FREIE UNIVERSITÄT BERLIN

Assisting in semantic enrichment of scholarly resources by connecting `neonion` and Wikidata

Jakob Höper, Claudia Müller-Birn

TR-B-18-03
2018

FACHBEREICH MATHEMATIK UND INFORMATIK
SERIE B • INFORMATIK

Abstract

Explicit semantic enrichments make digital scholarly publications potentially easy to find, to navigate, to organize and to understand. But whereas the generation of explicit semantic information is common in fields like biomedical research, comparable approaches are rare for domains in the humanities. Apart from a lack of authoritive structured knowledge bases formalizing the respective conceptualizations and terminologies, many experts from specialized fields of research seem reluctant to employ the technologies and methods that are currently available for the generation of structured knowledge representations. However, human involvement is indispensable in the organization and application of the domain-specific knowledge representations necessary for the contextualization of structured semantic data extracted from textual and scholarly resources.

Over the past decade, various efforts have been made towards openly accessible online knowledge graphs containing collaboratively edited, structured and cross-linked data. Such public knowledge bases might be suitable as a starting point for defining formalized domain knowledge representations, with which the subjects and findings of a research domain can be described. Extensive re-use of the widely adopted shared conceptualizations from a large collaborative knowledge base could be in more than one way beneficial to processes of semantic enrichment, especially those involving domain experts with less-technical backgrounds.

In this work, we discuss ways of enabling domain experts to semantically enrich their research resources by generating semantic annotations in text documents using the scholarly reading and annotation software `neonion`. We introduce features to the web-based software which improve various aspects of the semantic annotation process by connecting it to the collaboratively edited public knowledge base Wikidata. Furthermore, we argue that the re-use of external structured knowledge from Wikidata both fuels an enhanced workflow for assisted subject-matter-sensitive semantic annotation, and allows for the knowledge base to benefit from the structured data generated within `neonion` in return. Our prototype implementation extracts schematic terminological information from Wikidata objects linked by local annotations and feeds it into the new recommender system, where candidate descriptors for vocabulary amendment are being determined, most notably by the association rule mining recommender engine *Snoopy*.

This paper is a follow-up on the bachelor's thesis *"Assisting in semantic development of knowledge domains by recommending terminology"*, submitted by Jakob Höper under supervision by Prof. Dr. Claudia Müller-Birn. It elaborates in further detail on aspects of the presented implementation that have not been exhaustively covered in said thesis.

1 Introduction and problem statement

Although most publisher's digital publications are basically facsimilated counterparts of the printed version, with the PDF as a prevalent file format, vari-

ous suggestions for alternative forms of scholarly digital online publishing have been made over the past years. Apart from the desire to simply keep up with the technical progress by actually making use of the many arising possibilities, proposing new forms of digital publishing has been motivated by the necessity to keep track of, organize, and make use of an ever-growing amount of published scientific content.

The most frequently acknowledged issue is that scholars typically face a much too high number of new publications in their respective field of study, for them to be able to accomplish even the necessary amount of reading [17]. Many researchers and publishers expect or even already observe a deterioration of scientific standards, arguing that the increasing throughput leads to declining review process quality, and sloppy or fictatious citations [21, 19][1].

Conventional methods for information retrieval are based on the extraction of syntactical features of documents and weighing them by statistical and distributional measures. Due to inherent properties of natural languages, these approaches do not always provide satisfactory results.

Enriching digital publications with explicit structured representations of the core concepts occuring in their content, on the other hand, enables machines to make sense of embedded information in a way similar to a human recipient. Annotated with machine-readible representations of conceptualizations shared among publishing researchers, documents can be searched and navigated with much less noise involved [5]. This way, users get search results that are more useful to them, easier to discriminate amongst, and quicker to skim for their core information.

The past decades have seen huge progress in automated extraction of implicit semantic information from unstructured text resources, but most approaches require extensive configuration, large corpora of training data or close human supervision, and might not fit applications where these requirements can't be met [16].

Manual extraction of explicit semantic information by classification and identification of named entities, concepts, and relations of core interest to a publication's author might be a viable alternative to automated natural language processing in specified fields of research where a configuration applicable to the knowledge domain has yet to be set up. This is a potentially labour-intensive task, which furthermore might require more technical skill than anyone with the appropriate subject-matter expertise is willing or able to master. In fact, domain knowledge experts without a technical background are known to be hesitant about utilization of appropriate tools and technologies for the generation or assessment of semantic enrichment of scholarly documents [4].

[1] For instance, as pointed out in [19], whereas print journals can only publish as many submissions as affordable with their alotment for printing, binding and shipping, the overall quality of online journals has been said to be perceived as lower, as they are able to publish way more articles at the same cost. The assumption of less quality control in *e-journals*, on the other hand, would indeed cause them to receive poorer submissions, and in lower numbers.

2 User-centered semantic enhancement workflow

We present an approach which we expect to lower the hurdle that researchers have to take in order to be able to create semantic enrichments of their published research. To this end, a semi-automated workflow is being demonstrated which assists in refining a controlled vocabulary and in manually creating semantic annotations, while it at the same time hides the complexity of the underlying knowledge organization systems. Instead of letting researchers configure their domain-specific knowledge representations themselves, our approach integrates the support of recommender systems in order to provide users with meaningful vocabularies fitting the context of their research documents.

As a use case, unstructured textual documents from the open access online journal Apparatus are being used as an example for highly specialized academic research publications from the humanities. We developed tooling for the access and re-use of external knowledge representations from the linked data hub knowledge base Wikidata [6]. This enables us to assist users with intuitive means for contextualization, conceptualization and publication of the structured data they create as semantic enhancements of their textual resources.

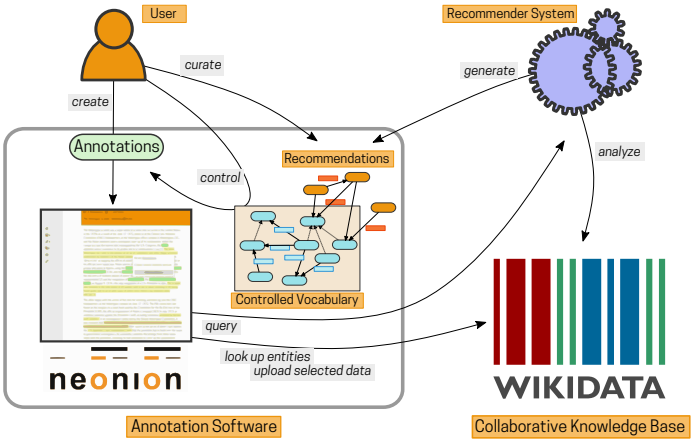

Figure 1: Integrating semi-automatic vocabulary alignment in `neonion` using external linked data services

Realization of the workflow proposed in this research was executed in several consecutive phases. Beginning with the implementation of a Wikidata client library for communication with Wikidata via multiple API endpoints and including the automated import of the actual journal articles meant for semantic enrichment into Wikidata, the development of tooling and integrated functionality needed for supporting the proposed workflow required a variety of extensions and modifications to be applied to the existing `neonion` codebase.

The following sections elaborate on different phases of development. Section 3 illustrates the pipeline for retrieval of journal metadata and their ingestion into Wikidata. In subsection 4.1, a general overview will be given on the underlying architecture of the original `neonion` implementation, in order to provide background information for further development. The section ends with a brief introduction of the three main categories under which all development towards Wikidata integration in `neonion` falls.

3 Import of research resources into Wikidata

In order for the proposed workflow to support contribution of bibliographic reference to matchable Wikidata statements, import of Apparatus articles into Wikidata is desirable for multiple reasons, including the implicit objective followed by the Wikidata community to build an "organically growing hub" that links different authority files [13].

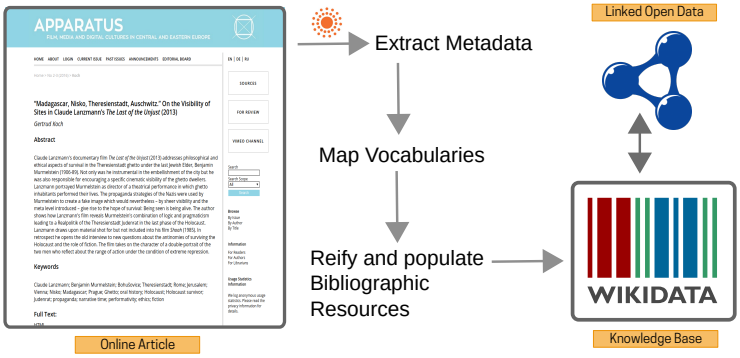

Figure 2: Illustration of the automated process designed for import of online journal articles into Wikidata

With the creation and population of a Wikidata item page holding an article's bibliographic information, it becomes a resource available via the Web of Data. The general import process is illustrated in Figure 2. Without a Wikidata item representing an article as a bibliographic resource, we could still use it as a bibliographic reference for Wikidata statements by using property *reference URL* (`P854`[2]), but neither would we be able to make statements about the article itself, nor could such a reference sustain its meaning if for any reason the web resource providing the article's metadata were to be moved or otherwise not accessable anymore. Thanks to reification of the article's metadata as its own item, however, its property record can be expanded or modified later on, and statements can be made about it either as subject or object, thereby relating it to other resources available via Linked Data. An Apparatus article that has

[2] https://www.wikidata.org/wiki/Property:P854

been given its own Wikidata item can be used as a bibliographic reference for statements by using the *stated in* (P248[3]) property.

Metadata can be found in each Apparatus article's HTML source code in the form of <meta> elements applying terms from the Dublin Core vocabulary using their @name attributes.

```
<meta name="DC.Rights" content="http://creativecommons.org/licenses/by/4.0"/>
<meta name="DC.Source" content="Apparatus. Film, Media and Digital Cultures of
↪ Central and Eastern Europe"/>
<meta name="DC.Source.ISSN" content="2365-7758"/>
<meta name="DC.Source.Issue" content="5"/>
```

Listing 1: Bibliographic metadata in an Apparatus article's web resource's HTML source (excerpt)

3.1 Metadata mapping

Mappings between Dublin Core metadata terms and Wikidata properties were defined following proposals for bibliographic metadata schema specifications like the Wikidata wiki article *Bibliographic metadata for scholarly articles in Wikidata*[4]. For a more detailed discussion of bibliographic information in Wikidata cf. [14].

Mappings for the Dublin Core vocabulary terms used in Apparatus article sources and exploited for their reification in Wikidata are listed in Table 1 along with their assigned Wikidata properties.

Some Dublin Core terms appear[5] to be used in combination with a finite set of possible values, either explicitly indicated by a @scheme attribute given to the <meta> element (e.g. DC.Date.issued), or simply because none other than a few values ever occur (e.g. DC.Rights). In these cases, the elements of those finite value sets were mapped to Wikidata items where applicable. Table 2 lists ISO639-1 language codes occuring throughout the Apparatus corpus metadata and the Wikidata items representing the natural language they identify. This method of defining a micro-glossary with named entities for a small set of values with regards to a particular predicate has been used for the Dublin Core properties Language, Rights and articleType.

The mappings between Dublin Core vocabulary terms and journal-typical values from the input to Wikidata item and property pages for the output were defined as a JSON object. It contains configuration for all Dublin Core terms which the input resource's (i.e. article's) Wikidata item page representation is

[3] https://www.wikidata.org/wiki/Property:P248

[4] https://www.wikidata.org/wiki/Wikidata:WikiProject_Source_MetaData/Bibliographic_metadata_for_scholarly_articles_in_Wikidata

[5] Section 3.3 will discuss the challenges met during the supposedly straight-forward process of translating the specific metadata at hand from its original vocabulary to Wikidata descriptors.

Dublin Core Term	Schema	Wikidata Property
Contributor.Sponsor		sponsor (P859)
Coverage.spatial		main subject (P921)
Coverage.temporal		main subject (P921)
Creator.PersonalName		author name string (P2093)
		author (P50)
Date.issued	Iso8601	publication date (P577)
Identifier		article ID (P2322)
Language	Iso639-1	language of work or name (P407)
Rights		license (P275)
Source.DOI		DOI (P356)
Source.Issue		issue (P433)
Source.Volume		volume (P478)
Subject		main subject (P921)
Title		title (P1476)
Title.Alternative		item label (rdfs:label)
Type.articleType		genre (P136)

Table 1: Dublin Core terms and their assigned Wikidata counterparts used in automatic import of Apparatus bibliographic resources into Wikidata: left column lists the Dublin Core vocabulary terms as provided by the Apparatus article source code in alphabetical order, stripped of their DC. prefix. Right column shows labels and identifiers (in brackets) of the Wikidata properties that have been used as equivalents during the imports.

to be populated with. Configuration objects, each of which being filed under its respective Dublin Core term identifier, can contain the following entries:

- property: Identifies the Wikidata property to be used as an equivalent for the Dublin Core input term.
- map: A small glossary of values or identifiers known to occur throughout the journal corpus, alongside identifiers of the corresponding Wikidata item pages.
- delimiter: Some attributes in the Apparatus metadata tend to contain multiple individual values within the same metadata object's content field. To handle these cases, one ore more delimiting characters can be specified as a regular expression, and will be used to split up input strings into multiple occurences of the respective property. Properties known to show this characteristic are Sponsor, Coverage.spatial, and Coverage.temporal.

3.2 Import Automation

In order to be able to make statements about the Apparatus articles as bibliographic resources in a way that enables easy conversion into Wikidata statements for their optional upload, it is desirable to have Wikidata item pages

lang	Wikidata item
cs	Czech (Q9056)
de	German (Q188)
en	English (Q1860)
lv	Latvian (Q9078)
pl	Polish (Q809)
ru	Russian (Q7737)
uk	Ukrainian (Q8798)

Table 2: ISO639-1 language codes and their assigned Wikidata item counterparts as used in imports of Apparatus metadata

```
URL=http://www.apparatusjournal.net/index.php
for url in $(curl -s $URL/apparatus/issue/archive | grep -o
 ↪  $URL'/apparatus/issue/view/[0-9]\+'); do
  curl -s "$url" | grep -o
   ↪  $URL'/apparatus/article/view/[0-9]\+/[0-9]\+';
done | sort -u
```

Listing 2: Crawler code for retrieval of individual Apparatus article resource locators

representing the individual publications published in any of the currently available Apparatus issues.

Fortunately, the html versions as which all Apparatus contributions have been published so far contain detailed metadata in `<meta>` tags within the html header. The following pages will detail on the steps that were taken in order to retrieve, clean, arrange and pre-process the Apparatus metadata, so that they could eventually be used for population of Wikidata item pages created as representations of the respective journal contributions.

Harvesting of Apparatus metadata and creation of Wikidata item pages representing Apparatus articles was executed by a collection of makeshift scripts performing the several tasks involved. First, a simple crawler is used for retrieval of all individual articles published in Apparatus. It is implemented as a BASH script and utilizes the command-line tools `curl` and `grep` to extract unique resource identifiers, starting at the Apparatus issue archive page[6]. Listing 2 shows the commands used to crawl the journal website and extract individual article URLs.

For any previously unknown resource, bibliographic metadata are extracted from the HTML source code with regular expressions and written to a temporary XML file. This file is being processed by a Python script converting the XML serialization of the metadata to a JSON file using the `ElementTree` API from the XML Python standard library (3). The purpose of this step is the serialization of harvested metadata into a file format more suitable for processing

[6] http://www.apparatusjournal.net/index.php/apparatus/issue/archive

```
metadata = {}
for article in tree.findall("resource"):
    url = article.attrib.get("url")
    id_meta_element =
     ↪  article.find('./meta[@name="DC.Identifier"]')
    ID = id_meta_element.attrib["content"]
    metadata[ID] = {"id": ID,
                    "url": url,
                    "meta": [child.attrib for child in
                     ↪  article.getchildren()]}
```

Listing 3: Python snippet converting an XML structure (`tree`) to a JSON object

with script languages like Bash and Python than XML is, and the execution of some further intermediate steps for pre-processing of the input and enhancement of the resulting data structure in terms of optimization for the following tasks. For instance, an additional HTTP request needs to be issued for each article in order to retrieve its abstract page, where its DOI can be extracted from.

During the final phase of this prototype pipeline, a Python script reads the pre-processed metadata from a Json file and applies the mapping configurations discussed in subsection 3.1.

Standard import procedure works as follows.The Python import script deserializes an article's list of metadata from a Json file and prepares to loop though it one item at a time. If needed, an empty Wikidata item page is being created with immediate effect on the remote side.

For every item, the Wikidata property equivalent to the metadata term at hand is being looked up in the mapping configurations. A request to the Wikidata API determines the type of the object value which that Wikidata property can be used with within a statement (*Literal*, *Date*, *URI*, etc.). According to this type contraints, a conforming Wikidata object instance is being prepared locally using the API methods provided by the `pywikibot`[7] library. If the property's retrieved specifications indicate that objects used alongside it are expected to be Wikibase item instances, an item search is being issued against the Wikidata Wikibase API endpoint[8]. Search results are being filtered for exact label matches in order to avoid false positives. The remaining search results are being omitted if their number exceeds one, as this implies ambiguities that require human involvement to resolve. However, if this strict Wikidata item search returns exactly one result, it is then used to fill the object slot of the Wikidata statement which is currently being prepared.

Eventually, once a complete statement has successfully been put together

[7] https://www.mediawiki.org/wiki/Manual:Pywikibot
[8] https://www.wikidata.org/w/api.php

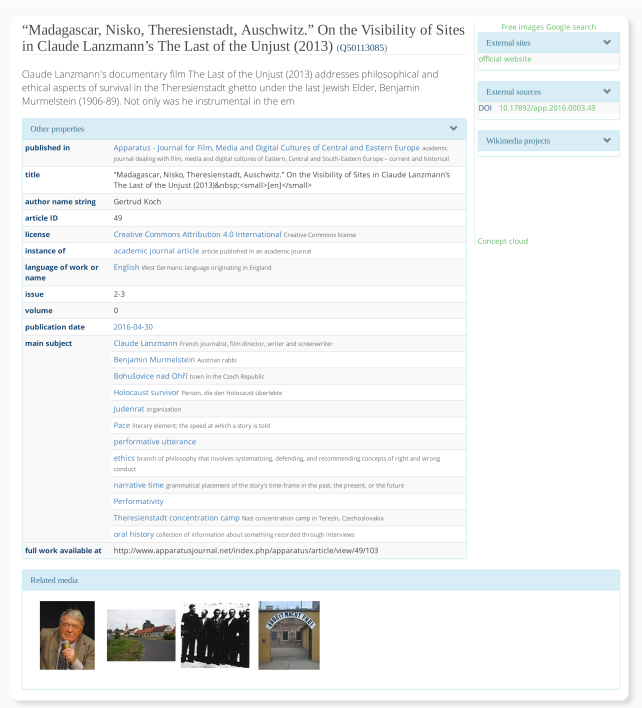

Figure 3: Bibliographic data of an Apparatus online article imported into a Wikidata item page, as displayed by the Wikidata Reasonator tool[9].

Dublin Core Term	Schema	Description
`Date.created`	ISO8601	creation date[10]
`Date.dateSubmitted`	ISO8601	date of submission[10]
`Date.modified`	ISO8601	last modified[10]
`Description`		Abstract (multilingual)
`Identifier.URI`		Article abstract page URL
`Source.ISSN`		Journal ISSN
`Source`		Journal name (multilingual)
`Type`		`Text.Serial.Journal`

Table 3: Dublin Core terms that are being used for metadata attributions by the Apparatus publishers, and for which no adequate Wikidata property could or needed be provided for the automated import method developed for this work.

expressing the bibliographic fact encoded in the current item of the input, it will be equipped with a Wikidata reference entry accounting for its legitimacy by specifying the article's URL on the Apparatus website. The result is being uploaded. Such input data that could not be successfully converted into Wikidata statements get logged, so that human experts can use them as a starting point for quality control.

3.3 Challenges and Shortcomings

Even though the metadata that get published during the journal's release workflow appears to be carefully created and validated, there are some imperfections that require specialized workarounds or render particular metadata fields useless to some degree.

The following section discusses some of the issues encountered during development of the import pipeline.

Not all field values used in the journal's article metadata are valid with regards to the specified standard: Throughout the journal corpus, the language Ukrainian is being identified by the ISO639-2/3 language code `ukr`, even though the `@schema` attributes of the enclosing `<meta>` elements indicate compliance with the ISO639-1 standard, where `uk` would be the correct representation.

There is also no actual consensus on the what exact set of properties is considered appropriate for item pages describing academic journal articles. As stated above (subsection 3.1 on page 5), the metadata import workflow was configured according to a community proposal for the canonical structuring of bibliographic metadata for scholarly articles, documented at pages in the *WikiProject Source MetaData* category of the Wikidata Mediawiki.

However, this proposal is technically a work in progress, for just as brand new Wikidata properties can be suggested, discussed and approved at any time, said selection of bibliographic properties might change over time as well.

[10] Chronological order of values for fields with `DC.Date.` prefixes is somewhat inconsistent throughout the Apparatus metadata.

This might be one reason for its occasional lack of specificity in defining a schema [11]. In the original metadata importer's implementation, Dublin Core term `Identifier.URI` used to be mapped to Wikidata property *reference URL* (`P854`), which is listed as a scholarly article metadata term suggestion in the *Source MetaData* proposal. However, shortly after the ingestion of a an Apparatus article into Wikidata, a bot would appear and replace the `P854` statement with one using property *full work available at* (`P953`[12]) for attribution of the URL linking to the article's abstract.

In response to a complaint about one particular edit made by that bot following this pattern, its developer recently suggested to use the property *described at URL* (`P973`[13]) instead, however both the discussion page and vote on the original proposal of that property attest some confusion as to its exact semantic meaning and proper use.

Up to the time of writing, no alternative equivalent Wikidata property could be determined for the `Identifier.URI` Dublin Core descriptor used for specification of an article's abstract page in the Apparatus resource metadata, under the premise that the original design choice in favor of adherence to the *Source MetaData* group's proposal is not to be compromised. For instance, the temporarily considered candidate property *official website* (`P856`[14]) might serve as an acceptable substitute for the original attribution, but it is not a part of the vocabulary proposed by the *WikiProject* concerned with structuring bibliographic metadata.

Other bibliographic attributes lost some of their explicit meaning, as more than one vocabulary descriptors from Dublin Core had to be mapped to a single Wikidata property. An example of such a compromise is the use of property *main subject* (`P921`[15]) for attributions via term identifier `Subject` as well as `Coverage.spatial` and `Coverage.temporal`.

From three ISO8601-encoded points in time at which different steps in the editing and publishing process were recorded, the presented workflow did only capture the one identified by Dublin Core term `Date.issued`, since the Wikidata property *publication date* (`P577`[16]) was the only applicable property for temporal information on bibliographic resources proposed by the *Source MetaData* project, and the metadata terms specification by the Dublin Core Metadata Initiative unambiguously defines *issued* as the term that is to be used when providing a "date of formal issuance (e.g. publication)" of a resource[17]. However, those Dublin Core terms concerned with temporal information occuring in the journal's metadata records (i.e. the ones in Table 1 and Table 3 whose

[11] For more on Wikidata's community conduct for property creation and obstacles during vocabulary mapping, cf. [13].

[12] https://www.wikidata.org/wiki/Property:P953

[13] https://www.wikidata.org/wiki/Property:P973

[14] https://www.wikidata.org/wiki/Property:P856

[15] https://www.wikidata.org/wiki/Property:P921

[16] https://www.wikidata.org/wiki/Property:P577

[17] http://dublincore.org/documents/dcmi-terms/

identifiers begin with prefix `DC.Date.`) turn out to better be handled carefully, as they draw attention with some unexpected manifestations.

Not only doesn't every Apparatus article contain the same number of temporal metadata fields. More importantly, there appears to be a lack of consistence as to what events or which stages of the journal's editorial and publishing process get recorded for a respective article.

Even though the Dublin Core specification does not detail on for instance whether its term *created* is supposed to indicate the moment of the original resource's creation or the point in time since when its copy has been present on the publisher's hosting infrastrusture, one can argue that either way the resource could not have come into existence *after* its formal issuance by the publisher.

Interestingly though, of the 57 individual contributions currently retrievable with the tooling introduced above, only 35 (61.4%) contain temporal metadata consistent with this assumption. Where the recorded publication date of a resource preceeds its creation, both dates differ by up to 592 days[18].All affected resources have the same journal issue number.

This section illustrated the fundamental flaw that lies in an automated approach towards the retrieval and processing of metadata from one loosely defined schema and its projection and ingestion into another one. Both schemas are subject to possible changes, defined rather vaguely with room for interpretation, and not being communicated with perfect accuracy. Therefore, tooling developed for dealing with the journal corpus in its current state and extent might well require being patched repeatedly in order to keep up with potential schema shifts or special cases in upcoming content.

4 Connecting `neonion` to Wikidata

The semantic annotating app and scholarly hyperreading workspace environment `neonion` offers functionality for collaborative methodical processing of Html and Pdf documents for academic research. It features shared commentary, precise highlighting and semantic tagging of text ranges, including interactive entity linking and classifying based on a structurally highly formalized, shared semantic vocabulary allowing for linking of ontological resources and annotation of semantic relations.

The graphical interface that users are presented with by the `neonion` frontend components utilizes the visual and interactive capabilities of contemporary internet browser engines and employs a balanced mixed-initiative paradigm in order to accommodate the methodical needs especially of non-technical scholars with regards to digitally powered working with textual resources.

[18]The average timespan between publication and creation date is 135.45 days, with the median being 2 days.

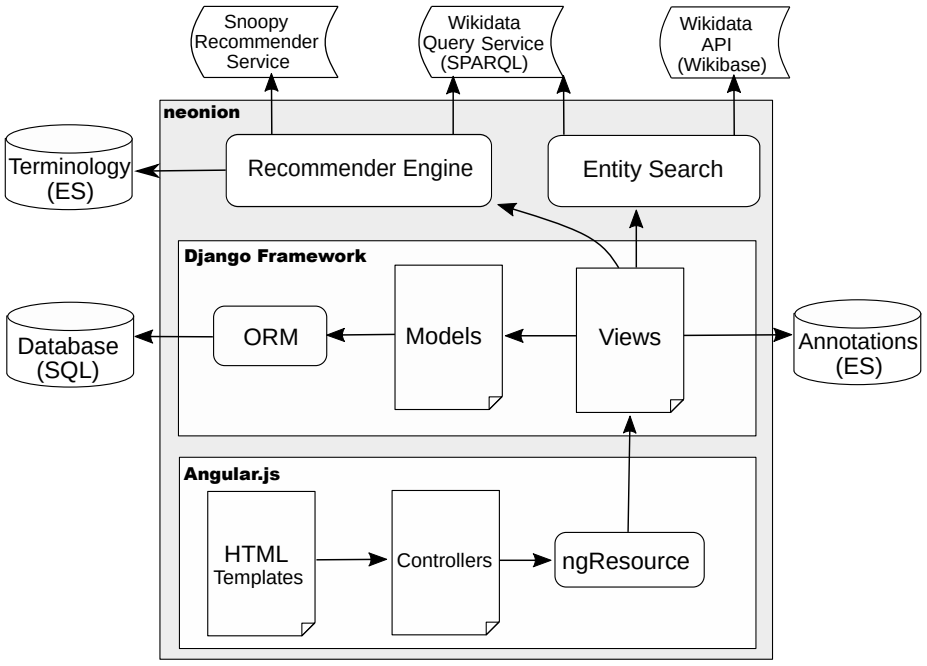

Figure 4: The architecture of the `neonion` web application

4.1 The `neonion` software architecture

The following segment will give a brief overview of `neonion`'s software architecture, its main components, and some of its implementation characteristics at the time when research for this work first began.

Although an in-depth documentation of the application's code base in its entirety would not well serve the purpose of this report, it is in order to provide some context for the design choices and implementation details that will be discussed in the upcoming sections of its remainder. This broad overview of the software's state prior to any changes initiated by the proposed work represents the practical background for those subsequently described extensions and modifications.

As stated above, `neonion` is a web application providing an environment for personalized collaborative enrichments of textual documents. Its graphical user interface is being provided by a frontend implementation for the AngularJS model-view-controller (MWC) javascript framework. Its backend consists of a Python implementation run by the web framework Django, and a single-node instance of the Elasticsearch document search engine.

The Django backend project is responsible for database access and object-relational mapping, storage and retrieval of annotations in and from the search engine instance, business logic, and operating HTTP services providing various

RESTful APIs for communication with the frontend. The `neonion` software architecture does not make much use of the web template system provided by the Django framework. Instead, many objects instantiating data models of the multiple packages (or apps) that are part of the running Django project's configuration are serialized as JSON objects and sent to the frontend application via the HTTP interface provided by Django views implementing basic REST API endpoints.

If the Django template system is being used, it mainly serves Angular templates as static resources, so that all databinding and most of the rendering of dynamic views is being delegated to the web client, i.e. the user's internet browser.

The frontend implementation contains corresponding service functions using Angular's own `ngResource` module for conveniently creating wrappers around RESTful HTTP endpoints. These services connect the numerous Angular controller classes to the backend application and serve them with javascript object representations of database objects. Controllers are javascipt functions that perform business logic and define a public scope for exposure of variables and methods to any views whose directives request access to them as controllers. The Angular framework injects all the declared dependencies into the respective services, controllers and view templates where dependency injections are requested and which it can get hold of, it takes care of bi-directional databinding between the views and their controller scopes, and finally processes and renders HTML templates into dynamic views in the graphical user interface.

When a user opens one of her text documents in `neonion`, she can choose between three modes for annotating its content: simply marking an arbitrary text selection by adding a highlighting to it, commenting on a text range by adding textual notes to it, or adding contextualization and/or classification to a text range using semantic tagging.

The event processing, basic logic and the user interface widgets used for realization of the features involved are based on the JavaScript library AnnotatorJS, or Annotator for short. The Annotator library takes care of extraction, addressing and positioning of annotation ranges (using relative XPATH expressions) with the purpose of storage and visualization, and provides means for interface customization, event handler registration, and extension of base functionality by adding plugins and custom widgets.

The `neonion` frontend application implements an Angular service and controller for setting up an annotator instance, and connecting it to the `neonion` data models and business logic. Apart from detailed configuration and added graphical widgets, the annotator instance gets outfitted with two plugins during its setup. The `Store` plugin is being shipped with the library and is used to let an annotator instance automatically exchange annotations with an endpoint that is to be specified by its URL, and expected to respond to a set of HTTP

methods according to the corresponding storage-related actions.[19] It is configured to communicate with an endpoint at the URL with relative path `/store/`.

The second plugin added to the annotator instance is the `neonion` plugin, made available by injection of a custom prototype into the library's namespace and inheriting from an `Annotator.Plugin()` instance. The `neonion` plugin defines hooks for various events fired by the Annotator library, and handlers implementing the respective behaviour expected by the three different annotation modes offered by `neonion`, including the creation of relations between two semantic taggings using predicates from a controlled vocabulary.

4.2 Accessing Wikidata from within neonion

In order to meet the requirements that have been identified above, several features need to be enhanced or newly developed and added to the `neonion` workbench.

We aim at enabling researchers and subject-matter experts even without any technical background by concealing the complexity of the semantic technologies which `neonion` provides for the customization of highly formalized controlled vocabularies and domain knowledge schema specifications. This objective necessitates functionality that contributes to assisting untrained users in generating structured enrichments of their textual resources in a comprehensive and intuitive manner, including integration into the graphical user interface in compliance with the paradigms followed by neonion's original design[12, 10].

The following section will discuss the specific additions this work made to `neonion` in order to support the proposed workflow and assumed use case one by one, and with focus on the user interface and interaction, beginning with an enhanced and live version of the built-in entity search. Then, the entirely new controls and functionalities created for the support of term suggestions for vocabulary amendment will be illustrated, before the likewise newly created page for the tabular representation of a document's embedded structured factual content and its matching against and optional ingestion into Wikidata will be presented.

Figure 5 shows screenshots of the frontend controls created or modified for providing user interfaces to the features discussed in the following. The top-left quarter shows the entity search dialog during creation of a semantic annotation (subsubsection 4.2.1). The screenshot on the bottom-left depicts a list of vocabulary recommendations (subsubsection 4.2.2). The right half of this figure shows a list of factual statements that have been embedded within a document via semantic annotation, each of them with a button whose appearance indicates whether that statement does already exist in Wikidata (subsubsection 4.2.3).

[19] I.e. `GET, PUT, POST, DELETE` and `search, update, create, destroy`, respectively.

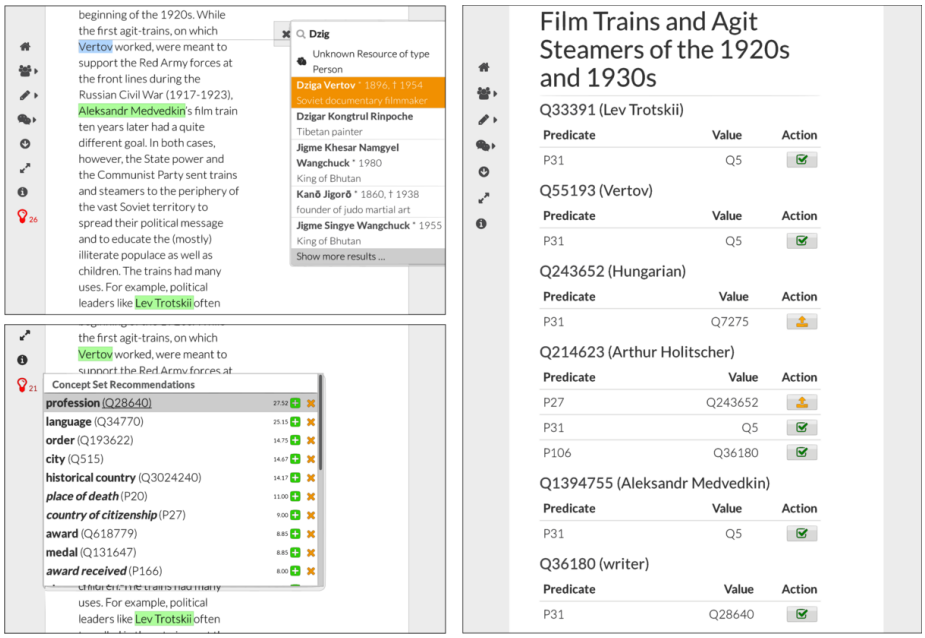

Figure 5: Screenshots of integrated Wikidata functionality in neonion.

4.2.1 Item search for entity linking

The search function has always been a fundamental feature of neonion's semantic tagging mode. However, when implementation of the proposed enhancement began, the then latest release came with a non-recent dump of Wikidata items representing instances of persons, along with instructions for importing these into an elasticsearch index. Although this added to the effort necessary for deployment of a fully-featured running neonion instance, the results were of little use due to the scarcity and age of the shipped data dump.

The feature's existing implementation was extended and overwritten where suitable, in order to replace its default search endpoint location and configuration with a newly created http endpoint providing an interface to the Wikidata client module which has been written for integration of `neonion` and Wikidata.

The newly integrated live entity search implementation involves two steps. First, the Wikidata Wikibase API[20] is being queried, using the `action` parameter to specify the `wbsearchentities` command[21]. The resulting Wikidata item page IDs are inserted into SPARQL queries which are then being sent to the Wikidata Query service (WDQ[22]). These additional requests are made for the sake

[20] https://www.wikidata.org/w/api.php

[21] Incidentally, an update of the API implementation released during the phase of development significantly improved the performance of the entity search

[22] https://query.wikidata.org/bigdata/namespace/wdq/sparql

of filtering the current search results with respect to value constraints implied by the information linked to the local vocabulary's concept used for annotation. More specifically, if the concept used for classification of the new annotation has an equivalent Wikidata item assigned to it, only those entity search results will be presented to the user that are an *instance of* (`P31`) either that very Wikidata item, or of any Wikidata item having a transitive *subclass of* (`P279`) relation with that item. The general SPARQL query used for this filtering step is shown in listing 4.

Additional SPARQL queries might be issued in order to retrieve concept-specific properties, such as a person's dates of birth and death.

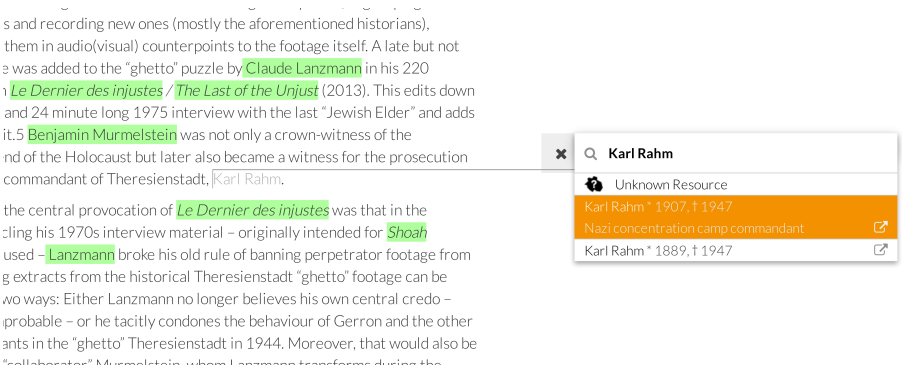

Figure 6: Screenshot depicting the entity linking feature of neonion's semantic tagging mode.

The user interface rendition of a resulting list of valid candidates for entity linking is shown in Figure 6. Initiated by a text range selection within the document content, the semantic tagging sequence starts with the user picking a concept for classification, followed by this search dialog being opened up for manual identification of the currently annotated named entity. External equivalents linked to the applied concept are used to restrict search results to valid candidates. In the case of instances of the concept *Person/Human* (`Q5`), dates of birth and death are displayed for orientation, if available. By clicking on the icon next to the search result label, the user can open the corresponding Wikidata item page in their browser in order to get more context.

4.2.2 Suggesting terminology for vocabulary amendment

Generation of recommendation candidates takes place in the background and mostly runs Wikidata client and utility code written in Python and located in the newly introduced `wikidata` Django app further detailed upon in subsection 5.2.

The recommender system consists of two main parts on the backend side: the terminology extractor and the actual recommendation module. The terminology extractor is being called every time an annotation is being created in the

```
SELECT distinct ?itemLabel ?item WHERE
{
  VALUES ?item { i_1, i_2, ..., i_n } . # results from wbsearchentities
   ↪ query
  ?item (wdt:P31/wdt:P279*) type . # type associated to neonion
   ↪ concept applied to annotation
  SERVICE wikibase:label {
    bd:serviceParam wikibase:language "en" .
  }
}
LIMIT 100
```

Listing 4: SPARQL query filtering Wikidata entity search results according to type information associated to the `neonion` concept used during classification of an annotated named entity

frontend and an external entity has been linked via the search function. When this happens, a registered AnnotatorJS hook calls a local endpoint specifying the linked entity and the concept used for classification, causing the terminology extractor to execute.

The system thereby receives all Wikidata statements involving the linked entity, plus all type information about the corresponding objects. Figure 8 illustrates the data retrieved and the information extracted during this step.

The resulting pieces of terminological knowledge are stored into the ElasticSearch node running alongside the `neonion` deployment, right next to the annotation store. The more terminological information is being collected, the more input data can be fed into the active recommenders.

The recommender engine receives a regular call from the corresponding Angular frontend controller, causing it to run all recommenders that have registered with it. Individual recommenders are simple Python functions returning a dictionary with term identifiers as keys and their corresponding scores as values. In order to register a function as a recommender, it simply need to be equiped with the Python decorator `@recommender`, which takes a argument specifying whether it generates concept or property candidates.

After execution of registered recommender implementations, the recommender module creates `PropertyRecommendation` and `ConceptRecommendation` objects from appropriate candidates using the Django ORM API. These objects can be retrieved by the Angular controller concerned with suggesting recommendations to the user and will be converted into regular `Concept` or `Property` instances if a user decides to add them to their vocabulary.

Presentation of vocabulary recommendations in the web frontend takes the form of a tabular list within a dropdown menu hidden in the document view's navigation bar. The availability of recommendations is being visually indicated, so that users can notice incoming recommendations. Frontend rendition of a current recommendations list is depicted in the bottom-left quarter of Figure 5.

```python
from wikidata import terminology

@recommender("concept")
def recommend_common_supertypes(conceptset):
  """ recommender description in docstring. """
  # retrieve terminological axioms grouped by object types
  types = terminology.faceted_statements(conceptset, 'tpo')
  # ...
  return {term_id:scoring_function(term_id) for term_id in
   ↪  candidates}
```

Listing 5: Registration of single recommender implementation using the `@recommender` decorator from `recommender` module.

For introduction of the new notion of recommendations, the `neonion` project's data model has been extended by multiple new classes inheriting from the Django modeling library. The wikidata Django app developed during this work defines classes for the new models `ConceptRecommendation` and `PropertyRecommendation`, both inheriting from the shared base class `Recommendation`.

Whenever an instance of the `Recommendation` class needs to be created based on the results of the recommender implementations discussed in subsection 5.2, a wrapper object for the linking of an equivalent external resource is created in form of `LinkedConcept` or `LinkedProperty`. It is assigned to the newly instantiated recommendation object and holds the information necessary to create a proper concept or property instance for use in a `neonion` concept set if a user chooses to accept a recommendation.

4.2.3 Share subject-matter expertise by publishing verifiable facts

An additional page was introduced so that all statements embedded in a document in form of semantic annotation and relation annotation can be viewed. The Wikidata client code asynchronously looks up each statement on Wikidata, and frontend controls get updated accordingly as soon as the results come in.

A button with a checkmark icon next to a statement indicates that the statement does exist in Wikidata, the color of the checkmark icon shows whether the remote statement has already been qualified with a bibliographical reference to the Wikidata representation of the current document. An upward-pointing arrow icon on the button next to a statement means that the statement is yet to be uploaded into Wikidata.

A screenshot of a document's embedded statement list alongside with Wikidata availability status indicators can be found in the right half of Figure 5.

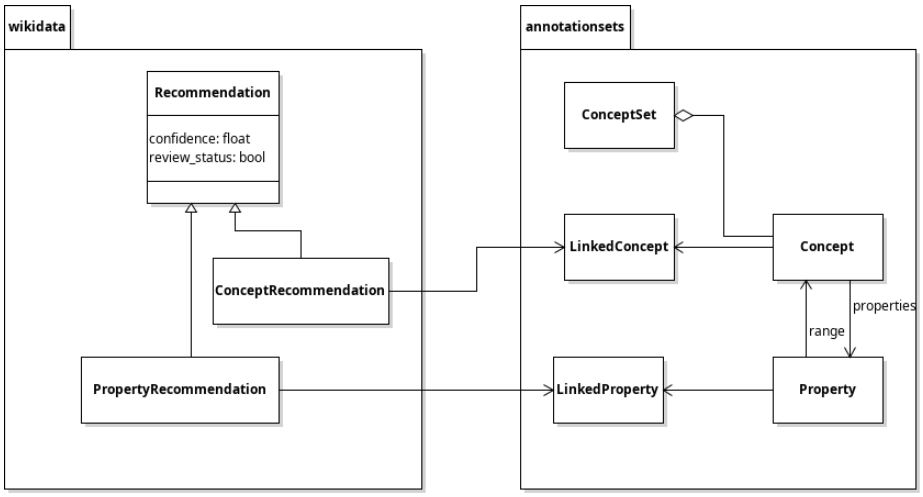

Figure 7: Extensions made to the `neonion` data model

5 Building an extensible engine for integrating content-based term recommenders

Connecting the semantic tooling provided by `neonion` to a remote source for knowledge representations, including constraint-abiding live entity search during semantic tagging of local resources, enables the system to request extending context information on previously generated structured content in concurrency to the user's manual semantic enrichment.

The following sections will introduce yet another part of the newly developed Wikidata client code and its contribution to the `neonion` functionality. Beginning with a collection of functions for extraction and analysis of implicit terminological knowledge from the external ontological knowledge source, this chapter contains discussion of different components involved in the generation of vocabulary recommendations and some specific implementations. Following the presentation of recommender strategies for both concept and property candidate generation in subsection 5.2, the potential for integration of external recommender services will be demonstrated by showing how the association rule mining engine *Snoopy* provides reliable property recommendations as an external web service via an HTTP endpoint. Based on the main objectives targeted by the approach pursued during development of the *Snoopy* system, its underlying assumptions and the implications of the results, the concluding passages will contain a recap on the various attempts towards viable term suggestion, and a brief discussion on their most important aspects.

5.1 The Terminology Module

The terminology module in the new wikidata Django app contains methods for retrieval of terminological knowledge from the remote knowledge base, its storage in and retrieval from a local knowledge representation store, and various operations on it. This section contains a brief description of the tasks it performs and the tools it offers.

In order to efficiently run some basic candidate generators implemented for concept and property recommendation over and over as user input is coming in, simplified representations of related contents from the target knowledge base are being stored in the local system. The idea behind this is that while the actual structured data fed into the system by the annotating user changes frequently, the terminological information as it is being harvested from the knowledge base does not. Based on this assumption and the desire for possible restructuring for optimizations, retrieval of related object types from a local store is being preferred over repeatedly querying these type data from the external SPARQL endpoint, same as argued by e.g. [20] and [15].

Retrieval and extraction of terminological knowledge about related objects is being triggered by each succesful entity linking during creation of an annotation in semantic tagging mode. The call issuing this operation is being invoked by a hook registered with the central annotator instance as provided by the AnnotatorJS library, which sends a request to a designated HTTP endpoint, specifying the linked entity and the concept set as well as the actual concept used in the annotation.

This endpoint gets handled by a Django view which calls the respective method in the terminology module. There, a `POST` request gets sent to the Wikidata query service endpoint with a SPARQL query in its body. A generalization of such a query is illustrated in 6. It requests all those Wikidata items along with their types (via the `P31` property) that are related to any of the specified input entitites in that they co-occur with them in Wikidata statements either as objects or subjects, along with the Wikidata properties that make up the predicates of the statements and the actual related items themselves.

```
SELECT * WHERE {
  ?obj wdt:P31 ?objecttype .
  { ?obj ?pr ?entity . }
  UNION { ?entity ?p ?obj . }
  MINUS { ?entity wdt:P279 ?obj . }
  VALUES ?entity { i_1, i_2, ..., i_n } .
}
```

Listing 6: SPARQL query used for retrieval of types which any instances of relate to a specific set of items

The resulting data, combined with the related type information available for

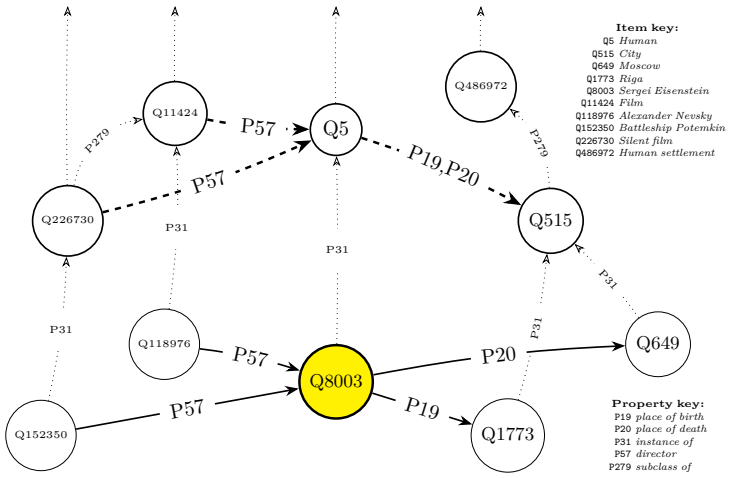

Figure 8: Example for implicit terminological knowledge and its extraction from Wikidata. Solid lines represent factual statements about the seed entity highlighted in yellow, dotted lines indicate ontological relations like *instance of* and *subclass of*, and dashed lines represent some of the terminological axioms that can be inferred from these.

the concept that has been used for the creation of an annotation, allows for the extraction of terminological axioms of the form $(c_s, p, c_o) \in (C \times P \times C)$, where c_s and c_o are types extracted from the subject and object of an assertional statement, respectively, both elements of the set of all known types C, and $p \in P$ is an element of the set of all known properties[23].

Figure 8 illustrates the kind of terminological knowledge that becomes available though application of the SPARQL query shown in 6, based on a simplified real-world example. The vertices in the depicted graph represent Wikidata item pages closely related to a given seed item `Q8003` (highlighted vertex at bottom-center). Vertices connected by solidly drawn edges are explicitly linked together, i.e. they co-occur in a Wikidata statement with the property indicated in the edge label. Dotted arrows represent taxonomical relationssships either of type *instance of* (`P31`) or *subclass of* (`P279`). Dashed lines show implicit terminological projections of the explicitly stated relationships onto an abstraction layer consisisting of the concepts being instantiated by the items directly linked to the seed entity. These inferred axioms about item abstractions are the basis for vocabulary suggestion.

With regards to the real-world data the example is based on, this means that the assertional statements about the seed entity `Q8003`, representing *Sergei Eisenstein*, constitute relationships between the well-known soviet film director and other entities represented as Wikidata items. Those item pages contain

[23]Either c_s or c_o is the Wikidata item specified as the equivalent of the concept from the vocabulary that has been used for classification of the original annotation.

information on class membership, expressed by *instance of* property (P31) occurences linking an item page to one or more class item pages. The seed entity *Sergei Eisenstein* is an *instance of* Wikidata class item *Human* (Q5), and the vertices Q1773 (*Riga*) and Q649 (*Moscow*) on the bottom right both represent Wikidata items that are instances of the class *City* (Q515). Based on the assertional statements found about the seed entity and the *instance of* relations they extend to, generalized claims can be made about concepts. In this example, the observation that a human being has a *place of birth* (P19) and a *place of death* (P20), and that both places are known to be cities suggests that in order to make sense of a person's bibliographic data, it might be necessary to be aware and to comprehend the concept *City*. Therefore, the relationship established between *Human* and *City* via the properties *place of birth* and *place of death* constitutes terminological axioms representing the fundamental discovery that cities are places where some people are born and some people die.

Consequently, the concept *City* as well as the the properties *place of birth* and *place of death* make for term candidates for vocabulary recommendation.

```
SELECT ?type ?supertype WHERE {
  VALUES ?type { t_1, t_2, ...t_n } .
  ?type wdt:P279 ?supertype .
}
```

Listing 7: SPARQL query for retrieval of all pairs of items connected by a *subclass of* (P279) relation that can be found for a set of items representing types.

Subsequently, a second SPARQL query, shown in 7, is sent to the Wikidata query service endpoint. Its purpose is to obtain taxonomical information on the hierarchical relationships between previously and most recently discovered types, and those that might be found in Wikidata as generalizations of the represented concepts, or in other words *broader* conceptualizations of currently known types.

This request made to the Wikidata query service results in a list of tuples, each of them containing one item from the original input set that has been sent with the query and one item that is defined as a broader conceptualization of it.

The resulting type taxonomy is a directed, acyclic[24] graph $G = \langle V, E \rangle$ with V and E being the set of vertices (types) and edges (subtype relationships), respectively.

The resulting terminological and taxonomical knowledge is stored into the search engine provided by the locally running ElasticSearch node otherwise used as the annotation store. This way, axiomatic statements about type relations,

[24]The acyclicness of the underlying type taxonomy is getting enforced by Wikidata bots.

property statistics and type hierarchy can be retrieved efficiently for the computation of candidates for concept and property recommendation.

5.2 The Recommender Module

The recommender module, located within the newly introduced wikidata app for the `neonion` Django project, evaluates the terminological information gathered from Wikidata during semantic annotation with entity linking, which was described in the previous section. The module contains a simple registry mechanism for concept or property candidates generators. Subscribing client code implementing candidate generators is internally being organized and executed when required. The thereby obtained concept and property candidates for recommendation are submitted to some logical restrictions, before appropriate results enter the phase of actual recommendations available to the user.

Proper vocabulary recommendations are reified as instances of the application's data model handled by the Django framework's object-relational mapping (ORM), so that they can be persisted, accessed, but also migrated and exchanged in the same way as documents, user groups, and concept sets managed by a deployed `neonion` instance.

Any candidates generated by the recommender module get checked for novelty against the concepts and properties that already exist within `neonion` by using Django's database-abstraction API. Those candidates specifying a Wikidata identifier that is not already linked to any of the existing `LinkedConcept` or `LinkedProperty` objects stored in the database backend are used as input for candidate reification. This procedure creates a `LinkedConcept` or `LinkedProperty` object for linking the external resource associated with the recommended term, and an instance of the corresponding `Recommendation` subclass shown by Figure 7 in subsubsection 4.2.2.

Recommendations instantiated from one of the two `Recommendation` subclasses are available to the frontend application via a route through the usual stack of Django views and URL mapping to HTTP interfaces, where REST service wrappers created by Angular's `ngResource` utility pick up the payload serialized to JSON and make it available to controller classes that bind the data to user interface components via directives for the Angular template rendering system.

The following passages give some specifics on how the recommender module generates candidate lists from the previously aggregated terminological knowledge.

5.2.1 Concept Recommendations

The basic concept recommender solution employed by the prototype begins with ranking all item types found in the terminological knowledge store by the

frequency of their occurence. It then runs a variant of the eigenvector centrality algorithm[25] on the type hierarchy taxonomy accessible via the terminology module.

The formula for recomputation of each vertex's score is shown in Equation 1. The algorithms runs iteratively as long as the observed momentum exceeds a certain threshold value.

$$v_i = \sum_{j \in N_i} x_{i,j} \cdot v_j \tag{1a}$$

$$x_{i,j} = \frac{1}{2 \cdot |N_j|} \tag{1b}$$

The ranked results of the eigenvector application undergo an additional step before being returned as a candidate list. Beginning with the highest scored type, the generator iterates over the ranked list and for every items removes all its subtypes from the list remainder. This prevents pollution of the recommendation list that would occur caused by not only presenting false positives, or items of little interest for the user, but also a potential number of their likewisely useless supertypes.

5.2.2 Property Recommendations

Property recommendations are generated by querying the *Snoopy* rule mining engine, running as an external webservice. As *Snoopy* requires collections of seed properties as input, an additional recommender strategy needs to cover cases where the user starts off with a vocabulary where no properties have been assigned to any concepts yet. This "fallback" property recommender works similar to the concept recommender introduced in the previous section, in that it processes the terminological information gathered by the module described in subsection 5.1.

Instead of simply rank property suggestions by the number of their occurence or the number of unique object item classes they co-occur with, variants of the *tf-idf* weighting scheme were chosen as a ranking function, with the property in question being considered the "document" [11].

Based on experiments with possible variants, the number of distinct types co-occuring with a property was chosen as the metric on which term and (inverse) document frequency was determined. The alternatives are listed in Figure 9, with tf-idf$_t$ being the method of choice.

While this approach towards recommending properties supposedly suitable for describing the entities that have been previously annotated exclusively relies on information available for those particular entities, the aforementioned *Snoopy* approach takes into account what globally emerging patterns the Wikidata properties occur in.

[25] Cf. introduction on PageRank in [11, pp. 424].

$p \in P$	idf_o	idf_t	idf_s	tf-idf_o	tf-idf_t	tf-idf_s
`P21` (sex or gender)	3.59	4.24	0.89	1.20	0.94	0.89
`P735` (given name)	3.59	2.52	0.78	0.98	2.75	0.78
`P2632` (place of detention)	3.31	4.24	1.28	2.65	1.70	1.28
`P106` (occupation)	2.78	2.15	0.73	1.62	3.23	0.73
`P27` (country of citizenship)	2.45	3.35	0.78	2.23	1.52	0.78
`P166` (award received)	2.36	2.38	1.28	5.20	6.66	1.28
`P19` (place of birth)	1.98	2.79	0.78	3.06	2.28	0.78
`P57` (director)	4.67	4.93	2.64	4.67	4.93	2.64

Figure 9: The tf-idf variant used for property weighting applied to different metrics available for properties

5.3 Using the *Snoopy* system

The *Snoopy* system[7] is an association rule mining engine designed to provide semi-automatic support for collaborative workflows involving manual population of semi-structured and schema-less knowledge bases[9].

Among its main objectives, it aims at the stabilizing enforcement of implicit schema information emerging from collaboratively aggregated data[8].

The upcoming sections touch upon the challenges that come with evolving knowledge bases, changing schema requirements, employment or absence of authoritatively maintained schema specifications, and other issues that arise from knowledge aggregation, especially in collaborative initiative.

Based on those considerations, the underlying principles and assumptions of the *Snoopy* system will be described and its application for alignment of collaboratively built semi-structured content with emergent schemas will be discussed both in general and regarding its actual employment during realization of the use case and workflow presented in this paper.

The *Snoopy* client code to be registered to the recommender module queries a *Snoopy* webservice for recommendations based on the list of properties that have been assigned to the same concept in the `neonion` local vocabulary. The more seed properties the input sent to *Snoopy* contains, the higher the confidence values of the candidate properties returned. By repeating this for every concept in a `neonion` concept set, the properties that are most likely to be useful to the annotator can be determined.

5.3.1 Schema proliferation and heterogeneity

The quintessential problem targeted by the *Snoopy* system lies in the heterogeneity of vocabulary usage inevitably materializing during the population of a semi-stuctured knowledge base by a large community of contributors. Without an explicit schema whose constraint specifications impose restrictions upon the range of valid properties and attribute values that are available to the contribu-

tors, heterogenous use of vocabulary terms is prone to lead to idiosyncrasy and inconsistency.

Absence of a fixed schema carries the risk of schema drift and proliferation, manifesting as synonymy of spawning vocabulary terms and properties, ambiguities due to lack of shared conceptualizations, and other effects leading to increased noise, and a decline in performance and meaningfulness of the aggregated contents.

However, defining static schemas, especially for coverage of numerous subject domains that might be part of a knowledge base intended for universal world knowledge, involves assumptions and predictions about the data expected to be modeled[2]. Any attempt to define and maintain such a fixed schema will likely run into problems while trying to keep up with the demands of contributions and contributors in a collaborative environment, and it has been shown that purposeful conduct with the intention of clear-cut, unambiguous and authorative schema maintenance does not prevent pollution and proliferation[8]. Moreover, there's the issues of over- and underspecification that might lead to complex data modeling challenges arising with the evolution of resources and divergence between them and even carefully designed schema components[1].

5.3.2 Towards alignment with emergent implicit schemas

The *Snoopy* system evaluates the entire knowledge base during computation of association rules. An association rule predicts the co-occurence of certain disjoint item subsets taken from a global set of items, and consists of an antecedent, a consequent, and a confidence value supporting the prediction that an observation of the former implies presence of the latter.

In our scenario, *Snoopy* receives a set of property identifiers and returns a list of property identifiers taken from the consequent parts of association rules that match the specified input list.

The underlying assumption is that the most likely properties to be useful for expressing information about a certain resource are the ones that are used most frequently in combination with the properties already used on that resource. This is a kind of popularity-based metric and thus enforces the adoption of an implicit schema embedded in the collaborative community efforts. Where the semi-structured, to a certain extent schema-free nature of a knowledge base allows for competing, synonymous, redundant properties to be applied, the *Snoopy* approach gives preference to those that have been used most. By suggesting recommendations generated with this approach, a collaborative system encourages its users to choose the most frequently applied property over its competitors that have been used in similar context. Thanks to the repeated evaluation of the knowledge base, every time the most popular choice gets picked, its support and thereby its likelyhood to be suggested to users increases.

When a property becomes more likely to be suggested to collaborators for adding to the information about a half-populated resource, the number of people who know of it also grows. This leads to a more consistent shared conceptualiza-

tion, making it less likely that is misused. Therefore, the implicit schema that is being enforced by the recommender suggesting the most popular properties over time aligns with the shared conceptualization that the user base agrees upon.

6 Future Work

The presented prototypical realization of features for the proposed workflow shows that domain-specific vocabulary customization can easily be integrated in existing software solutions by connecting a large collaborative knowledge base as an external terminological authority. This conclusion discusses further improvements that are thought to add to the usefulness of the presented results.

6.1 Semantics-aware, content-based term recommendation

The presented implementation presents users with a list of concept and property recommendations as suggestions for addition to the vocabulary they use, meaning that manual review of them is pending. These suggestions appear in order of their confidence value until the user either decides to add them to the vocabulary or to dismiss them entirely. However, apart from this definitive explicit user feedback on suggested terms, there is also the implicit feedback a user gives by continuously ignoring a suggested concept or property pending review. This implicit feedback, materializing in either the duration for which a recommendation has been sitting in the suggestion list or the number of times it has been suggested to the user as a result of them actively opening the recommendations list, could be taken into account for filtering and sorting. Such incorporation of implicit feedback by the absence of user interaction has been demonstrated in [18], where recommendations have a time-to-live (TTL) value, which decreases over time and affects the position at which the recommendation gets shown.

Our implementation's recommendation backend generates property recommendation in several ways, the most significant of which makes use of an external web service running the association rule mining engine *Snoopy* [7]. The results of these generators are used to assign properties to concepts where they are applicable in terms of domain and range, which means that only those properties get suggested that in effect can be used to relate two Wikidata items to each other. In contrast to this restriction, Wikidata statements about items can also assign literal values via some properties, like strings, URLs, or points in time. Accordingly, future improvements of our prototype could reflect this capability by evaluating the property target value type information available in the Wikidata property metadata, as it has been demonstrated in the automated import pipeline for reification of Apparatus journal articles into Wikidata item pages in subsection 3.2.

This would of course require the controls provided by the graphical user interface to adjust to the determined value types of the properties available for relation annotation, for instance during the entity linking phase of annotation

creation. Currently, the entity linking dialogue only suggests possible Wikidata item matches for the selected text range meant for annotation. For the sake of this projected enhancement, however, it should also allow for the classification of a text selection as a string literal or a date in the Gregorian calendar, so that the resulting annotation can be found by the connector function used for `neonion`'s relation annotation feature described in [3].

References

[1] Ziawasch Abedjan, Johannes Lorey, and Felix Naumann. "Reconciling ontologies and the web of data". In: *21st ACM International Conference on Information and Knowledge Management, CIKM'12, Maui, HI, USA, October 29 - November 02, 2012*. 2012, pp. 1532–1536. DOI: 10.1145/2396761.2398467. URL: http://doi.acm.org/10.1145/2396761.2398467.

[2] Ziawasch Abedjan and Felix Naumann. "Amending RDF entities with new facts". In: *European Semantic Web Conference*. Springer. 2014, pp. 131–143.

[3] André Breitenfeld. "Link the World – A usability study on annotation-based manual relation extraction". Master Thesis. Berlin, Germany: Freie Universität Berlin, Nov. 2016.

[4] André Breitenfeld et al. "Enabling Structured Data Generation by Non-technical Experts". In: *Mensch und Computer 2017 - Tagungsband*. Ed. by Manuel Burghardt et al. Regensburg: Gesellschaft für Informatik e.V., 2017, pp. 181–192.

[5] Michael Clarke and Pam Harley. "How Smart Is Your Content? Using Semantic Enrichment to Improve Your User Experience and Your Bottom Line". In: *Science Editor*. Vol. 37. Jan. 2014, pp. 40–44.

[6] Fredo Erxleben et al. "Introducing Wikidata to the Linked Data Web". In: *The Semantic Web - ISWC 2014 - 13th International Semantic Web Conference, Riva del Garda, Italy, October 19-23, 2014. Proceedings, Part I*. 2014, pp. 50–65. DOI: 10.1007/978-3-319-11964-9_4. URL: http://dx.doi.org/10.1007/978-3-319-11964-9_4.

[7] Wolfgang Gassler, Eva Zangerle, and Günther Specht. "Guided Curation of Semistructured Data in Collaboratively-built Knowledge Bases". In: *Future Generation Computer Systems* 31 (Feb. 2014), pp. 111–119. ISSN: 0167-739X. DOI: 10.1016/j.future.2013.05.008. URL: http://dx.doi.org/10.1016/j.future.2013.05.008.

[8] Wolfgang Gassler, Eva Zangerle, and Günther Specht. "The Snoopy Concept: Fighting heterogeneity in semistructured and collaborative information systems by using recommendations". In: *2011 International Conference on Collaboration Technologies and Systems, CTS 2011, Philadelphia, Pennsylvania, USA, May 23-27, 2011*. 2011, pp. 61–68. DOI: 10.1109/CTS.2011.5928666. URL: https://doi.org/10.1109/CTS.2011.5928666.

[9] Wolfgang Gassler et al. "SnoopyDB: Narrowing the Gap Between Structured and Unstructured Information Using Recommendations". In: *Proceedings of the 21st ACM Conference on Hypertext and Hypermedia*. HT 2010. Toronto, Ontario, Canada: ACM, 2010, pp. 271–272.

[10] Eric Horvitz. "Principles of mixed-initiative user interfaces". In: *Proceedings of the SIGCHI conference on Human Factors in Computing Systems*. ACM. 1999, pp. 159–166.

[11] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to Information Retrieval*. New York, NY, USA: Cambridge University Press, 2008.

[12] Müller-Birn, Claudia and Klüwer, Tina and Breitenfeld, André and Schlegel, Alexa and Benedix, Lukas. "neonion – combining human and machine intelligence". In: *Proceedings of the 18th ACM Conference Companion on Computer Supported Cooperative Work & Social Computing*. New York, 2015, pp. 223–226.

[13] Joachim Neubert. "Wikidata as a Linking Hub for Knowledge Organization Systems? Integrating an Authority Mapping into Wikidata and Learning Lessons for KOS Mappings". In: *Proceedings of the 17th European Networked Knowledge Organization Systems Workshop co-located with the 21st International Conference on Theory and Practice of Digital Libraries 2017 (TPDL 2017), Thessaloniki, Greece, September 21st, 2017*. 2017, pp. 14–25. URL: http://ceur-ws.org/Vol-1937/paper2.pdf.

[14] Finn Årup Nielsen, Daniel Mietchen, and Egon L. Willighagen. "Scholia, Scientometrics and Wikidata". In: *The Semantic Web: ESWC 2017 Satellite Events - ESWC 2017 Satellite Events, Portorož, Slovenia, May 28 - June 1, 2017, Revised Selected Papers*. 2017, pp. 237–259. DOI: 10.1007/978-3-319-70407-4_36. URL: https://doi.org/10.1007/978-3-319-70407-4_36.

[15] Tommaso Di Noia et al. "SPrank: Semantic Path-Based Ranking for Top-*N* Recommendations Using Linked Open Data". In: *ACM TIST* 8.1 (2016), 9:1–9:34. DOI: 10.1145/2899005. URL: http://doi.acm.org/10.1145/2899005.

[16] Eugen Ruppert. "Unsupervised Conceptualization and Semantic Text Indexing for Information Extraction". In: *The Semantic Web. Latest Advances and New Domains - 13th International Conference, ESWC 2016, Heraklion, Crete, Greece, May 29 - June 2, 2016, Proceedings*. 2016,

pp. 853–862. DOI: 10.1007/978-3-319-34129-3_54. URL: https://doi.org/10.1007/978-3-319-34129-3_54.

[17] David Shotton et al. “Adventures in Semantic Publishing: Exemplar Semantic Enhancements of a Research Article”. In: *PLOS Computational Biology* 5.4 (Apr. 2009), pp. 1–17. DOI: 10.1371/journal.pcbi.1000361. URL: https://doi.org/10.1371/journal.pcbi.1000361.

[18] Yordan Terziev et al. “Ontology-based Recommender System for Information Support in Knowledge-intensive Processes”. In: *Proceedings of the 15th International Conference on Knowledge Technologies and Data-driven Business*. i-KNOW ’15. Graz, Austria: ACM, 2015, pp. 1–8.

[19] F. T. Ulaby. “Electronic journals versus print: Publishing in the electronic age [Point of view]”. In: *Proceedings of the IEEE* 94.6 (June 2006), pp. 1043–1044. ISSN: 0018-9219. DOI: 10.1109/JPROC.2006.875788.

[20] Iacopo Vagliano et al. “Content Recommendation through Semantic Annotation of User Reviews and Linked Data - An Extended Technical Report”. In: *CoRR* abs/1709.09973 (2017). arXiv: 1709.09973. URL: http://arxiv.org/abs/1709.09973.

[21] Brian Whitworth and Robert S. Friedman. “Reinventing Academic Publishing Online. Part II: A Socio-technical Vision”. In: *First Monday* 14.9 (2009). URL: http://firstmonday.org/htbin/cgiwrap/bin/ojs/index.php/fm/article/view/2642.