machen. Angenommen, die Suche nach $q_1 = waterfall$ wird zu $q_2 = rural_area$ mit einer Ähnlichkeit von $s_2 = 0.3$ expandiert und dessen if-idf-Score des Wertes *residential=rural* wäre 200. Zudem habe q_1 einen unerwartet niedrigen if-idf-Score von 120 beim Tag *waterway=waterfall*, da z.B. die Wiki-Seite unbefriedigend erstellt worden ist. Durch die Gewichtung der Vektoren gemäß semantischer Ähnlichkeit zur ursprünglichen Eingabe, erhalten wir den erwarteten Suchtreffer.

Der finale Scoring-Vektor lässt sich demnach mit

$$\vec{fin} = \sum_{i=1}^e s_i * score'_i$$

berechnen.

2.4.6 top k - result collecting

Die Berechnung in subsection 2.4.5 liefert uns eine Vektor mit der Dimension in der Anzahl der Tags, und repräsentiert die Befriedigung der Eingabequery. Um den Benutzer eine Auswahl über mehrere mögliche passende Tags zu bieten, liefern wir mit Anwendung von *argmax* auf \vec{fin} davon die besten k Ergebnisse zurück.

Zudem liefern wir noch einen Wert im Intervall $(0, 1)$ zurück, der Angibt, wie passend der Tag die Anfrage widerspiegelt. Die Werte wurden dabei mithilfe der Funktion *softmax* adäquat skaliert.

3 Bedienung der Anwendung

Die Folgenden Abschnitte sollen beispielhafte Abläufe darlegen die das Starten des Servers, das Anfragen sowie das anschließende Auswerten der Response erläutern.

3.1 Starten

Zum Starten der Anwendung sind drei Files notwendig, die standardmäßig im selben Verzeichnis liegen müssen: Zuerst ist das Vektor-Modell `GoogleNews-vectors-negative300.bin` von Google notwendig, das auf ihrer Website zum Download angeboten wird [9]. Zudem muss die `tag.json`-Datei vorliegen, die die Inhalte aller OSM-Seiten beinhaltet. Zuletzt muss das eigentlichen Skript `search.py` im Verzeichnis bereit liegen.

Das Ausführen der Anwendung wird mit `python3 ./search.py` angestoßen.

Nun können Anfragen an das System gestellt werden, sobald dieses vollständig hochgefahren ist.

3.2 Aufbau der Anfrage

Eine beispielhafte Anfrage an das System nach dem Term *house*, dem Expansionsgrad *10* (Folglich $e = 9$) und der Rückgabe von anschließend *5* Tag-Ergebnissen sieht wie folgt aus:

```
curl -d '{"query":"house", "amount":5, "nearest_neighbor":10}'  
-H "Content-Type: application/json" -X POST http://localhost:8080
```

3.3 Aufbau der Rückgabe

Die Rückgabe besteht aus einem *json*-String, genauer gesagt eine Liste von Listen, die jeweils einen Tag und der dazugehörige Scoring-Wert liefern.

```
[["Tag:landuse=residential", 0.6396545082270853],  
 ["Tag:building=house", 0.11751306010961371],  
 ["Tag:building=cabin", 0.09408986107578671],  
 ["Tag:building=detached", 0.07558317020115245],  
 ["Tag:building=farm", 0.07315940038636194]]
```


4 Projektreflexion

4.1 Verbesserungsansätze

Nach Abschluss des Projektes möchten wir ein paar Ansätze nennen, an denen man in Zukunft arbeiten könnte, um das System effizienter und schneller zu machen und die Qualität des Rankings zu erhöhen.

4.1.1 Google Word2Vec

Das aktuelle Modell, das wir für das Expandieren der Wörter auf e Terms verwenden, besitzt eine Größe von circa 3.4 Gigabyte. Folglich nimmt das Starten des Servers eine erhebliche Zeit in Anspruch, zudem werden Speicherressourcen verschwendet.

Da in dem eigentlichen tf-idf-Scoring ein Treffer nur als ein solcher gezählt wird, wenn dieser als String innerhalb eines Wiki-Dokumente auftaucht, könnten wir in dem Google-Vektor Modell nur solche Wörter als Eingabe erlauben, die in der Wiki-Seiten vorkommen.

Das spart Platz im RAM, außerdem würde es die Suchzeit erheblich verringern.

4.1.2 Syntaktische um Semantische Suche ergänzen

Wir verwenden eine semantische Erweiterung der Startanfrage auf insgesamt e Suchanfragen. Leider verwendet unser aktuelles System für den eigentlichen Suchalgorithmus lediglich eine syntaktische Suche. Hier entstehen Treffer, die nicht als solche gezählt werden sollten. Beispielsweise wird ein Tag bei der Suche nach *waterfall* mit einem Treffer markiert, wenn es Formulierungen wie ‘... *not to be confounded with waterfall*’ beinhaltet.

Man könnte sich in diesem Zusammenhang mit dem Framework *NLTK* unter Python auseinanderzusetzen. Hier könnte neben der Wiki-Seiten, auch die Query semantisch analysiert werden, sodass gezielt Fragen beantwortet werden könnten

4.1.3 Struktur besser ausnutzen

In Anlehnung an page 9, wird in unserem Modell die interne Struktur der Wiki-Seiten noch nicht ausreichend genutzt. Man könnte noch mehr Semantik aus der Struktur den Seiten entnehmen, indem beispielsweise Tabellen besser eingebunden werden.

Schwierig bleibt jedoch auch hier wieder die Gewichtung der Semantik. Selbiges Problem wird in subsection 4.1.4 diskutiert.

4.1.4 Gewichtung der Content-Teile

In subsection 2.4.3 erläuterten wir das Vorgehen, verschiedene Teile der Wiki-Seiten unabhängig zu indizieren und anschließend den Score geeignet zu kombinieren. Eine begründete Wahl der c_i zu finden, stellt sich als Herausforderung dar. Der Leser möge sich die Frage “Welcher Tag passt am besten zur Eingabe ...?” konstruieren, um herauszufinden, welche Parameterwahl als passend erscheint.

Für mehr als 2 Freiheitsgrade der Parameter wird es sich allerdings als schwierig herausstellen, die Optimalität dieser Wahl zu begründen.

4.1.5 Automatische Eingabekorrektur

Ein Expandieren der Eingabe ist nicht möglich, wenn das Eingabewort nicht Teil des Word2Vec-Modells ist. So ein Fall tritt insbesondere dann auf, wenn die Eingabe durch einen Tippfehler verfälscht ist. Eine automatische Korrektur wäre hier sinnvoll, um beispielsweise *airprt* automatisch zu *airport* zu korrigieren.

4.2 Ergebnisqualität

Dieser Abschnitt erläutert anhand ausgewählter Anfragen die Charakteristika der entwickelten Such-Engine. Dieser Abschnitt kann unter anderem auch Inspiration für weitere Verbesserungen verstanden werden.

4.2.1 gute Ergebnisse

Im folgenden sind Beispiele gegeben, für die eine Händische Suche durch alle Tags ein analoges Ergebnis produziert hätte.

Search: airport	Search: cloths
Tag:aeroway=terminal	Tag:shop=dry_cleaning
Tag:aeroway=apron	Tag:shop=laundry
Tag:aeroway=gate	Tag:shop=clothes
Tag:aeroway=taxiway	Tag:amenity=lavoir
Tag:aeroway=hangar	Tag:shop=fabric
Search: ball	
Tag:sport=table_tennis	
Tag:sport=boules	
Tag:sport=tennis	
Tag:sport=beachvolleyball	
Tag:sport=lacrosse	

4.2.2 überraschende Ergebnisse

Diese Beispiele zeigen schön, dass die ursprüngliche Suche semantisch erweitert wurde. So finden wir auf der einer Seite passende Begriffe wie Bahnhof-Tags, zum anderen auch einen Tag für Restaurants, in Anlehnung an das gleichnamige Unternehmen.

Die zweite Suchanfrage sucht neben der Personenbezeichnung offensichtlich auch deren Wohnumgebung, obwohl in dem OSM-Artikel das Wort *knight* in dem gefunden Tag nicht einmal auftaucht.

Search: subway	Search: knight
Tag:railway=subway_entrance	Tag:castle_type=defensive
Tag:station=subway	
Tag:railway=subway	
Tag:public_transport=platform	
Tag:amenity=fast_food	

4.2.3 unzureichende Ergebnisse

Als Ansatz für zukünftige Optimierungen stellen wir hier noch ein paar Beispiele zur Schau, in die zukünftige Optimierungen einhaken könnten.

Search: bow	
Tag:sport=bandy	0.5166928762734

aber

Search: arrow	
Tag:sport=archery	0.35975876802262186
Tag:natural=ridge	0.1781519400540717
Tag:guidepost=simple	0.17534783227284406
Tag:highway=traffic_signals	0.14598696420532248
Tag:junction=roundabout	0.1407544954451397

Interessanterweise gibt es den Sport *Bogenschießen* als Tag innerhalb der OSM-Seiten. Leider wird bei Eingabe von *bow* nicht die logisch passende Sportart angezeigt. Die Suche nach *Pfeil* hingegen ist erfolgreich.

Search: church	
Tag:tower: type =bell_tower	0.5335585869452344
Tag:denomination=mormon	0.2835628039185601
Tag:landuse=churchyard	0.15180475416031017
Tag:building=cathedral	0.02060323943121495
Tag:amenity=place_of_worship	0.010470615544680439

aber

Search: building chruch	
Tag:building=church	0.3497134817186539
Tag:building=barn	0.19188670466761262
Tag:building=mosque	0.1561445060806261
Tag:building=farm_auxiliary	0.1543794781358878
Tag:building=construction	0.14787582939721966

Auch hier liegt ein ähnliches Problem wie im Beispiel oberhalb vor.

5 Anhang

Verwendete Frameworks

Crawler

- Scrapy (Crawl-Helfer)
- BeautifulSoup (html-Parser)

Suche

- sklearn(tf-idf Gewichtung)
- gensim (Google-model)
- numpy
- nltk (Stopwords)

References

- [1] wikibooks. Tag. <https://wiki.openstreetmap.org/wiki/Tags>. Accessed: 2018-10-08.
- [2] taginfo. building. <https://taginfo.openstreetmap.org/keys/building#values>. Accessed: 2018-10-08.
- [3] taginfo. building. <https://taginfo.openstreetmap.org/tags/highway=service>. Accessed: 2018-10-08.
- [4] OpenStreetMap Foundation (OSMF). Openstreetmap. <https://www.openstreetmap.org/>. Accessed: 2018-10-15.
- [5] The OpenStreetMap Foundation. De:hauptseite. <https://wiki.openstreetmap.org/wiki/DE:Hauptseite>. Accessed: 2018-10-15.
- [6] The OpenStreetMap Foundation. <https://taginfo.openstreetmap.org/>. Accessed: 2018-10-15.
- [7] The OpenStreetMap Foundation. About. <https://taginfo.openstreetmap.org/about>. Accessed: 2018-10-15.
- [8] The OpenStreetMap Foundation. Sitemap. <https://wiki.openstreetmap.org/sitemap-index-wiki.xml>. Accessed: 2018-10-17.
- [9] team of researchers led by Tomas Mikolov at Google. Word2vec. <https://code.google.com/archive/p/word2vec/>. Accessed: 2018-10-18.