Entwicklung einer Linked-Data Anwendung auf Basis des ZDF-Lobbyradars

Eine Projektarbeit im Rahmen der Lehrveranstaltung

Wissensrepäsentation

HTW Berlin, Angewandte Informatik

Dozent: Prof. Dr. Christian Herta

Thade Feddersen s0542833

Ido Sternberg s0540151

Gliederung

Einführung

In diesem Schreiben möchten wir die schrittweise Entwicklung einer Semantic-Web Anwendung, die mit den öffentlichen Daten des Lobbyradars arbeitet, beschreiben. Mittels dieser Anwendung möchten wir es erleichtern, nützliche Informationen aus den Daten zu generieren und diese übersichtlich darzustellen.

Zielstellung

Als Ziele dieses Projekts wurde folgendes vorgeschrieben:

- Verknüpfen von verschiedenen Datenquellen
- Konvertierung von Daten ins RDF-Format (mit entsprechender T-Box)
- Entwicklung von Ontologien
- Linked Data Anwendung
- Informationsextraktion und Konvertierung in RDF
- Entity Resolution

Über den ZDF Lobbyradar

Der Lobbyradar (www.lobbyradar.de) ist eine Datenbankvisualisierung, die von der ZDF heute.de, dem Medieninnovationszentrum Babelsberg MIZ und dem OpenDataCity bereitgestellt wurde. Ziel des Lobbyradars ist, nach eigener Aussage, die Verbindungen zwischen Politik und Interessenvertern darzustellen und den Lobbyismus in Deutschland transparenter zu machen. Konkreter werden Entitäten wie Personen, Parteien, Firmen, Verbände, PR-Agenturen und NGOs und ihre Beziehungen zueinander erfasst und als Graphen dargestellt. Der Lobbyradar will aber keine vollständige Aussagekraft über den Lobbyismus in Deutschland darstellen. Es werden nämlich in der Datenbank keine Freundschaften, frühere Mitgliedschaften oder Arbeits- bzw. Verwandschaftsverhältnisse erfasst. Ferner stammen alle Informationen aus öffentlich zugänglichen Datenbanken und Dokumenten, sodass Personen und Organisationen, die nicht in den Medien auftauchen, nicht abgebildet werden.

Genauere Informationen zu der Herrkunft der Daten wurde uns durch das Lobbyradar Team per Email mitgeteilt:

“*Es gibt zwei Wege, wie wir Daten aggregieren. Über Scraper, die die Daten direkt abgreifen auf bundestag.de, aus der Lobbyliste oder der Lobbypedia. Oder über Excel-Tabellen, die wir händisch erstellen oder bearbeiten (Nebentätigkeiten, Spenden, Länderkabinette etc.)”*

Die Datenbank des Lobyradars wird ständig aktualisiert und enthält zu dem Zeitpunkt ca. 20.000 Personen, 6.000 Organisationen und 30.000 Verbindungen.
Ein wichtiges Feature des Lobbyradar ist der Browser-Plugin. Dieser überprüft die Inhalte der besucheten Seiten und vergleicht diese mit den Einträgen in der Datenbank. Treffer werden markiert und über einen Klick kann man weitere Informationen aus dem Lobbyradar zu diesen bekommen.

Kritik an das Projekt

Der Unternehemens- und Lobbyistenkommunikationsberater Christof Fischoeder hat am 03.06.2015 im PRReport den Lobbyrader aufgrund folgender Argumenten als platt und pseudokritisch bezeichent:

- Durch die händische, verfizierunglose Recherche seien die Informationen in der Datenbank unvollkommen und enthalten fehlerhaften Quellenverweisen.
- Die Vernetzung von Akteuren in der Politik sei an sich nicht problematisch, werde aber durch das Lobbyradar so angegeben.
- Es fehlten, neben der quantitativen Quellenangabe, die Rollen welche Person oder Institution am politischen Prozess teilnimmt.

Der Journalist Lorenz Matzat bezeichente den Lobbyradar am 06.60.2015 im “Datenjournalist” als kaum zu gebrauchen. Es fehlen seiner Meinung nach folgende Funktionen:

- die Möglichkeit, die relevanten von den nicht relevanten Daten zu filtern, z.B.: alle Mitgliedschaften auszublenden
- die Möglichkeit, die Daten nach Thema zu gruppieren, z.B.: nur Wirtschaftsverbände anzuzeigen
- unterschiedliche Einfärbung der unterschiedlichen Knotenarten

Auf Matzat’s Kritik hat die ZDF-Lobbyradar in den Artikelkommentaren reagiert: “Nehmen wir uns zu Herzen! Filter sind schon in Planung.”

Eine weitere Kritik, die man mit Vorsicht geniessen sollte, wurde am 10.05.2015 von Peter Harth an der “Kopp Online” geäußert. Der Kopp-Verlag ist für Veröffentlichungen im Bereich der Verschwörungstheorien und Pseudowissenschaft bekannt.

Harth bezeichnet das Lobbyradar als ”Nichts Brauchbares” und kritisiert es dafür, dass die ZDF-Journalisten, die selber Lobbyisten seien, nicht in der Datenbank auftauchen. Ein weiterer Punkt ist es, dass die Verbindungen in der Datenbank untereinander nicht bewertet oder gewichtet werden, weshalb das Lobbyradar nicht mehr als eine Masse von Daten sei.

Lobbyradar - Technische Impelentierung
Das Projekt wurde mittels eine Mongo-Datenbank realisiert. Diese ist, im gegensatz zu einer relationalen Datenbank, schemafrei und Dokumentorientiert, und hat bei der Abbildung von Beziehungen zwischen Entitäten einen großen Nachteil, denn dokumentorientierte Datenbanken passen vor allem zu automaren Datensätzen, die miteinander nicht verbunden sind. Das Abfragen nach Relationen wäre mit einer relationalen Datenbank wesentlich einfacher gewesen.

Ziel des Projektes

Durch die Realisierung des Lobbyradars mittels einer MongoDB is die Generierung von Wissen aus den vorhandenen Daten sehr umständlich. Unser Ziel war es die Daten so umzuformatieren, dass man aus ihnen leicht Wert schöpfen kann. Dazu haben wir eine Anwendung entworfen, die Nutzern eine einfache Oberfläche zur Analyse und Erforschung des Lobbyradas zu Verfügung stellt. Als wesentlicher Schritt für die Umsetzung unseres Projektes müssten die Daten zunächst in RDF konvertiert werden.

Der Arbeitsprozess

In dem folgenden Abschnitt werden wir die verschiedenen Phasen bei der Entwicklung der Anwendung beschreiben und ihre Resultate präsentieren. Der Prozess lässt sich mit folgenden Schritten beschreiben:

- **Explorative Datenanalyse**
- **Ziel-orientierte Datenanalyse**
- **Entwurf einer Ontologie**
- **Konvertierung von Daten ins RDF-Format**
- **Entwurf der Anwendung**
- **Implementierung**

Explorative Datenanalyse

Eine Explorative Datenanalyse ist zwar nicht vorgeschrieben, stellt aber einen wesentlichen Teil jedes Projekts dar, bei dem es sich um Arbeit mit Daten handelt. Ziele der Analyse sind vor allem, die Daten anzuschauen, ihre Bedeutung und Kontext nachzuvollziehen und fehlerhafte sowie irrelavante Daten herauszusortieren.

Durch Anzeige aller Relations-Typen werden folgende Eigenschaften der bereitgestellten Daten deutlich:

- Die Sprache der erfassten Daten ist nicht konsistent, z.B.: bei "subsidiary" und "Tochterfirma", wobei beide Begriffe die gleiche Bedeuting haben.
- Manche Daten sind mit Schreib- bzw. Typfehler erfasst, z.b.: "ececutive" und "subisdiary". Solche und andere Arten von Fehler kommen in der Datenbank leider nicht selten vor.
- In manchen Fällen ist es nicht vollkommen klar, was unter die Beziehungstyp zu verstehen ist und es Bedarf weitere Recherche, z.B.: bei "publication" und "Hausausweise".

```
In [56]: p.pprint(db.relations.distinct('type'))
```

```
[Bundesdatenschutzbeauftragte,
 Hausausweise,
 Mitglied,
 Position,
 Tochterfirma,
 Vorsitzender,
 activity,
 association,
 business,
 committee,
 consulting,
 donation,
 ececutive,
 executive,
 government,
 lobbyist,
 member,
 mitglied,
 position,
 publication,
 sponsoring,
 subisdiary,
 subsidiary]
```

Die Typen der Relationen können beim Bauen von Ontologien eine wichtige Rolle spielen. Sinnvoll wäre herauszufinden, ob es Typen gibt, die sehr selten vorkommen und eventuell beim Kreieren der

Hierarchie nicht berücksichtigt, oder mit anderen Typen vereinigt werden sollten:

```
In [22]: types = list(db.relations.aggregate(pipeline, cursor={}))
         for typ in types:
             if typ["counter"] < 5:
                 p.pprint(typ)
```

```
{_id: publication, counter: 1}
{_id: ececutive, counter: 2}
{_id: lobbyist, counter: 1}
{_id: Bundesdatenschutzbeauftragte, counter: 1}
{_id: sponsoring, counter: 3}
{_id: Tochterfirma, counter: 4}
{_id: Vorsitzender, counter: 1}
{_id: subisdiary, counter: 2}
```

Um die Aussagekraft der entity-Tags lässt sich durch das Abfragen nach bekannten personen prüfen:

```
In [95]: query = {"name":"Angela Merkel"}
         projection = {"_id":0, "tags":1}
         display(query, projection)
```

```
{tags: [kabinette,
          kabinett-merkel-i,
          verwaltung,
          politik,
          kabinett-merkel-ii,
          kabinett-merkel-iii,
          mdb,
          bundestag,
          parteispenden,
          bundeskanzlerin]}
```

Merkwürdigerweise hat Frau Merkel keinen CDU-Tag.

In vielen Fällen liefern die Queries sehr vielen Resultaten, sodass diese ohne weiteres nicht überschaubar sind.

Ferner wird beim Arbeiten mit den Daten deutlich, dass die Analyse nicht nur Kenntnisse in Data-Mining bedarf, sondern auch Wissen über die deutsche Politik und Wirtschaft.
Eine umfangreiche Datenanalyse wäre userer Meinung nach vorteilhaft, wird aber im Rahmen dieses Projekt nicht gemacht, da andere Schwerpunkte vorgeschrieben sind.

Ziel-orientierte Datenanalyse

In diesem Teil werden die Relationen zwischen den Entitäten untersucht, um den Kontext fürs Bauen der Ontologien zu verstehen.

In dem Skript “relation_type_analysis” wird über alle Relationen eines gegebenen Typs iteriert, um folgende Outputs zu generieren:

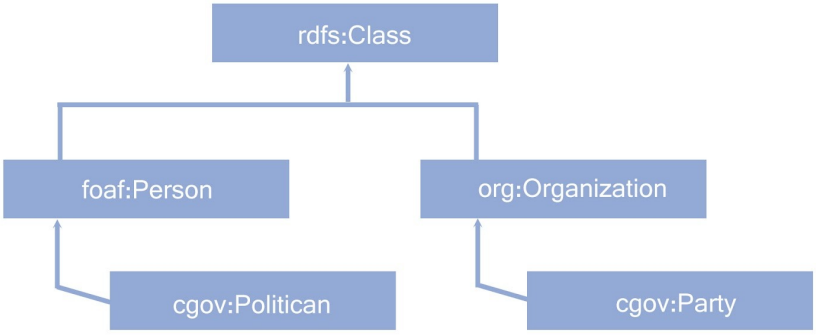
- Wer (Person bzw. Organisation) hat eine Relation zu wem (Organisation)
- Statistiken zu der Anzahl der jeweiligen Relationstypen

Folgender Bild zeigt die ersten Relationen des Typs ‘member’:

```
In [4]: show_relations_of_type('member', 4)
        Katrin Albsteiger (person) has relation member to CDU/CSU-Fraktion (entity)


        Stephan Albani (person) has relation member to CDU/CSU-Fraktion (entity)


        Kerstin Andreae (person) has relation member to Bundestagsfraktion der Grünen (entity)


        Peter Altmaier (person) has relation member to CDU/CSU-Fraktion (entity)


        statisics for relation type: member
        count relation: 4287
                 person to organization: 4098
                 organization to organization: 118
                 errors: 50
```

Entwurf einer Ontologie

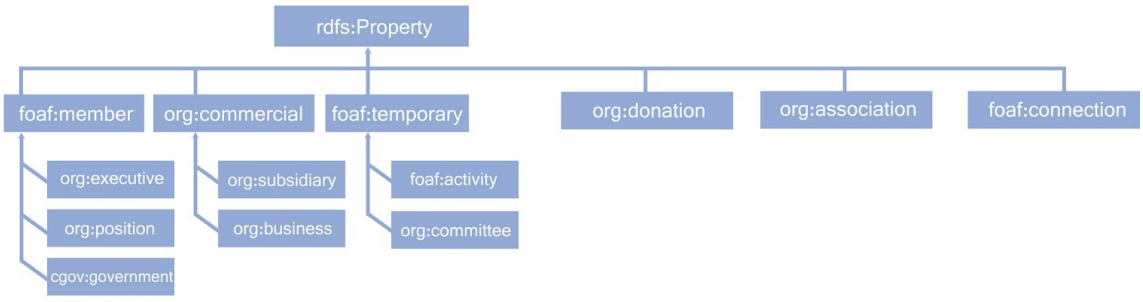
Bis zu diesem Punkt wurden die Daten des Lobbyradars analysiert und zusammengefasst dargestellt. Nun möchten wir untersuchen, ob die vorhandenen Daten in Hierarchien zueinander stehen und ob sie zu bereits existierten RDF- bzw. RDFS-Namespaces zugeordnet werden können. Folgende Grafiken zeigen die von uns entwickelten Hierarchien:



foaf: friend of a friend (http://xmlns.com/foaf/0.1/)

org: organization (http://www.w3.org/ns/org#)

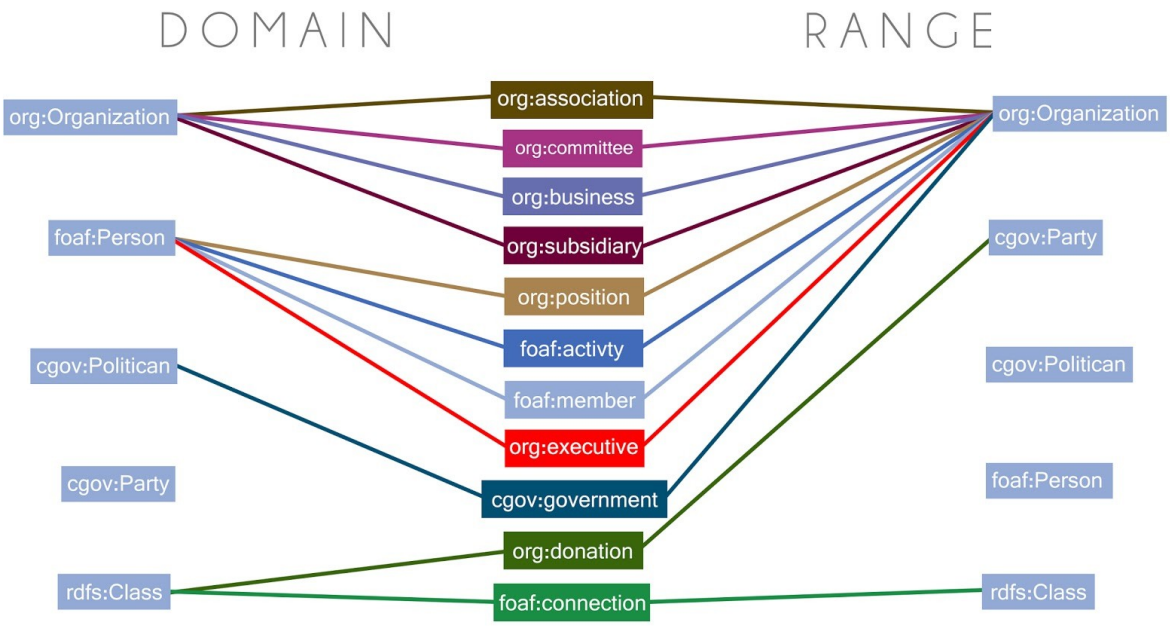
cgov: government (http://reference.data.gov.uk/def/central-government/)



foaf: friend of a friend (http://xmlns.com/foaf/0.1/)

org: organization (http://www.w3.org/ns/org#)

cgov: government (http://reference.data.gov.uk/def/central-government/)



Anhand von diesen Ontologien haben wir den Turtle-Datei “ontologie.ttl” geschrieben.

Konvertierung von Daten ins RDF-Format

Die Konvertierung der Lobbyradar-Daten von der MongoDB in einen RDF-Store kann in zwei wesentliche Schritte unterteilt werden.

1. Eintrag aller Entitäten als RDF-Nodes
2. Eintrag aller Verbindungen zwischen diesen Entitäten

Die Konvertierung der Lobbyradar-Entitäten ist sehr simple. Auch das MongoDB-Format von Kollektionen und abgeschlossenen Datensätzen ist hier von Vorteil.

```
for entity in Entities.find({}):
    node = BNode()

    g.add((node, DC.identifier, Literal(entity["_id"])))
    g.add((node, RDF.type, rdf_type[entity['type']]))
    g.add((node, RDFS.label, Literal(entity["name"])))
```

Es wird über die Kollektion “Entities” iteriert. Für jede Entität wird ein Blanknode angelegt.

Da, wie bereits erwähnt, der Hauptfokus auf den Relationen zwischen Entitäten liegt, werden nur die wichtigsten Daten einer Entität gespeichert. Dabei handelt es sich um seine ID und seinen Namen. Die ID ist, im Gegensatz zu dessen Namen, garantiert einzigartig.
Außerdem wird die Entität bereits hier als Person oder Organisation klassifiziert.
Nach Abschluss dieser Routine ist gewährleistet, dass alle Entitäten des Lobbyradar (Personen wie Organisationen) im RDF-Graphen vorhanden sind.

Das Verbinden dieser Einträge arbeitet relativ analog dazu.

```
for relation in Relations.find({}):

    if len(relation['entities']) < 2: continue;

    source = g.value(predicate=DC.identifier, object=Literal(str(relation['entities'][0])))
    target = g.value(predicate=DC.identifier, object=Literal(str(relation['entities'][1])))

    source_type = g.value(subject=source, predicate=RDF.type)
    target_type = g.value(subject=target, predicate=RDF.type)

    source_name = g.value(subject=source, predicate=RDFS.label)
    target_name = g.value(subject=target, predicate=RDFS.label)

    if not source or not target: continue
    if source_type == ORG.Organization and target_type == FOAF.Person:
        source, target = target, source
    prop = get_prop(relation['type'])
    make_special_deklaration(get_property_key(relation['type']), source, target, source_name, target_name)
    if(prop):
        g.add((source, prop, target))
    else:
        print(relation['type'])
```

Wieder wird über eine Kollektion iteriert. In diesem Fall über die Kollektion “Relations”, welche eine Beziehungen zwischen zwei Entitäten als ein Eintrag speichert.

Die zwei wichtigsten Daten eines solchen Eintrags sind die IDs der beteiligten Entitäten und der Typ der Beziehung. Über die IDs werden die Entitäten rausgesucht, die, nach der Konvertierung der Entitäten, bereits im Graphen vorhanden sind. Mit dem Typ der Relation wird entschieden, welche Property zwischen den beiden Entitäten gesetzt werden soll.

Die, den Entitäten, bisher zugewiesenen Typen sind sehr allgemein (Personen, Organisation).
Das liegt daran, dass die MongoDB-Einträge der Entitäten nur über zwei Typen verfügen. Mittels der Relationen zwischen zwei Entitäten lassen sich allerdings weitere, speziellere Typen rausfinden.
So wurde in der Ziel-orientierten Datenanalyse aufgedeckt, dass das Objekt einer Relation von Typ “donation” stets eine Partei ist, oder das Subjekt einer Relation vom Typ “goverment” stets ein Politiker/Politikerin.
Die Funktion “make_special_declaration” fügt mittels des Relations-Types gegebenenfalls weitere Typen zu einer Entität.

Entwurf der Anwendung

Unsere Anwendung soll ein Platform für Nutzer sein, die Daten das Lobbyradars durchsuchen zu können. An dieser Stelle fragen wir uns, welche Anforderung Nutzer des Lobbyradar an diese Anwendung haben wollen und welches Wissen für sie interessant sein kann.
Wir entscheiden uns für die Arbeit mit User-Stories, um die Anforderungen zu erfassen.
Die hier aufgenanten User-Stories sind zunächst einmal so formuliert, wie sie den Anforderungen entsprächen. Es wird dabei noch keinen Bezug auf die Realisierbarkeit genommen.

As a user I want to filter the Entities according to their type
As a user I want to see all the direct relations of any entity
As a user I want to see indirect relations of an entity
As a user I want to input two Entities and see if they are directly or indirectly related
As a user I want to see the top donors to a given party
As a user I want to see which organizations donate to more than one party
As a user I want to see who the politicians of a specific party are
As a user I want to see the infrastructure of of a given party

Implementierung

Wie bereits im Entwurf dargelegt, soll die Anwendung eine Plattform sein, mit der, Benutzer interagieren können, um den Lobbyradar-Datensatz zu endecken.

Die Anwendung selbst arbeitet auf der Kommandozeile und kommt auch in ihrer Benutzung einer Kommandozeile nah.

Der Benutzer wird gebeten einen Befehl und gegebenenfalls auch Parameter einzugeben. Ein Befehl entspricht in diesem Fall einer Query und die Parameter, der Variablen der Query.
Der vom Benutzer eingegebene String wird mittels eines regulären Ausdrucks verarbeitet.
Nach dieser Verarbeitung sollten Befehl und Parameter als einzelne Elemente in einer Liste verfügbar sein.
Mittels einer Map wird rausgefunden, ob der eingegebene Befehl existiert. In dieser Map können alle verfügbaren Befehle angegeben werden.
Ist der Befehl vorhanden, wird die dazugehörige Funktion mit den Parametern aufgerufen.
Jeder Befehl speichert sein Ergebnis in einer globalen Variable. Nach Abschluss der Query kann der Benutzer das Ergebnis entweder als Graph oder als Liste in der Kommandozeile ausgeben.

Hier ein Beispiel einer Query

```
def network_subject(entity, graph = g):
    query = """
        SELECT ?entity_name ?related_name ?relation
        WHERE {
              ?entity_node rdfs:label "%s" .
              ?entity_node rdfs:label ?entity_name .
              ?entity_node ?relation ?related_node .
              ?related_node rdfs:label ?related_name .
        }
        """ % entity
    query_result = graph.query(query)
    result = []
    for a,b,c in query_result:
        result.append((a.value, b.value, c))
    return result
```

Die Aufgabe dieser Funktion ist es, das egozentrische Netzwerk einer Entität zu finden.

Dabei soll der Klartext-Name der Entität als Paramter übergeben werden. Als weiterer Parameter wird außerdem der Graph mitgegeben, auf welchem die SPARQL-Anfrage abgesetzt wird, diese ist allerdings bereits mit einer globalen Standardvariable belegt und kann beim Aufruf ignoriert werden.
Die Funktion setzt sich aus zwei Teilen zusammen. Zum einen die SPARQL-Abfrage und zum anderen die nachträgliche Umkonventierung zu einer Liste mit Tupel.
An dieser Stelle würde es zu weit führen, SPARQL-Anfragen und ihre Syntax als Solche zu erklären.

Für uns genügt, dass es sich dabei um einen String handelt. Diesen setzten wir als Query auf dem Graphen ab.
Eine SPARQL-Query gibt eine SPARQL-Resultset-Instanz zurück. Da manche Queries sich aus grundlegenderen Queries zusammensetzen oder eine Nachbearbeitung der Tupel von Nöten ist, wird das Ergebnis noch Mal in eine Liste umgecastet. Damit ist mehr Kontrolle über die Ergebnismenge als Solche und in Verbindung mit weiteren Ergebnismengen gewährleistet.
Die Tupel einer Ergebnisliste beinhalten immer drei Elemente. Bei diesen handelt es sich um die Akteure eines RDF-Datensatzes, also Subjekt, Prädikat und Objekt.
Es ist allerdings wichtig anzumerken, dass die Reihenfolge bei den Tupeln der Ergebisliste Subjekt, Objekt, Prädikat ist. Diese wurde gewählt, um später beim Anzeigen der Liste als Graph weniger Programmieraufwand zu haben.

Es ist grundsätzlich möglich eigene Funktionen beliebiger Komplexität zu schreiben, solange sie eine Ergebnisliste, in der oben genannten Struktur zurückgeben. Im Rahmen der Projektarbeit wurden bereits einige Funktionen hinzugefügt. Dabei handelt es sich um grundlegende Funktionen, die dem Benutzer einen schnellen Einstieg ermöglichen sollen.

find_person/find_politician/find_organization/find_party/find_entity

Das Lobbyradar ist ein relativ großer Datensatz mit vielen Entitäten. Da es in erste Linie interessant für den Benutzer ist, überhaupt erstmal rauszufinden, ob eine bestimmte Entität im Graphen existiert, werden diese Such-Funktionen zur Verfügung gestellt.
Diese Suche verläuft über Freitext d.h. der Benutzer kann einen beliebigen String eingeben. Danach werden alle Entitäten auf diesen String überprüft. Selbst wenn der String nur als Teil eines größeren String vorkommt, gilt es als Treffer. Groß- oder Kleinschreibung wird ignoriert. So kann der Benutzer nicht nur rausfinden, ob die Entität überhaupt existiert, sondern auch mit welchem vollständigen Namen sie im Graphen aufzurufen ist.
Die Such-Funktionen arbeiten im Kern alle gleich, je nach Funktion kann lediglich die Suchmenge eingeschränkt werden. Wer alle Entitäten durchsuchen möchte (weil er den Typ des Gesuchten zum Beispiel nicht kennt) kann die Funktion find_entity verwenden. Sie sucht über alle Entitäten.
Alle Such-Funktionen als Parameter einen Entitätennamen.

network

Das Funktion für das egozentrische Netzwerk einer Entität wurde bereits teilweise mit dem Beispiel-Screenshot gezeigt. Es geht darum, alle direkt mit der gegebenen Entität verbundenen Entitäten zu finden. Hierzu sollte angemerkt werden, dass Relationen in RDF unidirektional sind d.h. vom Subjekt zum Objekt verlaufen, nicht umgekehrt. Um also alle verbundenen Entitäten zu finden, muss die eingegebene Entität einmal als Subjekt und einmal als Objekt behandelt werden. Der Screenshot zeigt die Funktion, in der die Entität als Subjekt behandelt wird. Gefunden werden soll jedwedige Verbindung im Graphen, in der die Entität Subjekt eines Datensatzes ist. Parallel dazu gibt es die network_object-

Funktion. In dieser verhält es sich genau andersrum. Die network-Funktion ist das Zusammenfügen beider Ergebnisse.
Zusätzlich ist es bei dieser Funktion möglich, die Tiefe anzugeben d.h. welches Level des egozentrischen Netzwerkes angezeigt werden soll. Das wird iterativ erreicht. Zuerst wird die network-Funktion auf die eingegebene Entität angewendet und auf seine Nachbarn und dann auf deren und so weiter, bis das gewünschte Level erreicht ist. Ganz nach dem “Smal World Principle” sind Tiefen über zwei mit Vorsicht zu genießen, da diese bereits ernorm groß sind.

connection

Eine weitere wichtige Information für den Benutzer kann es sein, rauszufinden ob und wenn ja, wie zwei Entitäten verbunden sind. Dieses soll die Funktion connection aufzeigen. Für die Suche kommt eine graphbasierende Breitensuche zum Einsatz. Diese Arbeitet zusammen mit der network-Funktion. Als Eingabe werden zwei Entitäten erwartet. Als erstes werden die direkten Nachbaren von Entität1 rausgesucht und überprüft, ob die gesuchte Entität2 unter ihnen ist. Sollte das nicht der Fall sein, werden die Nachbarn selbst der Reihne nach auf ihre Nachbarn überprüft und ob unter diesen die Entität2 zu finden ist. Aufgrund der dichten Vernetzung des Graphen wurde darauf verzichtet, eine unendliche tiefe Suche zuzulassen. Die Tiefe muss manuell als dritter Parameter angegeben werden, ohne ihn wird von der Tiefe 1, also nur die direkten Nachbarn, ausgegangen. Sollte also keine Verbindung bei einer bestimmten Tiefe gefunden werden, ist es durchaus möglich, das eine solche in einer höheren Tiefe vorhanden ist. Doch auch hier gilt, dass jede Tiefe über zwei lange Rechenzeit bedeuten kann.

```python
def connection(entity, target, max_level = 1, graph = g):
    queue = []
    path = []
    network = network_entity(entity)
    queue.extend([Node(o, 0, Node(s, -1, None, None), p) for (s,o,p) in network])
    for node in queue:
        if(node.name == target):
            current_node = node
            while(current_node.parent.name != entity):
                path.append((current_node.name, current_node.parent.name, current_node.con))
                current_node = current_node.parent
            path.append((current_node.name, current_node.parent.name, current_node.con))
            return path
        if(node.lvl >= max_level):
            return []
        element_network = network_entity(node.name.encode('utf-8'))
        for s,o,p in element_network:
            if(not is_in_node_list(o, queue)):
                queue.append(Node(o , node.lvl + 1, node, p))

def is_in_node_list(name, node_list):
    for element in node_list:
        if(name == element.name):
            return True
    return False

class Node:
    def __init__(self, name, lvl, parent, con):
        self.name = name
        self.lvl = lvl
        self.parent = parent
        self.con = con
```

Auszug aus query.py

Gemäß eines Graphen-Such-Algorithmus werden bereits überprüfte Node makiert.

doner_of/donated

Da es sich bei dem Lobbyradar hauptsächlich um eine Datenbank über Lobbyisten handelt, ist es für den Benutzer von Interesse rauszufinden, welche Parteien von wem Spenden erhalten haben bzw. welche Organisationen an welche Parteien gespenden haben.
Diese beiden Aufgaben werden von den Funktionen doner_of und donated übernommen. Beide erwarten einen Parameter, welcher im Falle von doner_of eine Partei und im Falle von donated eine Organisation ist. Beide Funktionen sind in ihrer Vorgehensweise sehr simple. Es wird nach Datensätzen mit der Property “donation” mit der Partei als Objekt (doner_of) oder der Organisation im Subjekt (donated) gesucht.

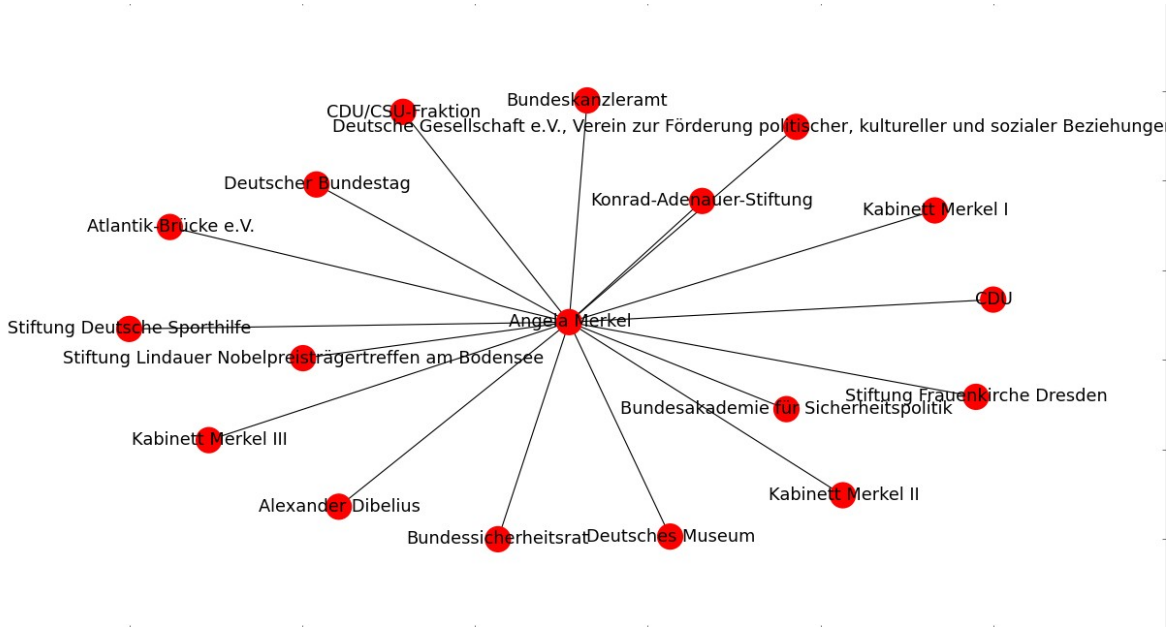
Hier sind noch weitere Befehlen, bei denen es sich nicht um Queries handelt d.h. sie geben kein Ergebnis zurück, sondern sind vielmehr Grundfunktionen für die Platform selbst.

help

Zeigt die Hilfe für die Anwendung. Man sieht eine Übersicht über alle verfügbaren Befehle und wie man sie nutzt.

plot

Mit dieser Funktion wird das letzte Ergebnis als Graph gerendert.
Dazu kommt das Python Modul networkx zum Einsatz. Jedes Ergebnis ist eine Liste bestehend aus Tupeln. Diese Tupel enthalten drei Element, welche dem Subjekt, Objekt und ihrer Verbindung entsprechen. In der Funktion plot werden alle Knoten und ihre Verbindung dem graph hinzugefügt und dann gerendert. Wichtig hier: Der eben genannte Graph ist nicht gleichzusetzt mit dem sonst genannten Graph. Bei letzterm handelt es sich um den RDF-Graphen, welcher als RDF-Store alle Knoten und Verbindungen enthält. Bei ersterem um eine Instanz von Typ Graph wie sie networkx zur Verfügung stellt, welcher nur einen Teilgraph des RDF-Graphen anzeigt.



Das Ergebnis des Befehlts network “Angela Merkel” wird mit der plot-Funktion gerendert

list

Mit der List-Funktion wird das letzte Ergebnis einfach direkt als Liste auf der Kommandozeile ausgegeben. Das ist dann hilfreich, wenn man den Namen einer Entität kopieren möchte. In manchen Fällen kann die aufgegebene Liste sogar übersichtlicher sein als ein Graph. Das gilt vor allem bei großen Ergebnismengen

quit

Mit der Quit-Funktion kann die Anwendung beendet werden.

Fazit

Leider ist die Kovertierung der Relationen nicht so reibungslos, wie das Hinzufügen der Entiäten. Das liegt im Wesentlichen daran, dass es fehlerhafte Relations-Einträge geben kann.
So kann es zum Beispiel vorkommen, dass das Feld “entites”, welches normalerweise eine Liste mit zwei Einträge ist (die IDs der beteiligten Entitäten) nur ein Element, keins oder gar mehrere enthält. Da das Vorhandensein von zwei Entitäten für den Aufbau einer Relation essenziell ist, wurden alle Einträge mit einem oder keinem Eintrag ignoriert. Sind mehrere Einträge vorhanden, wurden nur die ersten Beiden genutzt.
Doch selbst wenn zwei Entiäten ausmachbar sind, müssen diese nicht automatisch vorhanden sein. Das hängt damit zusammen, dass das Lobbyradar unablässig erweitert wird und somit gelegentlich Einträge an Aktualität verlieren d.h. auf Entitäten verweisen, die nicht länger existieren.

Der Typ einer Relation ist ausschlaggebend für das Prädikat, welche zwischen zwei Entitäten erstellt wird. Leider lässt sich der Typ der Relation nicht 1:1 auf die Prädikat überführen, da es mehrere Relationen mit falsch geschriebenen Bezeichnern oder sogar anderen Bezeichnern mit selber Bedeutung gibt (zum Beispiel in einer anderen Sprache).
Das Problem wurde mit einer Map gelöst, die von allen unterschiedlichen Bezeichnung selber Art auf das jeweilige Prädikat zeigt.

Bei dem Hinzufügen speziellerer Typen (Funktion make_special_declaration, import.py) kommt es ebensfalls zu Problemen durch fehlerhafte Relations-Einträgen. Das bereits erwähnte Feld “entites”, welches die IDs der beteilten Entitäten enthält ist eine Liste mit (meist) zwei Einträgen. Daraus geht allerdings noch nicht hervor, welcher dieser beiden Einträgen das Subjekt und welcher das Objekt ist. Im Zuge der Entwicklung wurde eine Konvention aufgestellt: Der erste Einträg ist stets das Subjekt und der zweite stets das Objekt. Diese Konvention funktioniert solange gut, wie auch die Lobbyradar-Daten konsistent bleiben. Leider gibt es einige wenige Relationen mit vertauschten Entitäten.
Über die Ziel-orientierte Datenanalyse wurde rausgefunden, dass das Objekt einer Relation vom Typ “donation” immer eine Partei ist.
Das hat zur Folge, das Spender (meist Organisationen aber auch Personen) fälschlicherweise als Parteien klassifiziert werden.
Dieses Problem wurde ebenfalls über eine Map gelöst. Jedes Objekt einer Relation vom Typ “donation” wird als potenzielle Partei anerkannt. Den Typ Partei bekommt die Entität allerdings erst wenn sie mehrere Male als Objekt vorkommt. Fehlerhafte Einträge sind relativ selten und das Objekt (in diesem Fall eine Organisation oder eine Person) wird sich nur ein, zwei Mal als Partei vorschlagen. Eine

wirkliche Partei hingegen bekommt (meist) viele Spenden und wird sich dadurch öfters als Partei vorschlagen. Nach mehreren Versuchen hat sich ein Schwellwert raus kristallisiert.

Abgesehen von der allgemeinen Möglichkeit, die Anwendung beliebig mit weiteren Befehlen zu erweitern, gibt es einige Punkte, die doch sehr wünschenswert wären.

Die bisherige Klassifizierung von Politikern ist unzureichend. Derzeit wird ein Politiker erkannt, wenn ein Datensatz mit dem Relations-Typ “goverment” auftritt. Durch die Ziel-orientierte Datenanalyse haben wir rausgefunden, dass das Subjekt einer solchen Relation stets ein Politiker ist. Die Konklusion tritt druchaus zu, dennoch handelt es sich dabei lediglich um Politiker, die ein Amt in einer öffentlichen Einrichtung ausüben. Damit werden Politiker kleinerer Parteien, die keine oder kaum Regierungs- oder Parlamentssitze haben übersehen. Hier wäre eine genauere Analyse des Datensatzes vorzuziehen, welche im Rahmen des Projektes leider nicht vollzogen werden konnte.

Für die Klassifizierung von Parteinen gilt ähnliches. Bisher wird eine Partei als Objekt eines Datensatzes mit der Relation vom Typ “donation” erkannt. Hier lässt sich bemerken, dass Parteien ohne Spender auch nicht als Parteien erkannt werden. Das ist, im Kontext des Lobbyradars, allerdings ein nicht ganz so großes Problem, da der Datensatz des Lobbyradars, sich auf Lobbyarbeiten bezieht und damit nur Parteien vorkommen, die auch Spenden erhalten haben.

Das Problem ist vielmehr, dass Parteien nicht zwangläufig nur eine Entität haben. So hat die Partei CDU zunächst einmal die Entität “CDU”, diese wird vom System als Partei erkannt, da auch alle Spenden mit dieser Entität verbunden sind. Doch die Mitglieder der Partei sind alle mit einer anderen Entität verbunden, der Entität “CDU/CSU-Fraktion”. Es wäre wünschenswert auch diese Entität als Partei zu erkennen. Das gilt insbesondere für das Finden aller Mitglieder einer Partei. Denn hier kommt es leider zu Inkonsistenzen. Die Mitglieder der Partei CDU sind wie bereits erwähnt mit der Entität “CDU/CSU-Fraktion” verbunden, während die Mitglieder der Partei CSU mit der, von uns richtig erkannten Partei “CSU” verbunden sind.
Hierzu ist wichtig anzumerken, dass die Entitäten und von uns erkannten Parteien CDU und CSU beide eine Verbindung zu CDU/CSU-Fraktion haben, daher ist es durchaus vorteilhaft, dass nicht die Mitglieder beider Parteien mit einer Partei verbunden sind. Doch es wäre noch einfacher gewesen, wenn die Mitglieder der Partei CDU auch mit der von uns erkannten Partei CDU verbunden wären.

Desweiteren ist es natürlich auch nicht undenkbar die Klassen-Ontologie zu erweitern und so weitere Unterscheidungen zu machen. Da die Parteien richtigerweise über die Relation vom Typ “donation” als Objekt gefunden wurden, ist es auch durchaus möglich, das Subjekt als Spender zu klassifizieren. Selbiges gilt auch für die Propertiy-Hierachie.

Die Benutzbarkeit der Anwendung ist ebenfalls ausbesserbar. Abgesehen davon, dass die gesamte Anwendung auch mit einer grafischen Oberfläche angeboten werden könnte, ist es bei der derzeitigen Konsolenversion problematisch, dass alle Wörter richtig geschrieben werden müssen. Das gilt für die

Befehle wie auch für die Entitäten Namen. Schöner wäre es wenn die Befehle, wie in einer Shell übrig eine Auto-Vervollständigung mitbrächten. Auch eine Historie, welche mit Pfeiltaste nach oben die letzten Befehle anzeigt wäre denkbar. Bei den Namen der Entität ist es vorallem unschön, dass der gesamte Entitäten-Name vollständig und mit all seinen Zeichen richtig eingegeben werden muss. Hier wäre es angenehmer, wenn die eingegebenen Strings noch einmal auf die Such-Funktionen angewandt werden, um den Benutzer dann mögliche Treffer auswählen zu lassen, sodass dieser lange oder komplizierte Namen nicht manuell eingeben muss.

Das Rendern von einem Ergebnis als Graph ist bei großen Ergebnismengen problematisch. Hier könnte ein anderes Werkzeug in Betracht gezogen werden.

Verweise

ZDF Lobbyradar. Online in Internet: https://www.lobbyradar.de (Stand 09.08.2015)

Fischoeder, C. Dieser platte Ansatz hilft der Transparenz-Debatte nicht.
PRREPORT, 03.06.2015. Online im Internet:
http://prreport.de/home/aktuell/article/9843-dieser-platte-ansatz-hilft-der-transparenz-debatte-nicht
(Stand 10.08.2015)

Matzat, L. Warum das Lobbyradar so kaum zu gebrauchen ist. Datenjournalist 06.06.2015. Online im Internet: http://datenjournalist.de/warum-das-lobbyradar-so-kaum-zu-gebrauchen-ist/ (Stand 10.08.2015)

Bradshaw, S., Dekena, G., Udacity: Data Wrangling with MongoDB. Online im Internet:https://www.udacity.com/course/data-wrangling-with-mongodb--ud032 (Stand 14.08.2015)

Harth, P. ZDF-Lobbyradar: Auf dem zweiten Auge bleiben Sie blind. Online im Internet:
http://info.kopp-verlag.de/hintergruende/enthuellungen/peter-harth/zdf-lobbyradar-auf-dem-zweiten-auge-bleiben-sie-blind.html (Stand 21.09.2015)