Enabling Spatio-Temporal Search in Open Data

Sebastian Neumaier
Axel Polleres
Enabling Spatio-Temporal Search in Open Data

Sebastian Neumaier\textsuperscript{a,1}, Axel Polleres\textsuperscript{a,b,c,2}

\textsuperscript{a}Vienna University of Economics and Business, Vienna, Austria
\textsuperscript{b}Complexity Science Hub Vienna, Austria
\textsuperscript{c}Stanford University, CA, USA

Abstract

Intuitively, most datasets found in Open Data are organised by spatio-temporal scope, that is, single datasets provide data for a certain region, valid for a certain time period. For many use cases (such as for instance data journalism and fact checking) a pre-dominant need is to scope down the relevant datasets to a particular period or region. Therefore, we argue that spatio-temporal search is a crucial need for Open Data portals and across Open Data portals, yet - to the best of our knowledge - no working solution exists. We argue that - just like for regular Web search - knowledge graphs can be helpful to significantly improve search: in fact, the ingredients for a public knowledge graph of geographic entities as well as time periods and events exist already on the Web of Data, although they have not yet been integrated and applied – in a principled manner – to the use case of Open Data search. In the present paper we aim at doing just that: we (i) present a scalable approach to construct a spatio-temporal knowledge graph that hierarchically structures geographical, as well as temporal entities, (ii) annotate a large corpus of tabular datasets from open data portals, (iii) enable structured, spatio-temporal search over Open Data catalogs through our spatio-temporal knowledge graph, both via a search interface as well as via a SPARQL endpoint, available at \url{data.wu.ac.at/odgraphsearch/}

Keywords: open data, spatio-temporal labelling, spatio-temporal knowledge graph

1. Introduction

Open Data has gained a lot of popularity and support by governments in terms of improving transparency and enabling new business models: Governments and public institutions, but also private companies, provide open access to raw data with the goal to present accountable records [1], for instance in terms of statistical data, but also in fulfillment of regulatory requirements such as, e.g., the EU’s INSPIRE directive.\textsuperscript{3} The idea to provide raw data, instead of only human-readable reports and documents, is mainly driven by providing direct, machine-processable access to the data, and enable broad and arbitrary (through open licences) reuse of such data [2, 3].

Yet, it is still humans who mostly consume this data, so far mostly developers of apps who are needed as intermediaries to make the said data actually accessible to end users. Even worse, search in Open Data is largely unaddressed, since most known search techniques rely on keywords and human-readable cues in documents. With the advent of “Knowledge Graphs” search recently has been revolutionized in that search results can be categorized, browsed and ranked according to well-known concepts and relations, which cover typical search scenarios in search engines.

But these scenarios are different for Open Data: in our experience, we note that search in Open Data probably needs to be targeted from a different angle than keyword-search (alone). Intuitively,
most datasets found in Open Data – as it is mostly regional/national census-based – are organized by spatio-temporal scope, that is, single datasets provide data for a certain region, are valid for a certain time period. For many use cases (such as for instance data journalism and fact checking) the pre-dominant need is to scope down the relevant datasets to a particular period or region. Therefore, we argue that spatio-temporal search is a crucial requirement across Open Data portals [4].

We further argue that also for this use case, knowledge graphs can be helpful: in fact, the ingredients for a public knowledge graph of geographic entities as well as time periods and events exist already, although they have not yet been integrated and applied – in a principled manner – to the use case of Open Data search. In the present paper we aim at doing just that: We present a scalable approach to (i) construct a spatio-temporal knowledge graph that hierarchically structures geographical entities, as well as temporal entities, (ii) annotate a large corpus of tabular Open Data, currently holding datasets from eleven European (governmental) data portals, (iii) enable structured, spatio-temporal search over Open Data catalogs through this spatio-temporal knowledge graph, available at http://data.wu.ac.at/odgraphsearch/.

In more detail, we make the following concrete contributions:

- A detailed construction of a hierarchical base knowledge graph for geo-entities and temporal entities and links between them.
- A scalable labelling algorithm for linking open datasets (both on a dataset-level and on a record-level) to this knowledge graph.
- Indexing and annotation of datasets and metadata from 11 Open Data portals from 10 European countries and an evaluation of the annotated datasets to illustrate the feasibility and effectiveness of the approach.
- A prototypical search interface, consisting of a web user interface allowing faceted and full-text search, a RESTful JSON API that allows programmatic access to the search UI, as well as API-access to retrieve the indexed dataset and respective RDF representations
- A SPARQL endpoint that exposes the annotated links and allows structured search queries.

- Code, data and a description on how to re-run our experiments, which we hope to be a viable basis for further research extending our results, are available for re-use (under GNU General Public License v3.0).

The remainder of this paper is structured as follows: In the following Section 2 we introduce (linked) datasets, repositories and endpoints to retrieve relevant temporal and spatial information. Section 3 provides a schematic description of the construction and integration of these sources into our base knowledge graph; its actual realization in terms of implementation details is fully explained in Appendix A. In Section 4 we present the algorithms to add spatio-temporal annotations to datasets from Open Data portals, and evaluate and discuss the performance (in terms of precision and recall based on a manually generated sample) and limitations of our approach. The vocabularies and schema of our RDF data export are explained in Section 5 and the back-end, the user interface and the SPARQL endpoint (including example queries) are presented in Section 6. We provide related and complementary approaches in Section 7, and eventually we conclude in Section 8.

2. Background

When people think of spatial and temporal context of data, they usually think of concepts rather than numbers, that is “countries” or “cities” instead of coordinates or a bounding polygon, or an “event” or “time period” instead of e.g. start times end times. In terms of data search that could mean someone searching for datasets containing information about demographics for the last government’s term (or in other words between the last two general elections).

In order to enable such search by spatio-temporal concepts, our goal is to build a concise, but effective knowledge base, that collects the relevant concepts from openly available data into a coherent, base knowledge graph, for both (i) enabling spatio-temporal search within Open Data portals and (ii) interlinking Open Data portals with other datasets by the principles of Linked Data.

The following section gives an overview of the available datasets and sources to construct the base knowledge graph of temporal- and geo-entities.
GeoNames.org. The GeoNames database contains over 10 million geographical names of entities such as countries, cities, regions, and villages. It assigns unique identifiers to geo-entities and provides a detailed hierarchical description including countries, federal states, regions, cities, etc. For instance, the GeoNames-entity for the city of Munich has the parent relationship “Munich, Urban District”, which is located in the region “Upper Bavaria” of the federal state “Bavaria” in the country “Germany”, i.e. the GeoNames database allows us to extract the following hierarchical relation for the city of Munich:

\[
\text{Germany} > \text{Bavaria} > \text{Upper Bavaria} > \text{Munich, Urban District} > \text{Munich}
\]

The relations are based on the GeoNames ontology which defines first-order administrative division (gn:A), second-order (gn:A.ADM2) , ... (until gn:A.ADM5) for countries, states, cities, and city districts/sub-regions. In this work we make use of an RDF dump of the GeoNames database, which consists of alternative names and hierarchical relations of all the entities.

OpenStreetMap (OSM). OSM was founded in 2004 as a collaborative project to create free editable geospatial data. The map data is mainly produced by volunteers using GPS devices (on foot, bicycle, car, ..) and later by importing commercial and government sources, e.g., aerial photographies. Initially, the project focused on mapping the United Kingdom but soon was extended to a worldwide effort. OSM uses four basic “elements” to describe geo-information:

- **Nodes** in OSM are specific points defined by a latitude and longitude.
- **Ways** are ordered lists of nodes that define a line. OSM ways can also define polygons, i.e. a closed list of nodes.
- **Relations** define relationships between different OSM elements, e.g., a route is defined as a relation of multiple ways (such as highway, cycle route, bus route) along the same route.
- **Tags** are used to describe the meaning of any elements, e.g., there could be a tag `highway=residential` (tags are represented as key-value pairs) which is used on a way element to indicate a road within settlement.

There are already existing works which exploit the potential of OSM to enrich and link other sources. For instance, in [5] we have extracted indicators, such as the number of hotels or libraries in a city, from OSM to collect statistical information about cities.

Likewise, the software library Libpostal uses addresses and places extracted from OSM: it provides street address parsing and normalization by using machine learning algorithms on top of the OSM data. The library converts free-form addresses into clean normalized forms and can therefore be used as a pre-processing step to geo-tagging of streets and addresses. We integrate Libpostal in our framework in order to detect and filter streets and city names in text and address lines.

Sources to obtain Postal codes and NUTS codes. Postal codes are regional codes consisting of a series of letters (not necessarily digits) with the purpose of sorting mail. Since postal codes are country-specific identifiers, and their granularity and availability strongly varies for different countries, there is no single, complete, data source to retrieve these codes. The most reliable way to get the complete dataset is typically via governmental agencies (made easy, in case they publish the codes as open data). Another source worth mentioning for matching postal codes is GeoNames: it provides a collection of postal codes for several countries and the respective name of the places/districts.

---

4http://www.geonames.org/
5http://www.geonames.org/6559171/
6http://www.geonames.org/ontology/ontology\_v3.1.rdf
7Here, gn: corresponds to the namespace URL http://www.geonames.org/ontology/ontology.html, last accessed 2018-01-05
8https://www.openstreetmap.org/
9A detailed description can be found at the OSM documentation pages: http://wiki.openstreetmap.org/wiki/Main\_Page
10http://www.geonames.org/export/zip/, last accessed 2018-01-05
12https://medium.com/@albarrentine/statistical-nlp-on-openstreetmap-b9d573e6cc86, last accessed 2017-09-12
13For instance, the complete list of Austrian postal codes is available as CSV via the Austrian Open Data portal: https://www.data.gv.at/katalog/dataset/f76ed887-00d6-450f-a158-9f8b1cbeebf, last accessed 2018-04-03
14https://www.data.gv.at/katalog/dataset/f76ed887-00d6-450f-a158-9f8b1cbeebf, last accessed 2018-04-03
Partially, postal codes for certain countries are available in the knowledge bases of Wikidata and DBpedia (see below) for the respective entries of the geo-entities (using “postal code” properties). However, we stress that these entries are not complete, i.e., not all postal codes are available in the knowledge bases as not all respective geo-entities are present, and also, the codes’ representation is not standardized.

NUTS (French: nomenclature des unités territoriales statistiques). Apart from national postal codes another geocode standard has been developed and is being regulated by the European Union (EU). It references the statistical subdivisions of all EU member states in three hierarchical levels, NUTS 1, 2, and 3. All codes start with the two-letter ISO 3166-1 [6] country code and each level adds an additional number to the code. The current NUTS classification lists 98 regions at NUTS 1, 276 regions at NUTS 2 and 1342 regions at NUTS 3 level and is available from the EC’s Webpage. 15

Wikidata and DBpedia. These domain-independent, multi-lingual, knowledge bases provide structured content and factual data. While DBpedia [7] is automatically generated by extracting information from Wikipedia, Wikidata [8], in contrary, is a collaboratively edited knowledge base which is intended to provide information to Wikipedia. These knowledge bases already partially include links to GeoNames, NUTS identifier, and postal code entries, as well as temporal knowledge for events and periods, e.g., elections, news events, and historical epochs, which we also harvest to complete our base knowledge graph.

PeriodO. The PeriodO project [9] offers a gazetteer of historical, art-historical, and archaeological periods. The user interface allows to query and filter the periods by different facets. Further, the authors published the full dataset as JSON-LD download16 and re-use the W3C skos, time and dcterms:spatial vocabularies to describe the temporal and spatial extent of the periods. For instance, the following (shortened) PeriodO entry describes the period of the First World War:

```sparql
@prefix dbr: <http://dbpedia.org/resource/> .
@prefix skos: <http://www.w3.org/2004/02/skos/core#>
@prefix dcterms: <http://purl.org/dc/terms/> .
@prefix time: <http://www.w3.org/2006/time#> .

<http://n2t.net/ark:/99152/p0kh9ds3566> dcterms:spatial dbr:United_Kingdom ;
skos:altLabel "First World War" ;
time:intervalFinishedBy [ skos:prefLabel "1918" ;
time:hasDateTimeDescription [ time:year "1918"^^xsd:gYear ] ] ;
time:intervalStartedBy [ skos:prefLabel "1914" ;
time:hasDateTimeDescription [ time:year "1914"^^xsd:gYear ] ] .
```

3. Base Knowledge Graph Construction

The previous section listed several geo-data repositories as well as datasets containing time periods and event data – some already available as Linked Data via an endpoint – which we use in the following to build up a base knowledge graph: Section 3.1 describes the extraction and integration of geo-spatial, and Section 3.2 of temporal knowledge.

Herein, we describe the composition of the graph by presenting conceptual SPARQL CONSTRUCT queries. This means that (most of) the presented queries cannot be executed because either there is no respective endpoint available or the query is not feasible and times out. While this section shall serve as a schematic specification of the constructed graph, we detail the actual realization of the queries in Appendix A.

Still, we deem the use of these conceptual SPARQL CONSTRUCT useful as a mechanism to declaratively express knowledge graph compilation from Linked Data sources, following Rospocher et al.’s definition, who describe knowledge graphs as “a knowledge-base of facts about entities typically obtained from structured repositories”[10]. 17

3.1. Spatial Knowledge

Our knowledge graph of geo-entities is based on the GeoNames hierarchy, where we want to extract

• geo-entities and their labels

---

15http://ec.europa.eu/eurostat/web/nuts/overview, last accessed 2018-01-05
• links parent entities and particularly the containing country
• coordinates in terms of points and (if available) geometries in terms of polygons
• postal codes (again, if available)
• sameAs-links to other sources such as DBpedia, OSM, or Wikidata (again, if available)

The respective SPARQL CONSTRUCT query over the GeoNames dataset in Figure 1 displays how the hierarchical data could be extracted from a (currently nonexistent) GeoNames SPARQL endpoint for a selected country ?c, i.e., if a respective SPARQL endpoint existed to access GeoNames' published RDF data, we could get all the relevant data for our knowledge graph per country, by replacing ?c in this query with a concrete country URL, such as http://sws.geonames.org/2782113/ (for Austria). The GeoNames RDF data partially already contains external links to DBpedia (using rdfs:seeAlso) which we model as equivalent identifiers using owl:sameAs. The hierarchy is constructed using the gn:parentFeature property. As GeoNames offers various different properties containing names, we extract all official English and (for the moment) German names, as we will use those later on for building our search index.

The query in Figure 2 then displays how we integrate the information in Wikidata into our spatial knowledge graph. In particular, Wikidata serves as a source to add labels and links for postal codes (gn:postalCode) and NUTS identifiers (wdt:P605) for the respective geographical entities. Further, we again add external links (to OSM and Wikidata itself) that we harvest from Wikidata as owl:sameAs relations to our graph.

The query in Figure 3 conceptually shows how and which data we extract for certain OSM entities into our knowledge graph. We note here that OSM does not provide an RDF or SPARQL interface, but the idea is that we - roughly - perceive and process data returned by OSM's Nominatim API in JSON as JSON-LD; details and pre-processing steps in Appendix A.2 below.

3.2. Temporal Knowledge

As for temporal knowledge, we aim to compile into our knowledge graph a base set of temporal-entities (that is, named periods and events from Wikidata and PeriodO) where we want to extract

• named events and their labels,
• links parent periods that they are part of, again to create a hierarchy,
• temporal extent in terms of a single beginning and end date, and
• links to a spatial coverage of the respective event or period (if available).

We observe here that temporal knowledge is typically less consolidated than geospatial knowledge, i.e. “important” named entities in terms of periods or events are not governed by internationally agreed and nationally governed structures such as border-agreements in terms of spatial entities. Even worse, cross-cultural differences such as different calendars or even timezones) add additional confusion. We still believe that the two sources, which we try to integrate here, which cover events of common interest in a multilingual setting on the one hand (Wikidata), and historical periods and epochs from the literature (PeriodO), provide a good starting point. In the future, it might be useful to index as well news events, or recurring periods or points

in time, such as public holidays, that occur regularly, but we did not find structured datasets available on Linked Data for that, which is the focus of our current work. So, we have to defer these to future work, or respectively, the creation of respective structured datasets as a challenge for the community: one obvious existing starting point here would be the work by Rospocher et al. [10] and the news events datasets they created in the EU Project NewsReader,19 which however we for the moment did not consider due to its fine granularity, which in our opinion is not needed in a majority of Open Data Search use cases.

Again, we model knowledge graph extraction in terms of conceptual SPARQL queries. We use the query in Figure 4 to extract events information from Wikidata. Note, that this query times out on the public Wikidata endpoint. Therefore, in order to extract the relevant events and time periods as described in Figure 4, we converted a local Wikidata dump to HDT [12], extracted only the relevant triples for the query, materialized the path expressions, and executed the targeted CONSTRUCT query over these extracts on a local endpoint; the full details are provided in Appendix A.3.

As you can see, again, we do not just extract existing triples from the source, but try to aggregate/flatten the representation to a handful of well-known predicates from Dublin Core (prefix dcterms:) and the OWL time ontology (prefix time:).

Likewise, we use the query in Figure 5 to extract periods from the PeriodO dataset, again using the same flattened representation. To execute this query, in this case we could simply download the available PeriodO dump into a local RDF store.

Note that in these queries – in a slight abuse of the OWL Time ontology – we “invented” the properties timex:hasStartTime and timex:hasEndTime that do not really exist in the original OWL time ontology. This is a compromise for the desired compactness of representation in our knowledge graph, i.e. these are mainly introduced as shortcuts to avoid the materialization of unnecessary blank nodes in the (for our purposes too) verbose notation of OWL Time. A proper representation using OWL Time could be easily reconstructed by means of the following CONSTRUCT query:

CONSTRUCT {
  ?X time:hasBeginning [ time:inXSDDateTime ?StartDate ] ;
  time:hasEnd [ time:inXSDDateTime ?EndDate ]
} WHERE {
  ?X timex:hasStartTime ?StartDate ;
  timex:hasEndTime ?EndDate
}

For this purpose we define our own vocabulary extension of the OWL Time ontology, for the moment, under the namespace http://data.wu.ac.at/ns/timex#.

4. Dataset Labelling Algorithm

In order to add spatial and temporal annotations to Open Data resources we use the CSV files
CONSTRUCT {  
timex:hasStartTime ?StartDateTime ; timex:hasEndTime ?EndDateTime ; dcterms:spatial ?geoentity .  
} WHERE {  
# find events with (for the moment) English, German, or non-language-specific labels:  
FILTER(LANG(?label) = "en" || LANG(?label) = "de" || LANG(?label) = "" ).  
# restrict to certain event categories, e.g. (for the moment) elections and sports events:  
{ # elections #sports competitions  
}  
{ # with a point in time or start end end date  
{ ?event wdt:P585 ?StartDate . FILTER( ?StartDate > "1900-01-01T00:00:00"^^xsd:dateTime ) }  
UNION  
{ ?event wdt:P580 ?StartDate . FILTER( ?StartDate > "1900-01-01T00:00:00"^^xsd:dateTime )  
}  
{ ?event wdt:P582 ?EndDate . FILTER( DATATYPE(?EndDate) = xsd:dateTime ) }  
}  
OPTIONAL { ?event wdt:P361 ?Parent }  
# specific spatialCoverage if available  
OPTIONAL { ?event wdt:P2676(?wdt:P171|wdt:P131) ?geoentity }  
OPTIONAL { ?event wdt:P2676(?wdt:P625) ?geocoordinates }  
BIND( if(bound(?EndDate), ?EndDate, xsd:dateTime(concat(str(xsd:date(?StartDateTime)),"T23:59:59"))) AS ?EndDateTime )  
} }

CONSTRUCT {  
timex:hasStartTime ?StartDateTime ; timex:hasEndTime ?EndDateTime .  
} WHERE {  
OPTIONAL { ?P periodo:spatialCoverage ?geo }  
OPTIONAL { ?P dcterms:spatial ?geo }  
OPTIONAL { ?P dcterms:isPartOf ?Parent. }  
OPTIONAL { ?End time:hasDateTimeDescription ?EndDate .  
OPTIONAL( ?EndDate time:year ?EndYear )  
OPTIONAL( ?EndDate periodo:latestYear ?EndYear )  
OPTIONAL( ?Start time:hasDateTimeDescription ?StartTime .  
OPTIONAL( ?StartTime time:year ?StartYear )  
OPTIONAL( ?StartTime periodo:earliestYear ?StartYear )  
OPTIONAL( ?Start ?(periodo:aux)+ ?StartTime. FILTER(isLiteral(?StartTime)) )  
OPTIONAL( ?End ?(periodo:aux)+ ?EndDate. FILTER(isLiteral(?EndDate)) )  
FILTER( ?StartYear >= "1900"^^xsd:gYear | xsd:integer(?StartYear) >= 1900 ||  
?EndYear >= "1900"^^xsd:gYear | xsd:integer(?EndYear) >= 1900 )  
BIND( xsd:dateTime(concat(str(?StartTime),"-01-01T00:00:00")) as ?StartDate )  
BIND( xsd:dateTime(concat(str(?EndDate),"-12-31T23:59:59")) as ?EndDate )  
}  
}
and metadata from the resources’ data portals as signals. The metadata descriptions and download links are provided by our Open Data Portal Watch framework [13] which monitors and archives over 260 data portals, and provides APIs to retrieve their metadata descriptions in an homogenized way using the W3C DCAT vocabulary [14]. Regarding the meta-information, we look into several available metadata-fields: we consider the title, description, the tags and keywords, and the publisher. For instance, the upper part of Figure 6 displays an example metadata description. It holds cues in the title and the publisher field (cf. “Veröffentlichende Stelle” - publishing agency) and holds a link to the actual dataset, a CSV file (cf. lower part in Figure 6), which we download and parse.

4.1. Geo-spatial labelling

The geo-spatial labelling algorithm uses the different types of labels in our base knowledge graph to annotate the metadata and CSV files from the input data portals.

4.1.1. CSVs

Initially, the columns of a CSV gets classified based on regular expressions for NUTS identifier and postal codes. While the NUTS pattern is rather restrictive, the postal codes pattern has to be very general, potentially allowing many false positives. Basically, the pattern is designed to allow all codes in the knowledge graph, and to filter out other strings, words, and decimals.

Potential NUTS column (based on the regular expression) get mapped to the existing NUTS identifier. If this is possible for a certain threshold (set to 90% of the values) we consider a column as NUTS identifier and add the respective semantic labels. In case of potential postal codes the algorithm again tries to map to existing postal codes, however, we restrict the set of codes to the originating country of the dataset. This again results in a set of semantic labels which we only accept with a threshold of 90%.

The labelling of string columns, i.e. set of words or texts, uses all the labels from GeoNames and OSM and is based on the following disambiguation algorithm:

Value disambiguation. The algorithm in Figure 7 describes shows how we disambiguate a set of string values based on the surroundings. First, the function get_context(values) counts all potential parent GeoNames entities for all of the values. To disambiguate a single value we use these counts and select the GeoNames candidate with the most votes from the context values’ parent regions; cf. disamb_value(value). The function get_geonames(value) returns all potential GeoNames entities for an input string. Additionally, we use the origin country of the dataset (if available) as a restriction, i.e., we only allow GeoNames labels from the matching country.

For instance, in Figure 6 the Austrian “Linz” candidate gets selected in favor of the German “Linz” because the disambiguation resulted in a higher score based on the matching predecessors “Upper Austria” and “Austria” for the other values in the column (Steyr, Wels, Altheim, ...).

If no GeoNames mapping was found the algorithm tries to instantiate the string values with OSM labels from the base knowledge graph. Again, the same disambiguation algorithm is applied, however, with the following two preprocessing steps for each input value:

\[20][A-Z][{}2][A-Z0-9][{}0,3]\]
\[21][((A-Z[{}d][{}2,4])|[[A-Z][{}1,2]++]=d(2,5)}\), \(s[A-Z][{}2,5})\), \(t([d[1,4]])=]
```python
# disambiguate a set of input values
def disamb_values(values, country):
    cont_par = get_context(values)
    disambiguated = []
    for v in values:
        v_id = disamb_value(v, country, cont_par)
        disambiguated.append(v_id)
    return disambiguated

# disambiguate a single value based on
# the parents of the surrounding values
def disamb_value(value, country, cont_par):
    candidates = get_geonames(value)
    c_score = {}
    for c in candidates:
        if country and country != c.country:
            continue
        else:
            parents = get_all_parents(c)
            for p in parents:
                c_score[p] += cont_par[p]
    top = sorted(c_score)[0]
    return top

# counts all parent values
def get_context(values):
    cont_par = {}
    for v in values:
        for c in get_geonames(value):
            parents = get_all_parents(c)
            for p in parents:
                cont_par[p] += 1
    return cont_par
```

Figure 7: Python code fragment for disambiguating a set of input values.

1. In order to better parse addresses, we use the Libpostal library (cf. Section 2) to extract streets and place names from strings.
2. We consider the context of a CSV row, e.g., if addresses in CSVs are separated into dedicated columns for street, number, city, state, etc. To do so we filter the allowed OSM labels by candidates within any extracted regions from the metadata description or from the corresponding CSV row (if geo-labels available).

4.1.2. Metadata descriptions

The CSVs’ meta-information at the data portals often give hints about the respective regions covering the actual data. Therefore, we use this additional source and try to extract geo-entities from the titles, descriptions and publishers of the datasets:

1. As a first step, we tokenize the input fields, and remove any stopwords. Also, we split any words that are separated by dashes, underscores, semicolon, etc.
2. The input is then grouped by word sequences of up to three words, i.e. all single words, groups of two words, ..., and the previously introduced algorithm for mapping a set of values to the GeoNames labels is applied (including the disambiguation step).

Figure 6 gives an example dataset description found on the Austrian data portal data.gv.at. The labelling algorithm extracts the geo-entity “Upper Austria” (an Austrian state) from the title and the publisher “Oberösterreich”. The extracted geo-entities are added as additional semantic information to the indexed resource.

4.2. Temporal labelling

Similarly to the geospatial cues, temporal information in Open Data comes in various forms and granularity, e.g., as datetime/time span information in the metadata indicating the validity of a dataset, or year/month/time information in CSV columns providing timestamps for data points or measurements.

4.2.1. Metadata descriptions

We extract the datasets’ temporal contexts from the metadata descriptions available at the data portals in two forms: (i) We extract the published and last modified information in case the portal provides dedicated metadata fields for these. (ii) We use the resource title, the resource description, the dataset title, the dataset description, and the keywords as further sources for temporal annotations. However, we prioritize the sources in the above order, meaning that we use the temporal information in the resource metadata rather than the information in the dataset title or description.22

The datetime extraction from titles and descriptions is based on the Heideltime framework [15] since this information typically comes as natural text. Heideltime supports extraction and normalization of temporal expressions for ten different languages. In case the data portal’s origin language is not supported we use English as a fallback option.

22For instance, consider a dataset titled “census data from 2000 to 2010” that holds several CSVs titled “census data 2000”, “census data 2001”, etc.: This metadata allows to infer that the temporal cues in the CSVs’ titles are more accurate/precise than the dataset’s title, which gives a more general time span for all CSVs.
4.2.2. CSVs

To extract potential datetime values from the datasets we parse the columns of the CSVs using the Python dateutil library.\(^{23}\) This library is able to parse a variety of commonly used date-time patterns (e.g., ‘‘January 1, 2047’’, ‘‘2012-01-19’’, etc.), however, we only allow values where the parsed year is in the range of 1900 and 2050.\(^{24}\)

For both sources of temporal information, i.e. metadata and CSV columns, we store the minimum and maximum (or start and end time) value so that we can allow range queries over the annotated data.

**Datetime periodicity patterns.** The algorithm in Figure 8 displays how we estimate any pattern of periodicity of the values in a column for a set of input datetime values. Initially, we check if all the values are the same (**static**), e.g., a column where all cells hold “2018”. Then we sort the values; however, note that this step could lead to unexpected annotations, because the underlying pattern might not be apparent in the unsorted column.

We compute all differences (**deltas**) between the input dates and check if all these deltas have approximately – with 10% margin – the same length. We distinguish daily, weekly, monthly, quarterly, and yearly pattern; in case of any other recurring pattern we return other.

```python
def datetime_pattern(dates):
    # all the dates have the same value
    if len(set(dates)) == 1:
        return 'static'

    # sort the dates and compute the deltas
    dates = sorted(dates)
    deltas = [(d - dates[i-1]) for i, d in enumerate(dates)[1:]]

    for p, l in [('daily', lambda: delta(days=1)),
                 ('weekly', lambda: delta(days=7)),
                 ('monthly', lambda: delta(days=30)),
                 ('quarterly', lambda: delta(days=91)),
                 ('yearly', lambda: delta(days=365))):
        if all(l - l * 0.1 < d < l + (l * 0.1) for d in deltas):
            return p

    # none of the pre-defined pattern
    if len(set(deltas)) == 1:
        return 'other'

    # values do not follow a regular pattern
    return 'various'
```

**Figure 8:** Python code fragment for estimating the datetime patterns of a set of values.

4.3. Indexed Datasets & Evaluation

Our framework currently contains CSV tables from 11 European data portals from 10 different countries, cf. Table 1. Note, that the notion of **datasets** on these data portals (wrt. Table 1) usually groups a set of resources; for instance, typically a dataset groups resources which provide the same content in different file formats. A detailed description and analysis of Open Data portals’ resources can be found in [13]. The statistics in Table 1, i.e. the number of datasets and indexed CSVs is based on the third week of March 2018. The differing numbers of **CSVs** and **indexed** documents in the table can be explained by offline resources, parsing errors, etc. Also, we currently do not index documents larger than 10MB due to local resource limitations; the basic setup (using Elasticsearch for the indexed CSVs, cf Section 6) is fully scalable.

Table 1: Indexed data portals

<table>
<thead>
<tr>
<th>portal</th>
<th>datasets</th>
<th>CSVs</th>
<th>indexed</th>
</tr>
</thead>
<tbody>
<tr>
<td>govdata.de</td>
<td>19464</td>
<td>10006</td>
<td>5646</td>
</tr>
<tr>
<td>data.gv.at</td>
<td>20799</td>
<td>18283</td>
<td>2791</td>
</tr>
<tr>
<td>offenedaten.de</td>
<td>28372</td>
<td>4961</td>
<td>2530</td>
</tr>
<tr>
<td>datos.gob.es</td>
<td>17132</td>
<td>8809</td>
<td>1275</td>
</tr>
<tr>
<td>data.gov.ie</td>
<td>6215</td>
<td>1194</td>
<td>884</td>
</tr>
<tr>
<td>data.overheid.nl</td>
<td>12283</td>
<td>1603</td>
<td>828</td>
</tr>
<tr>
<td>data.gov.uk</td>
<td>45153</td>
<td>7814</td>
<td>594</td>
</tr>
<tr>
<td>data.gov.gr</td>
<td>6648</td>
<td>414</td>
<td>496</td>
</tr>
<tr>
<td>data.gov.sk</td>
<td>1402</td>
<td>877</td>
<td>384</td>
</tr>
<tr>
<td><a href="http://www.data.gouv.fr">www.data.gouv.fr</a></td>
<td>28401</td>
<td>6038</td>
<td>258</td>
</tr>
<tr>
<td>opingogn.is</td>
<td>54</td>
<td>49</td>
<td>41</td>
</tr>
</tbody>
</table>

Table 2 lists the total number of annotated datasets. With respect to the spatial labelling algorithm, we were able to annotate columns of 3518 CSVs and metadata descriptions of 11231 CSVs (of a total of 15k indexed CSVs). Regarding the temporal labelling, we detected date/time information in 2822 CSV columns and in 9112 metadata descriptions.

Table 2: Total numbers of spatial and temporal annotations of metadata descriptions and columns.

<table>
<thead>
<tr>
<th>Spatial Columns</th>
<th>Metadata Counts</th>
<th>Temporal Columns</th>
<th>11231</th>
<th>2822</th>
<th>9112</th>
</tr>
</thead>
<tbody>
<tr>
<td>3518</td>
<td>9112</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Here we focus on evaluating the annotated geo-entities, and neglect the temporal annotations with the following two main reasons: First, the datetime detection over the CSV columns is based on the standard Python library `dateutil`. The library parses standard datetime formats (patterns such as `yyyy-mm-dd`, or `yyyy`) and the potential errors here are that we incorrectly classify a numerical column, e.g., classifying postal codes as years. As a very basic pre-processing, where we do not see a need for evaluation, we reduce the allowed values to the range 1900-2050 (with the drawback of potential false negatives), however, using the distribution of the numeric input values [16] would allow a more informed decision. Second, the labelling of metadata information is based on the temporal tagger Heideltime [15] which provides promising evaluations over several corpora.

**Manual inspection of a sample set.** To show the performance and limitations of our labelling approach we have randomly selected 10 datasets per portal (using Elasticsearch’s built-in random function\(^{25}\)) and from these again randomly select 10 rows, which resulted in a total of 101 inspected CSVs, i.e. 1010 rows (with up to several dozen columns per CSV). As for the main findings, in the following let us provide a short summary; all selected datasets and their assigned labels can be found at [https://github.com/sebneu/geolabelling/tree/eu-data/jws_evaluation](https://github.com/sebneu/geolabelling/tree/eu-data/jws_evaluation).

Initially, we have to state that this evaluation is manually done by the authors and therefore restricted to our knowledge of the data portals’ origin countries and their respective language, regions, sub-regions, postal codes, etc. For instance, we were able to see that our algorithm correctly labelled the Greek postal codes in some of the test samples from the Greek data portal [data.gov.gr],\(^{26}\) but that we could not assign the corresponding regions and streets.\(^{27}\) However, as we are not able to read and understand the Greek language (and the same for the other non-English/German/Spanish portals) we cannot fully guarantee any potential mismatches or missing annotations that we did not spot during our manual inspections.

We categorize the datasets’ labels by assessing the following dimensions: are there any correctly assigned labels in the dataset (\(c\)), are there any missing annotations (\(m\)), and did the algorithm assign incorrect links to GeoNames (\(g\)) or OSM (\(o\)); a result overview is given in Table 3.

<table>
<thead>
<tr>
<th></th>
<th>c</th>
<th>m</th>
<th>g</th>
<th>o</th>
</tr>
</thead>
<tbody>
<tr>
<td>total</td>
<td>101</td>
<td>87</td>
<td>37</td>
<td>9</td>
</tr>
</tbody>
</table>

Table 3: Evaluation result of sample CSVs.

Out of 101 inspected datasets we identified in 87 CSVs correct annotations. In particular, for the Spain and the Greek data portal only in 50% of the test samples there were correct links, while for the 9 other indexed data portals we have a near to 100% rate. Regarding any missing annotations, we identified 37 datasets where our algorithm (and also the completeness of our spatial knowledge graph) needs improvements. For instance, in some datasets from the Netherlands’ data portal\(^{28}\) and also the Slovakian portal\(^{29}\) we identified street names and addresses that could potentially mapped to OSM entries.

Regarding incorrect links there were only 9 files with wrong GeoNames and 9 files with wrong OSM annotations. An exemplary error that we observed here was that some file\(^{30}\) contains a column with the value “Norwegen” (“Norway”): Since the file is provided at a German data portal, we incorrectly labelled the column using a small German region Norwegen instead of the country, because our algorithm prefers labels from the origin country of the dataset. Another example that we want to consider in future versions of our labelling algorithm is this wrong assignment of postal codes:\(^{31}\) We incorrectly annotated a numeric column with the provinces of Spain (which use two-digit numbers as postal codes).

---


\(^{26}\) E.g., [https://github.com/sebneu/geolabelling/blob/eu-data/jws_evaluation/data_gov_gr/0.csv](https://github.com/sebneu/geolabelling/blob/eu-data/jws_evaluation/data_gov_gr/0.csv), the datasets use “T.K.” in the headers to indicate these codes.

\(^{27}\) The Greek data portal uses the Greek letters in their metadata and CSVs which would require a specialized label mapping wrt. lower-case mappings, stemming, etc.

\(^{28}\) E.g., [https://github.com/sebneu/geolabelling/tree/eu-data/jws_evaluation/data_overheid_ml/1.csv](https://github.com/sebneu/geolabelling/tree/eu-data/jws_evaluation/data_overheid_ml/1.csv)


\(^{30}\) [https://github.com/sebneu/geolabelling/blob/eu-data/jws_evaluation/offenedaten_de/0.csv](https://github.com/sebneu/geolabelling/blob/eu-data/jws_evaluation/offenedaten_de/0.csv)

\(^{31}\) [https://github.com/sebneu/geolabelling/blob/eu-data/jws_evaluation/datos_gob_es/7.csv](https://github.com/sebneu/geolabelling/blob/eu-data/jws_evaluation/datos_gob_es/7.csv)
5. Export RDF

We make our base knowledge graph and RDFized linked data points from the CSVs available via a SPARQL endpoint. Figure 9 displays an example extract of the RDF export of the knowledge graph. The sources of the aggregated links between the different entities and literals in our graph are indicated in the figure; we re-use the GeoNames ontology (gn:) for the hierarchical enrichments generated by our algorithms (see links [m]), and owl:sameAs for mappings to OSM relations and NUTS regions, cf. Figure 9.

Annotated data points. We export the linked data points from the CSVs in two ways: First, for any linked geo-entity <g> in our base knowledge graph, we add triples for datapoints uniquely linked in CSV resources (that is, values appearing in particular columns) by the following triple schema: if the entity <g> appears in a column in the given CSV dataset, i.e., the value VALUE in that column has been labeled with <g>, we add a triple of the form <g> <u#col> "VALUE" .

That is, we mint URIs for each column col appearing in a CSV accessible through a URL u by the schema u#col, i.e., using fragment identifiers. The column’s name col is either the column header (if a header is available and the result is a valid URI) or a generic header using the columns’ index column1, column2, etc. These triples are coarse grained, i.e. they do not refer to a specific row in the data. We chose this representation to enable easy-to-write, concise SPARQL queries like for instance:

```sparql
SELECT ?geo ?value
WHERE {
}
```

The above query selects all values and their geo-annotations for the selected column named “Postal CODE” in a specific dataset about dog breeds in Vienna per district.32

Second, a finer grained representation, which we also expose, provides exact table cells where a certain geospatial entity is linked to, using an extension of the CSVW vocabulary [17]: note that the CSVW vocabulary itself does not provide means to conveniently annotated table cells in column col and row n which is what we need here, so we define our own vocabulary extension for this purpose (for the moment, under the namespace http://data.wu.ac.at/ns/csvwx#):

```sparql
@prefix csvw: <http://data.wu.ac.at/ns/csvw#> .
@prefix csvwx: <http://data.wu.ac.at/ns/csvwx#> .

<#col> csvwx:cell [ a csvw:Cell ; csvwx:rownum n ; csvwx:rowURL <#row> ;
                  rdf:value "VALUE" ;
                  csvwx:refersToEntity <g> ] .
```

We use the CSVW class csvw:Cell for an annotated cell and add the row number and value (using csvw:rownum and rdf:value) to it. We extend CSVW by the property csvw:cell to refer from a csvw:Column (using again the fragmented identifier u#col) to a specific cell, and the property csvwx:rowURL to refer to the CSV’s row (using the schema u#row=n). Here, the property csvwx:refersToEntity connects the labelled geo-entity <g> to a specific cell.

Analogously, in case of available (labelled) temporal information for a cell, we use the property csvwx:hasTime:

```sparql
@prefix csvw: <http://data.wu.ac.at/ns/csvw#> .
@prefix csvwx: <http://data.wu.ac.at/ns/csvwx#> .

<#col> csvwx:cell [ a csvw:Cell ; csvwx:rownum n ; csvwx:rowURL <#row> ;
                  rdf:value "VALUE" ; csvwx:hasTime "DATE"^^xsd:dateTime .
                  ] .
```

Moreover, we denote the geo-spatial scope of the column itself by declaring the type of entities within which geographic scope appearing in the column. The idea here is that we annotate – on column level – the least common ancestor of the spatial entities recognized in cells within this column. E.g.,

```sparql
<#col> csvwx:refersToEntitiesWithin <g1> .
```

with the semantics that the entities linked to col in the CSV u all refer to entities within the entity g1 (such that g1 is the least common ancestor in our knowledge graph.

This could be seen as a shortcut/materialization for a CONSTRUCT query as follows:

```sparql
CONSTRUCT { ?UrlCol csvwx:refersToEntitiesWithin ?G_1 .}
WHERE {
  ?Col csvwx:cell [ csvwx:refersToEntity ?G_1 .]
  ?G_1 gn:parentFeature* ?G_1
  FILTER NOT EXISTS { ?UrlCol csvwx:refersToEntitiesWithin ?G_1 .
    FILTER NOT EXISTS {
    }
  }
}
```

32Link to the query: https://tinyurl.com/y897rwr1
Obviously, this query is very inefficient and we rather compute these least common ancestors per column during labeling/indexing of each column.

**CSV on the Web.** In order to complete the descriptions of our annotations in our RDF export, we describe all CSV resources gathered from the annotated Open Data Portals and their columns using the CSV on the Web (CSVW) vocabulary, re-using the following parts of the CSVW schema.

Firstly, we use the following scheme to connect our aforementioned annotations to the datasets:

```sparql
@prefix csvw: <http://www.w3.org/ns/csvw#> .
@prefix dcat: <http://www.w3.org/ns/dcat#> .

d a dcat:Dataset [ dcat:hasURL u ;
    csvw:tableSchema [ csvw:columns [ u#col1 ; u#col2 ; ... u#coln ] ] ] .

d dcat:Dataset [ dcat:hasURL u ;
    csvw:tableSchema [ csvw:columns [ u#col1 ; u#col2 ; ... u#coln ] ] ] .

<http://example.com/dataset1> a dcat:Dataset .
<http://example.com/dataset2> a dcat:Dataset .
```

Then, we enrich this skeleton with further CSVW annotations that we can extract automatically from the respective CSV files. Figure 10 displays an example export for a CSV resource. The blank node _:_csv represents the CSV resource which can be downloaded at the URL at property csvw:url.

The values of the properties dcat:byteSize and dcat:mediaType are values of the corresponding HTTP header fields. The dialect description of the CSV can be found via the blank node _:_dialect at property csvw:dialect and the columns of the CSV are connected to the _:_schema blank node (describing the csvw:tableSchema of the CSV).

```sparql
<http://example.com/dataset1> a dcat:Dataset .
```

**6. Search & Query Interface**

Our integrated prototype for a spatio-temporal search and query system for Open Data currently consists of three main parts: First, the geo-entities DB and search engine in the back end, second the user interface and APIs, and third, access to the above described RDF exports via an SPARQL endpoint.

**6.1. Back End**

All labels from all the integrated datasets and their corresponding geo-entities are stored in a
look-up store, where we use the NoSQL key-value database MongoDB. It allows an easy integration of heterogeneous data sources and very performant look-ups of keys (e.g., labels, GeoNames IDs, postal codes, etc. in our case).

Further, we use Elasticsearch to store and index the processed CSVs and their metadata descriptions. In our setup, an Elasticsearch document corresponds to an indexed CSV and consists of all cell values of the table (arranged by columns), the potential geo-labels for a labelled column, metadata of the CSV (e.g., the data portal, title, publisher, etc.), the temporal annotations, and any additional labels extracted from the metadata.

6.2. User interface

The user interface, available at http://data.wu.ac.at/odgraphsearch/, allows search queries for geo-entities but also full-text matches. Note, that the current UI implements geo-entity search using auto-completion of the input (but only suggesting entries with existing datasets) and supports full-text querying by using the “Enter”-key in the input form. The screenshot in Figure 11 displays an example query for the Austrian city “Linz”. The green highlighted cells in the rows below show the annotated labels, for instance, the annotated NUTS2 code “AT31” in the second result in Figure 11. Likewise we allow to filtering datasets relevant to a particular period either by auto-completion in a separate field to filter the time period by a period/event label, or by choosing start and end dates via sliders (cf. Figure 11).

The chosen geo-entities and durations which are fixed via these lookups in our search index through the UI are passed on as parameters as a concrete geo-ID and/or start&end-date to our API, which we describe next.

Additionally, the web interface provides APIs to retrieve the search results, all indexed datasets, and the RDF export per dataset. To allow programmatic access to the search UI we offer the following HTTP GET API:

```
/locations/search?l={GeoIDs} &limit={limit} &offset={offset} &mstart={startDate} &mend={endDate} &start={startDate}&end={endDate} &periodicity={dateTimePattern} &q={keyword}
```

The API takes multiple instances of geo IDs, that is, GeoNames or OSM IDs (formatted as osm:{ID}) using parameter l, a limit and an offset parameter, which restricts the amount of items (datasets) returned, and an optional white space separated list of keywords (q) as full-text query parameter to enable conventional keyword search in the metadata and header information of the datasets. To restrict the results to a specific temporal range we implemented the parameters mstart, mend (for filtering a time range from the metadata-information), and start, end (for the min and max values of date annotations from CSV columns). The parameter periodicity allows to filter for datetime periodicity patterns such as “yearly”, “monthly”, or “static” (in case there is only a single datetime value in this column), cf. Section 4.2.2 for a detailed description of the periodicity patterns.

The output consists of a JSON list of documents that contain the requested GeoNames/OSM IDs or any tables matching the input keywords.

6.3. SPARQL endpoint

We offer a SPARQL endpoint at http://data.wu.ac.at/odgraphsearch/sparql where we provide the data as described in Section 5. Currently, as of the first week of April 2018, the endpoint contains 88 million triples: 15 million hierarchical relations using the gn:parentFeature
relation, 11768 CSVs (together with their CSV on the Web descriptions), where we added a total of 5 million geo-annotations using the csvwx:refersToEntity property, and 1.3 million datetime-annotations using the csvwx:hasTime annotation.

Example queries. The first example lists all datasets from Vienna, using the csvwx:refersToEntity metadata annotation, and only lists CSVs where there exists a column with dates within the range of the last Austrian legislative period, using the Wikidata entities of the past two elections:

```sql
SELECT ?d ?url WHERE {
  ?d dcat:hasTime ?start.
  ?d dcat:hasTime ?end.
  # find another cell in the same row where the 
  # geo-entity has the spatial coverage area of 
  # the found period as the parent country
  ?col1 csvv:cell {
    csvv:rownum ?rownum; 
    csvv:refersToEntity {
      gn:parentCountry ?spatial
    }
  }.
  FILTER((?Time >= ?start) && (?Time <= ?end))
}
```

GeoSPARQL. GeoSPARQL [18] extends SPARQL to a geographic query language for RDF data. It defines a small ontology to represent geometries (i.e., points, polygons, etc.) and connections between spatial regions (e.g., contains, part-of, intersects), as well as a set of SPARQL functions to test such relationships. The example query in Figure 12 uses the available polygon of the Viennese district “Leopoldstadt” to filter all annotated data points within the borders of this district.

While we do not yet offer a full GeoSPARQL endpoint for our prototype yet (which we leave to a forthcoming next release), our RDFized datasets and knowledge graph is GeoSPARQL “ready”, i.e. having all the geo-coordinates and polygons in the endpoint using the GeoSPARQL vocabulary; an external GeoSPARQL endpoint could already access our data using the SERVICE keyword and evaluate the GeoSPARQL specific functions locally, or simply import our data.

7. Related Work

The 2013 study by Janowicz et al. [19] gives an overview of Semantic Web approaches and technologies in the geo-spatial domain. Among the in the article listed Linked Data repositories and ontologies we also find the GeoNames ontology (cf. Section 2), the W3C Geospatial Ontologies [20], and the GeoSPARQL Schemas [18]. However, when looking into the paper’s listed repositories, most of them (6/7) were not available, i.e. offline, which seems to indicate that many efforts around GeoLinked data have unfortunately not been pursued in a sustainable manner.

The 2012 project LinkedGeoData [21] resulted in a Linked Data resource, generated by converting a subset of OpenStreetMap data to RDF and deriving a lightweight ontology from it. In [22] the authors describe their attempts to further connect GeoNames and LinkedGeoData, using string
Figure 12: Example GeoSPARQL query over using the available geometries – not yet available via the endpoint.

similarity measures and geometry matching. However, while LinkedGeoData is also listed in [19] as a geospatial Linked Data repository, unfortunately it is currently not available online. Also, this work was complementary to ours, as we do not focus on matching and entity alignment, but rather the integration of existing structured entities from different Geo and Temporal (Linked) Data sources. The GeoKnow project [23] is another attempt to provide and manage geospatial data as Linked Data. GeoKnow provides a huge toolset to process these datasets, including the storage, authoring, interlinking, and geospatially-enabled query optimization techniques.

The project PlanetData (2010 to 2014), funded by the European Commission, released an RDF mapping of the NUTS classifications34 [24] using the GeoVocab vocabulary.35 This dataset models the hierarchical relations of the regions, provides labels and polygons. Unfortunately, the project does not include external links to GeoNames, or Wikidata, except for the country level, i.e. there are only 28 links to the corresponding GeoNames entries of the EU member states.

Complementary to our approach to access street addresses via OSM, Open Addresses36 is a global collection of address data sources, which could be considered for future work as an additional dataset to feed into our base knowledge graph. The manually collected and homogenized dataset consists of a total of 478M addresses; street names, house numbers, and postal codes combined with geographic coordinates, harvested from governmental datasets of the respective countries.

A conceptually related approach, although not focusing on geo-data, is the work by Taheriyyan et al. [25], who learn the semantic description of a new source given a set of known semantic descriptions as the training set and the domain ontology as the background knowledge.

In [26] Paulheim provides a comprehensive survey of refinement methods, i.e., methods that try to infer and add missing data to a graph, however, these approaches work on graphs in a domain-independent setting and do not focus on temporal and spatial knowledge. Still, in some sense, we view our methodology of systematic Knowledge Graph aggregation from Linked Data sources via declarative, use-case specific, minimal mappings as potentially complementary to the domain-independent methods mentioned therein. I.e., we think in future works, such methods should be explored in combination.

Most related wrt. the construction of the temporal knowledge graph is the work by Gottschalk and Demidova [27] (2018): they present a temporal knowledge graph that integrates and harmonizes event-centric and temporal information regarding historical and contemporary events. In contrast to [27] we also integrate data from PeriodO [9], and we focus on periods in a geospatial context. This work is built upon [28] where the authors extract event information from the Wikipedia Current Events Portal (WCEP). In future work we want to

---

34 http://nuts.geovocab.org/, last accessed 2018-01-05
35 http://geovocab.org/, last accessed 2018-01-05
36 https://openaddresses.io/, last accessed 2018-04-01
connect the resource from [27], since the additional data extracted from the WCEP and WikiTimes interface is in particular interesting for our framework.

In [10] Rospocher et al. build a knowledge graph directly from news articles, and in [29] by extracting event-centric data from Wikipedia articles. These approaches work over plain text (with the potential drawback of noisy data) while we integrate existing structured sources of temporal information; therefore these are complementary/groundwork to our contributions.

8. Conclusions

Governmental data portals such as Austria’s data.gv.at or the UK’s data.gov.uk release local, regional and national data to a variety of users (citizens, businesses, academics, civil servants, etc.). As this data is mainly collected as part of census collections, infrastructure assessments or any other, secondary output data, these resources almost always contain or refer to some kind of geographic and temporal information; for instance, think of public transport data, results of past elections, demographic indicators, etc. Search across these dimensions seems therefore natural, i.e., we have identified the spatial and temporal dimensions as the crucial, characterizing dimensions of datasets on such data portals.

In order enable such search and to integrate these datasets in the LOD cloud (as they are mainly published as CSVs [13]) we have achieved the following tasks in this work:

- We have described a hierarchical knowledge graph of spatial and temporal entities in terms of SPARQL queries, as well as the integration of temporal information and its interlinkage with the geospatial-knowledge from various Linked data sources (GeoNames, OSM, Wikidata, PerioDo), where our general approach is extensible to adding new sources, further details of the construction are provided in the appendix.

- To demonstrate the performance and limitations of our spatio-temporal labelling we have evaluated the annotations by manual inspection of a random sample per data portal, where we identified correct geo-annotations for around 90% of the inspected datasets.

- To access and query the data we offer a user interface, RESTful APIs and a SPARQL endpoint, which allows structured queries over our spatio-temporal annotations.

To the best of our knowledge, this is the first work addressing a spatial-temporal labelling and allowing structured spatio-temporal search of datasets based on a knowledge graphs of temporal and geo-entities.

To further improve geo-labelling, we are currently working on parsing coordinates in datasets. To do so we have to consider that the long/lat pairs potentially come in column groups, i.e., one column per coordinate, which we might need to combine. Having all the geometries for the geo-entities and data points, we want to integrate a visual representation and search interface for datasets by displaying and filtering the datasets/records on a map.

While CSV is a popular and dominant data-publishing format on the Web [13], we also want to extend our indexing to other popular Open Data formats (such as XLS and JSON). Additionally, we want to test how well our approaches could be applied to unstructured or semi-structured data and other domains such as tweets or web pages (e.g., newspaper articles), or complementarily, we could use our knowledge graph along with known methods for temporal and geo-labelling of such unstructured sources link them to (supporting) Data, to enable for instance fact checking. The applications of Open Data sources searchable and annotated in such a manner seem promising and widespread.

References

[2] M. B. Gurstein, Open data: Empowering the empowered or effective data use for everyone?, First Monday 16 (2).
in open data portals using the analytic hierarchy process, Government Information Quarterly. doi:https://doi.org/10.1016/j.giq.2017.11.003.


URL https://www.iso.org/standard/63545.html


URL http://doi.acm.org/10.1145/2629489


Appendix A. Realizing the Queries from Section 3

As mentioned earlier, we extract the relevant RDF Data for constructing our knowledge graph from different Linked Data Sources, which provide RDF\(^{37}\) data either in the form of dumps or via SPARQL endpoints, where we presented the respective SPARQL queries that theoretically should suffice to extract the data relevant for us in Section 3. A common problem with these sources is however that either such a SPARQL endpoint is not available or does not support complex queries. To this end, we discuss in this appendix how such limitations could be circumvented in the specific cases. We note that we expect the presented workaround could be similarly applied to other use cases for extracting relevant data from large RDF dumps or public endpoints, so we hope the discussion herein might be useful also for others.

Appendix A.1. Extracting postal codes and NUTS identifier from Wikidata

Due to the fact that the query in Figure 2 resulted in timeouts at the Wikidata SPARQL endpoint we split the query in sub-queries.\(^{38}\) The task of extracting the NUTS identifier provides mappings for 1316 (out of 1716) NUTS codes. The missing 400 codes are NUTS regions where no Wikidata and/or GeoNames entry exists because, strictly speaking, there is no such corresponding administrative region. For instance, the Austrian NUTS regions AT126 and AT127 are called “Wiener Umland/Nordteil” and “Wiener Umland/Südteil”, however, these are no political districts, but statistical entities grouping a set of districts Wikidata/GeoNames entity to map.

To complement the set of postal codes in Wikidata we use the extra postal code dataset by GeoNames\(^ {39}\) which consists of a total of 1.1M entries from 84 countries. For each code it provides a place name, and (depending on the country) several parent region/subdivision names. Based on these names we use a simple heuristic to map the postal codes to GeoNames entities: We split place names in the dataset by separators (white spaces, “,”, “/”)\(^ {40}\) and try to find GeoNames entries, in the respective country, with matching names.

Appendix A.2. Extracting Spatial Data from OSM

Since there exists – to the best of our knowledge – no available and integrated linked data version of OSM, we extract OSM relations, ways and nodes and map these to our spatial knowledge graph. To do so we perform the following steps on a local extract of OSM:\(^ {41}\)

1. OSM provides different administrative levels for their relations, e.g., the relation which represents the states of a country, their subdivisions, and so on.\(^ {42}\) We use the alignment of these administrative levels with the previously introduced NUTS identifier to add the mappings to GeoNames: We perform lookups with the GeoNames labels of the NUTS 1, 2, and 3 regions at OSM’s Nominatim service.\(^ {43}\) This service returns a set of potential candidate OSM relations for a given label. We select the correct relation (i.e. OSM region) by choosing the OSM relation at the same administrative/NUTS level as the corresponding GeoNames region.

2. Having the mapping for the countries’ regions we again use OSM Nominatim to get the polygons for all sub-regions. These polygons can be

\(^{37}\)We note OSM here as an exception: the JSON-data we extract from OSM is not directly in an RDF serialization, but the provided JSON can be easily converted to JSON-LD.

\(^{38}\)SELECT ?s ?nutes ?geonames WHERE {
{?s wdt:P605 ?nutes.
?s wdt:P1566 ?geonames} to get the NUTS-to-GeoNames mappings. Similarly for the postal code property wdt:P281.

\(^{39}\)http://download.geonames.org/export/zip/, last accessed 2018-03-28

\(^{40}\)We add this preprocessing step because there are many translated place names separated by slash or comma.

\(^{41}\)http://nominatim.openstreetmap.org

\(^{42}\)http://wiki.openstreetmap.org/wiki/Tag:boundary\%3Dadministrative

\(^{43}\)http://download.geofabrik.de/
used to extract any street names, places, etc. from an OSM data extract.\footnote{OSM provides a tool, Osmosis \url{http://wiki.openstreetmap.org/wiki/Osmosis}, to process polygons on OSM data dumps}

The OSM polygons returned by OSM’s Nominatim service in Item 2 are not available as RDF, so we try to interpret the JSON from Nominatim as JSON-LD. This could be done relatively straightforwardly by adding to the JSON you get by e.g. calling \url{https://nominatim.openstreetmap.org/reverse?osm_id=1990594&osm_type=R&polygon_geojson=1&format=json} for obtaining the data for OSM id 1990594 (i.e. Vienna’s district “Leopoldstadt”, and extending the returned JSON with a JSON-LD \footnote{There is ongoing work to fix it, which, however points to the same problem as an outstanding issue, cf. \url{https://github.com/json-ld/json-ld.org/issues/397}, retrieved 2018-03-29.} context:

```
@context: {
  "@vocab": "https://data.vu.ac.at/ns/osm#"
}
```

However, the query from Figure 3 still would not work “as is”, since OSM returns the coordinates of its entities as GeoJSON \footnote{https://www.wikidata.org/wiki/Wikidata:Database_download}, which due to the way that GeoJSON represents geometries as nested JSON arrays, is incompatible with JSON-LD.\footnote{We therefore pre-convert, GeoJSONs nested way of representing polygon’s to the format compatible with GeoSPARQL \footnote{https://query.wikidata.org/bigdata/ldf}, by replacing JSON attributes of the form:

```
"geojson": {
  "type": "polygon",
  "coordinates":
    [[[lat_1,long_1], ... , [lat_n,long_n]]]
}
```

with:

```
"geojson": {
  "type": "Polygon",
  "coordinates":
    "POLYGON((lat_1 long_1, ... , lat_n long_n))"
}
```

and extend the context to:

```
@context: {
  "@vocab": "http://www.opengis.net/ont/geosparql#wktLiteral"
}
```

in a simple pre-processing step. The query in Figure 3 works as expected on this respectively pre-processed data from Nominatim.

\subsection*{Appendix A.3. Extracting Temporal Data from Wikidata}

The query to extract event and time period data from Wikidata is shown in Figure 4; however as mentioned above, this query times out on the public endpoint. We note that Wikidata contained (at the time of writing) 4.8b RDF triples, so retrieving a dump and trying to extract the relevant information by setting up a local SPARQL endpoint also didn’t seem an attractive solution. Rather, we propose a combination of

1. extracting relevant triples to answer the query via HDT \footnote{\url{https://www.w3.org/Submission/HDT/}}

2. executing targeted \texttt{CONSTRUCT} queries to the full SPARQL endpoint for specific sub-queries in order to materialize path expressions.

As for Item 1, we downloaded the complete Wikidata dump,\footnote{https://www.wikidata.org/wiki/Wikidata:Database_download} converted it locally to HDT \footnote{https://github.com/json-ld/json-ld.org/issues/397} and executed the following triple pattern queries over it to collect all data to match non-property-path triple patterns in Figure 4. We note that alternatively, we could have used Wikidata’s Triple Pattern Fragment API \footnote{https://query.wikidata.org/bigdata/ldf} at \url{https://query.wikidata.org/bigdata/ldf} similarly.

We then executed the following extraction queries separately on the dump, to extract the necessary component data:

\begin{verbatim}
CONSTRUCT WHERE { ?S wp:P17 ?O } \to 6613664 triples
CONSTRUCT WHERE { ?S wp:P131 ?O } \to 3928939 triples
CONSTRUCT WHERE { ?S wp:P276 ?O } \to 697238 triples
CONSTRUCT WHERE { ?S wp:P580 ?O } \to 26354 triples
CONSTRUCT WHERE { ?S wp:P582 ?O } \to 19241 triples
CONSTRUCT WHERE { ?S wp:P585 ?O } \to 91509 triples
CONSTRUCT WHERE { ?S wp:P625 ?O } \to 4158225 triples
\end{verbatim}

In order to retrieve the remaining triples, that is instances of (subclasses of) the Wikidata classes of elections (\wdb{Q40231}) and sports competitions (\wdb{Q13406554}) we executed the following queries against the Wikidata SPARQL endpoint:

\begin{verbatim}
CONSTRUCT { ?S a \wdb{Q13406554}. ?S rdfs:label ?label. } WHERE { ?S \wdd{P31} \wdd{P279} \wdb{Q13406554}. ?S rdfs:label ?label. FILTER( LANG(?label) = "en" || LANG(?label) = "de" || LANG(?label) = "" ) } \to 418136 triples
\end{verbatim}

\begin{verbatim}
CONSTRUCT { ?S a \wdb{Q40231}. ?S rdfs:label ?label. } WHERE { ?S \wdd{P31} \wdd{P279} \wdb{Q40231}. ?S rdfs:label ?label. FILTER( LANG(?label) = "en" || LANG(?label) = "de" || LANG(?label) = "" ) } \to 6613664 triples
\end{verbatim}
We then loaded these triples into a local triple store and executed the following query on it, which is equivalent to Figure 4 (namespaces same as above):

```
CONSTRUCT {
  ?event rdfs:label ?label ;
  dcterms:isPartOf ?Parent ;
  times:hasStartTime ?StartDateTime ;
  times:hasEndTime ?EndDateTime ;
  dcterms:coverage ?geocoordinates ;
  dcterms:spatial ?geoentity .
} WHERE {
  { # with a point in time or start end end date
      ?event wdt:P585 ?StartDateTime .
      FILTER(?StartDateTime > "1900-01-01T00:00:00"^^xsd:dateTime)
    } UNION
  { ?event wdt:P580 ?StartDateTime .
    FILTER(?StartDateTime > "1900-01-01T00:00:00"^^xsd:dateTime)
      ?event wdt:P582 ?EndDateT .
      FILTER(DATATYPE(?EndDateT) = xsd:dateTime)
  }
  OPTIONAL { ?event wdt:P361 ?Parent. }
  # specific spatialCoverage if available
  OPTIONAL {
  }
  OPTIONAL {
    ?event wdt:P2767/wdt:P625 ?geocoordinates
  }
  BIND ( if(bound(?EndDateT), ?EndDateT, xsd:dateTime(concat(str(xsd:dateTime(?StartDateTime)), "T23:59:59"))) AS ?EndDateTime )
}
```