Project Report

Student Project Semantic Web Technologies HWS18

Presented by

Nele Ecker
Alex Lütke
Marvin Messenzehl

Submitted to the
Data and Web Science Group
Prof. Dr. Heiko Paulheim & Sven Hertling
University of Mannheim

October - December 2018

Contents

1 Application Domain and Goals

1.1 Problem Statement And User Needs

Everyone who does regular city trips knows the problem. You visit a foreign city and just don't get the same experience as people who live there and know the place well. Typical information sources, like tourist centers, provide a very biased opinion on what one should see and what not. But what if somebody wants to look for something special or has special interests? If somebody really wants to get to know a city and the different districts, this takes a lot of time and manual preparation.

This problem should be solved within this project through an application that uses semantic web technologies. This application will be named *Semaps - A semantic approach to maps.*

1.2 Application Goal

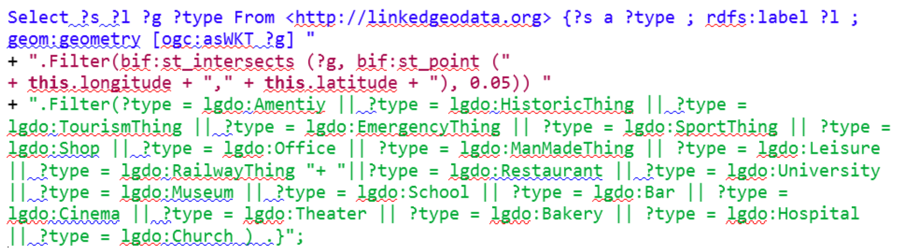
The overall goal of the developed application is to give the user a visual help to easily identify the particularities of the different city districts. This should be done with the help of a map as the main user interface where the different districts are marked in different colors in the form of rectangles.
Examples of this would be categories like *shopping, tourists, universities* and a lot more. The detailed categories depend on the districts, building and the offer of the respective city.

In the end, the user should visit the web application where he/she sees a map of a city where different special districts are marked. Therefore trips can be planned more individual and efficient.

2 Datasets

The application is based on two different datasets, namely LinkedGeoData and Dbpedia. It was decided to work with both datasets separately, due to the fact that LinkedGeoData is already very detailed and most data of Dbpedia is already in the other dataset. Having to different datasets allows the user to compare the results and can also give hints how accurate Dbpedia actually is.

To get the data from LinkedGeoData, different approaches where tried. The first approach was to download all the data and use it offline. This soon appeared to be not sophisticating, because of multiple reasons. On the first hand the dataset is split, based on the main categories Amenity, SportThings, HistoricalThings, etc,



```
Select ?s ?l ?g ?type From <http://linkedgeodata.org> {?s a ?type ; rdfs:label ?l ;
geom:geometry [ogc:asWKT ?g] "
+ ".Filter(bif:st_intersects (?g, bif:st_point ("
+ this.longitude + "," + this.latitude + "), 0.05)) "
+ ".Filter(?type = lgdo:Amentiy || ?type = lgdo:HistoricThing || ?type =
lgdo:TourismThing || ?type = lgdo:EmergencyThing || ?type = lgdo:SportThing || ?type =
lgdo:Shop || ?type = lgdo:Office || ?type = lgdo:ManMadeThing || ?type = lgdo:Leisure
|| ?type = lgdo:RailwayThing "+ "||?type = lgdo:Restaurant || ?type = lgdo:University
|| ?type = lgdo:Museum || ?type = lgdo:School || ?type = lgdo:Bar || ?type =
lgdo:Cinema || ?type = lgdo:Theater || ?type = lgdo:Bakery || ?type = lgdo:Hospital
|| ?type = lgdo:Church ) .}";
```

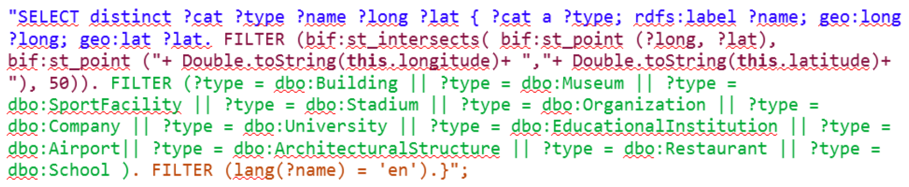
Figure 1: Sparql query to select all data of LinkedGeoData

with each subdataset having a large size and in case that you want to use all the data more than 40 datasets have to be loaded. On the other hand loading single datasets into a model when starting the application takes very long. Small datasets, for example all MilitaryThings need only a few seconds to be loaded, but others as for example HistoricThings need approximately one hour. As this is not feasible, to load every time that a user starts a request, it was decided to look for other ways to load the data. The most easiest way was to directly use the sparql endpoint of LinkedGeoData, which can be simply used in Jena. The used query can be found in Figure 1.

The query consists of three parts which are differently colored. The blue part selects the values of the needed variables, while the red and the green part show two different filters. The first filter is used to not read the data of the whole world, but only in a given region. The region that should be used is defined by the user and then used by this.latitude and this.longitude. Using st_intersects makes it possible to get all points that are in a specified radius around a given point. It was decided to use a 50km radius around the point that the user chose.

In the first version of the application only the location filter was used. That allowed us to use all existing categories, what would have made the clustering very dynamic. But as there are many different categories, also some very general and thus for our use case uninteresting ones, the time to get all triples was too long. The requests were interrupted after 15 minutes waiting time, but in our opinion even a minute waiting time is not acceptable for the application.

Due to these experiences the second filter was introduced, this time filtering the categories. Because of the separation of the dataset based on the main categories, as it is done for the download, we are already aware of the main categories which are also represented in the filter. Furthermore some more specialized categories, like museum and university are added, as they are more interesting for the clustering. Having the second filter the request needs only a few seconds, what makes the application usable.



```
"SELECT distinct ?cat ?type ?name ?long ?lat { ?cat a ?type; rdfs:label ?name; geo:long
?long; geo:lat ?lat. FILTER (bif:st_intersects( bif:st_point (?long, ?lat),
bif:st_point ("+ Double.toString(this.longitude)+ ","+ Double.toString(this.latitude)+
"), 50)). FILTER (?type = dbo:Building || ?type = dbo:Museum || ?type =
dbo:SportFacility || ?type = dbo:Stadium || ?type = dbo:Organization || ?type =
dbo:Company || ?type = dbo:University || ?type = dbo:EducationalInstitution || ?type =
dbo:Airport|| ?type = dbo:ArchitecturalStructure || ?type = dbo:Restaurant || ?type =
dbo:School ). FILTER (lang(?name) = 'en').}";
```

Figure 2: Sparql query to select all data of Dbpedia

To get the data of Dbpedia, the corresponding sparql endpoint was directly used. The query is shown in Figure 2. It is similar to the one used for LinkedGeoData, except a third filter was added, due to the fact that Dbpedia allows to have multiple labels for each resource with different language tags. Without using the language filter, there would be multiple instances for each class, one per label. Using the filter makes it sure that only the english label is used for the application and thus only one instance is created, as expected. Having multiple labels was not experienced using LinkedGeoData, that's why the filter is not in the first query. Basically the same information was selected, except the longitude and latitude are read separately, as this worked better with the location filter. The filter on the categories has been modified, to filter the categories that are attached to a Dbpedia resource. The general idea of the types has not changed, as it is also possible to look for museums and universities, but the URIs are different.

While exploring the different resources of Dbpedia manually, one major issue for the application was found. Not all points of interest that can be found in Dbpedia have information about their location. These points can therefore not be used for the clustering and falsify the result comapared to LinkedGeoData.

3 Techniques

One of the main ideas during development was the decoupling of major steps in Semap's processing pipeline. So the project is split into three smaller part: The user interface, the processing pipeline and REST interfaces. The frontend website is primarily responsible for the rendering of visual elements. The computation of clusters takes place on a separate backend server, which is contacted by the website using a REST-interface on the same backend server. The following sections will be based on this trisection and elaborate in more detail on each of the components.

3.1 User Interface

The primary access to the Semaps application happens in form of a website, where users will get visual information about a city. As a starting point, the city of Mannheim is set. The whole website is a responsive, single page web application with following user interface elements:

- a search bar, where the user can search for specific places, cities, countries, and addresses
- Three buttons with additional information like a legend of colors, information about the project and a link to the open source repository which contains the code of the frontend
- A map in the background with colored clusters and relevant data markers in the city

In terms of technology, Semaps is entirely developed with the help of the *React*[1]. React is a *JavaScript*-based frontend framework which follows component-based design philosophy. This means that the interface is broken down to small, atomic subsections, also described as components, which can be reused in the entire application.

The base functionality of modern web applications can be enhanced through so-called package managers. Because React is based on the server architecture of *NodeJS*[2] the used package manager here is *NPM*[3], which is short for Node Package Manager. With the help of such package managers, developers can use external functionalities bundled in packages, provided by other developers. In the case of Semaps, different packages were adapted. For example the package *axios*[4], which provides an easier way of handling asynchronous HTTP requests. The two main packages used were *leaflet*[5] and *react-leaflet*[6]. Those packages provide functionalities to implement different map features of *Open Street Maps*[7]. At the same time, this builds the center of the Semaps User interface.

To depict the computed clusters in the backend the defined standard format of *geoJSON* files was used. A geoJSON file is a special kind of JSON files which includes a set of features. In the case of Semaps, those features were interesting

[1] `https://reactjs.org`
[2] `https://www.nodejs.org`
[3] `https://npmjs.org`
[4] `https://github.com/axios/axios`
[5] `https://leafletjs.com`
[6] `https://react-leaflet.js.org`
[7] `http://openstreetmap.org`

data points and places, visualized by markers and the computed clusters in forms of colored rectangles. With the help of the geoJSON format, there is no major business logic necessary on the frontend. The geoJSON data is simply laid over the map and the different clusters are colored respectively.

In the end, the application is deployed with the help of *Heroku*[8], a popular and easy to use Platform-as-a-Service provider.

3.2 REST Interface

When a request is sent to the REST interface, the server reads given parameters and schedules the crawling and clustering of the data as well as the assembly of a response in the GeoJSON format. So the interface does not execute any functional computations; it rather passes on the request to different execution modules using Java dependency injection. The REST interface itself relies on the *Spring Boot* framework and is hosted at the Platform-as-a-Service provider *Heroku*. Spring Boot is a framework built on top of the spring framework, which facilitates the development of web applications and requires only very little configuration overhead. So it follows the architectural notion of microservices.

3.3 Processing Pipeline

A core object structure underlies the processing pipeline, which is utilized whenever the backend server is contacted by the UI website. I.e. After crawling the data from respective servers, the results are saved in internal object representations and read when the processing takes place. The core object structure is tuned towards the task of semantically clustering regions and thus facilitates the use of multiple clustering implementations with diverse libraries. The UML class diagram in figure 3 summarizes the object structure:

The figure shows the centrality of the Instance class. All other attributes interact with it, as they rely on the information held in instance objects. Practically speaking, an instance represents a real-world institution like a specific bar or school. Each object of the instance class can have a number of Categories assigned to it. Categories make up the semantic context of an institution. They have a name like “museum” or “bar” and a weight. The semap project assumes, that the deeper a category is in its ontology, the more informative it is. That is why a detailed category like “Irish pub” is meant to get a higher weight than higher-level categories like “thing”. The clustering algorithm can then prefer to place instances based on the deep-level knowledge into clusters, resulting in more specific clusters.

[8]`https://heroku.com/`

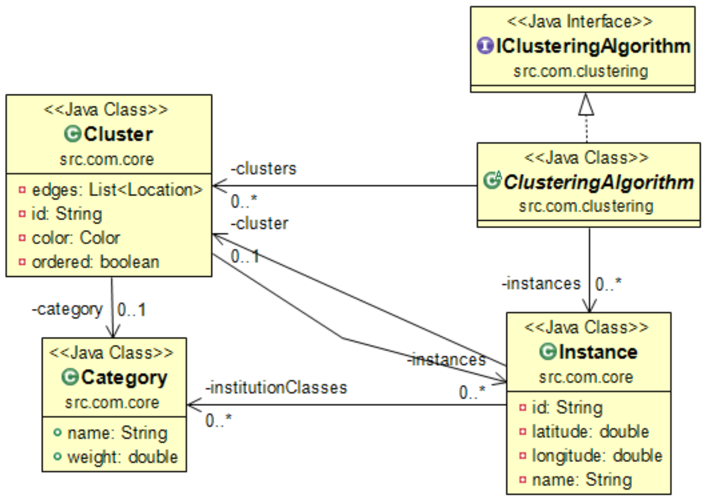


Figure 3: UML class diagram of the core object structure

Before clustering takes place, no instance is assigned to a cluster, as no cluster exists. Ideally after clustering each instance is assigned to exactly one cluster. When running a clustering algorithm (here represented by an abstract class), it will create new objects of type Cluster using a list of instances and assign those instances to a cluster. The different implementations of clustering algorithms are not depicted in the figure for abstraction purposes. But in the end, the actual implementations are of type ClusteringAlgorithm, so they just inherit from the abstract class ClusteringAlgorithm and the interface IClusteringAlgorithm. In the project, a simple square clustering algorithm and a DBScan were implemented.

Simple Square Clustering The simple square clustering algorithm a rather straight-forward implementation of clustering algorithms. It considers each element from the list of instances and creates clusters based on their position. So for each instance, the simple square clustering implementation will check if there is already a cluster, in which the current instance is located and will assign it to this cluster or create a new cluster correspondingly. After having done so for the entire list of instances, each clusters gets a category. The computation is based on a scoring function, which determines the times, a category appears in a cluster and weights each appearance by the category's weight-property. The category with the highest weight is then allocated to a cluster. Algorithm 1 formalizes the procedures again. In sum, the simple square clustering algorithm does not take into account the semantic context (i.e. the categories) of instances and only uses positional information initially. That is why the results are not guaranteed to be semantically meaningful. However, the algorithm runs very fast, as it is linked to very little computational overhead.

Algorithm 1 Sketch of the simple quare clustering algorithm

```
$I \leftarrow$ set of all instances
$C \leftarrow \emptyset$
for all $i \in I$ do
  for all $c \in C$ do
    if $i\ located\ in\ c$ then
      $c.add(i)$
    else
      $x \leftarrow createCluster(i.latitude, i.longitude)$
      $C \leftarrow C \cap x$
    end if
  end for
end for
$CAT \leftarrow \emptyset$
for all $i \in C.getInstances()$ do
  for all $category \in i.getCategories()$ do
    $CAT_{category} \leftarrow CAT_{category}.getScore() + category.getScore()$
    $CAT \leftarrow CAT \cap CAT_{category}$
  end for
end for
return $max(CAT.getAllScores())$
```

DBScan DBScan is the one rather advanced clustering algorithm in the semap project and relies on the Weka library implementation. The DBScan tries to build clusters based on two pieces of information: The position of instances and their categories. The position is determined by the longitude and latitude coordinates and is normalized on a scale from zero to one; alternatively it can be z-normalized by a parameter setting in the Java coding. Categories are one-hot-encoded, so that each instance gets a vectors consisting of values in the range from one to zero. The vector represents the set of all categories. Each category is represented by a positional value in the vector. All the categories an instance does not belong to, take the value 0 in the vector. The categories, an instance belongs to, take a normalized value greater than zero. The value is determined by the formula:

$$n := number\ of\ categories\ of\ an\ instance \tag{1}$$

$$w_i := weight\ of\ the\ ith\ category\ of\ an\ instance \tag{2}$$

$$w_c := weight\ of\ current\ category \tag{3}$$

$$val_c = \frac{w_c}{\sum_{i=0}^{n} w_i} \tag{4}$$

So the value depends on the number of instance's categories and the weight of the respective categories. This method ensures, that the weighted vector values have the same impact on the clustering as the positional information latitude and longitude. Before the DBScan algorithm can start working in Weka, a preprocessing pipeline transform the internal object representation as described in figure 3 into the object representation needed by the Weka library. It therefore writes an artificial .arff-file to disk and reads it again with tools from the weka library. Afterwards, the DBScan is executed and the results again returned into the core object representation of semap. The entire procedure of the DBScan in the semap project is summarized in algorithm 2 as follows:

Algorithm 2 Sketch of the DBScan clustering algorithm

```
$I \leftarrow$ set of all instances
$C \leftarrow \emptyset$
for all $i \in I$ do
   $x \leftarrow normalize(i.latitude)$
   $i.setLatitude(x)$
   $x \leftarrow normalize(i.longitude)$
   $i.setLongitude(x)$
   $\vec{y} \leftarrow oneHotEncode(i.categories)$
   for all $z \in \vec{y}$ do
      $z \leftarrow normalize\ z\ as\ to\ formula\ (4)$
   end for
   $i.categories \leftarrow \vec{y}$
end for
$dBScan(I)$
$CAT \leftarrow \emptyset$
for all $i \in C.getInstances()$ do
   for all $category \in i.getCategories()$ do
      $CAT_{category} \leftarrow CAT_{category}.getScore() + category.getScore()$
      $CAT \leftarrow CAT \cap CAT_{category}$
   end for
end for
return $max(CAT.getAllScores())$
```

In sum, the value normalization and object transformations are computationally expensive operations. While for example the simple square algorithm clusters multiple millions of instances within few milliseconds, the DBScan requires about 30 seconds to do so with only 15 thousand instances. However, the major part of this timeframe is not spent in the normalization and object transformation implementation. The most time is actually needed by Weka's DBScan algorithm itself. So this runtime behavior is a major limitation of the DBScan algorithm. Nonetheless, the DBScan algorithm is capable of finding much more meaningful results. It considers categories of instances already when performing the clustering and not afterwards. It is also not limited to geographically speaking squared clusters, but has arbitrary decision boundaries. That makes this algorithm a very powerful tool for detailed analyses on regional, public data.

4 Results

Main outcome of this report's underlying work is a live application publicly accessible on the Internet. The application hosts the user interface described in the previous section. When opening the URL on the Internet, a map will be shown including a colorful overlay indicating the different clusters for a region. Since the clusters are generated on-the-fly in response to a user query, the overlay is only depicted for a limited region. Figure 4 shows a sample map including the overlay for Mannheim's city center. The figure summarizes the different pieces of information, the live application is able to provide: The blue points represent various points of interest taken from either the LinkedGeoData or DBPedia databse; and the colorful rectangles display computed clusters. In this case, those results of Mannheim serve as a reference for the evaluation in the following, as team members had the most knowledge about this region.

Most importantly, the aspect of *meaningfulness* plays a role for the substantial correctness of the application. For this purpose, meaningfulness can be defined as a sensible labelling of clusters. Meaningfulness cannot be formalized; however, cluster labels should be as specific as possible. General terms such that a human could not derive valuable information from it, is to be avoided. According to this definition, the sample result of Mannheim shown in figure 4 are now to be assessed. Generally, the legend reveals that cluster labels are quite understandable and meaningful to a human. Only the label *shop* leaves room for interpretation. Taking into account the spatial dimension of clusters, the label shop offers potential for additional criticism. The district *Jungbusch*, known for its lively bar scene. The name shops does not come very close to this information. Likewise, the dataset is very much biased towards schools, i.e. many Mannheim schools are registered in the

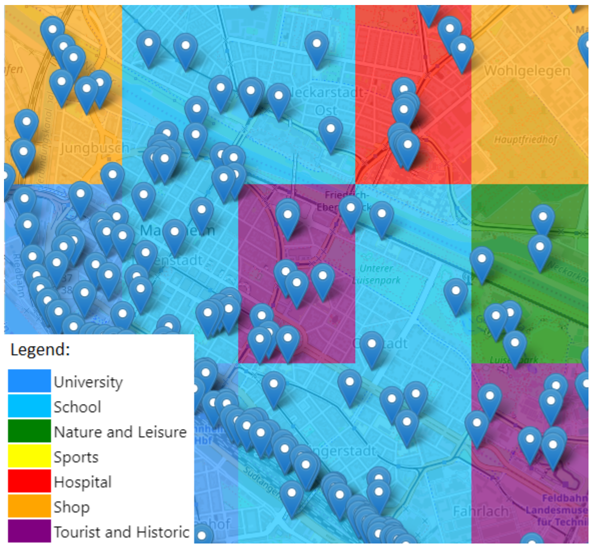


Figure 4: Evaluation based on sample result in Mannheim

database, which leads to many datasets getting the label *school*. Nonetheless, several other clusters provide quite sensible information. The university cluster as well as the touristic cluster in the area of the Mannheimer Wasserturm or even a Universitätsklinikum cluster are explicitely recognized by the application.

So, to sum it all up, the application is capable of finding useful information, but heavily depends on the data given in the origin datasets DBPedia and LinkedGeoData.

5 Known Limitations

In which domains does the application not work?
Are there queries which cannot be answered?
Why?
How could you overcome those limitations, given more time?

For the processing period of one month, the results and the information level of the application are very satisfactory. Nevertheless, there were some limitations, and therefore room for improvement, which should be described shortly in the following chapter.

One major limitation of the project is the data quality. Of course, the clustering algorithm can only be as good as the data input. For some cities, there are a lot of imbalances when it comes to interesting city points. One example here would be the type *ArchitecturalStructure* or *Building* in the DB-Pedia dataset. Those to labels are rather general and therefore way more common than other labels like *University* or *Museum*. As a result, there are way more clusters in those general categories. To improve this problem in future development efforts, the data should be analyzed more specific and general labels should be ranked lower to create more balance.

Another limitation besides the data quality is the availability of the data itself. Because of the fact that there is generally more data available for larger cities, the result for small cities and countryside villages is usually worse. Where the capital city of Germany, Berlin, has way over 100 data points and markers to evaluate clusters upon, the city of Bad Dürkheim for example only has four data points in the DB Pedia dataset. Thus the computed clusters are limited by the availability of the data. One way to solve this would be the combination of different data sources in real-time. Especially for smaller cities, so that there is more of a balance between different cities.

A third limitation is described by the infrastructure itself. As already mentioned before, the Platform-as-a-Service provider Heroku was chosen for deployment for front- and backend. Because of limited resources the free tier of the deployment platform was chosen which brings one major disadvantage. If the application is inactive for a while it must be re-activated with the first request which usually results in a loading time around 15 seconds. Therefore some delays can happen on the user interface. This problem can simply be solved by upgrading to a Heroku premium plan[9].

6 Lessons Learned

Besides the overall satisfaction with the development and the results of the project, there were still some obstacles and challenges during development that had to be faced.

One of the major challenges was to find relevant data sources that provide enough data that could be used for the clustering. Besides that, the integration of the data turned out to be hard. Because of the different formats a lot of the available

[9]more infos under `https://www.heroku.com/pricing`

data had to be re-formatted for a simpler usage.

As mentioned before the development team implemented the major parts of the different application areas independently. In the end, all results had to be brought together and it had to be ensured that every part works seemingly together with the rest of the application. This was especially difficult for dynamic user inputs. If the user searches for a specific city this input has to be routed to the backend server where the data is dynamically fetched, prepared and afterward clustered. The final result has to be brought into a JSON format and routed back to the application frontend where the result is displayed. This processing pipeline contains complex, individual steps and a lot of data cleaning is needed to ensure the correct functionality. Additionally, a lot of different techniques and technologies were used in the project, so that common standards for development and communication had to be found. A unified, more homogenous tech stack would make this challenge a lot easier.

All in all the project was overall successful and the achieved results are satisfactory and fulfill the predefined application goal. The application of Semaps brings a semantic web approach to maps and makes it easier to get to know a city by simply looking at the map.