RWTH AACHEN UNIVERSITY
Chair of Computer Science 5
Information Systems & Databases
Prof. Dr. Stefan Decker

Master Thesis

A Batch Recommender for Fast Ontology Prototyping

Enis Zejnilovic
Matr.-No.: 331233
Study Program: Master of Computer Science
July 13, 2021

Supervisors: Prof. Dr. Stefan Decker
Chair of Computer Science 5 (DBIS)
RWTH Aachen University

Dr. Michael Cochez
Knowledge Representation and Reasoning Group
Vrije Universiteit Amsterdam

Advisors: Johannes Lipp, M.Sc.
Data Science and AI
Fraunhofer FIT

Lars Gleim, M.Sc.
Chair of Computer Science 5 (DBIS)
RWTH Aachen University

Eidesstattliche Versicherung
Statutory Declaration in Lieu of an Oath

Zejnilovic, Enis	331233
Name, Vorname/Last Name, First Name	Matrikelnummer (freiwillige Angabe) Matriculation No. (optional)

Ich versichere hiermit an Eides Statt, dass ich die vorliegende ~~Arbeit/Bachelorarbeit~~/ Masterarbeit* mit dem Titel
I hereby declare in lieu of an oath that I have completed the present paper/Bachelor thesis/Master thesis* entitled

A Batch Recommender for Fast Ontology Prototyping

selbstständig und ohne unzulässige fremde Hilfe (insbes. akademisches Ghostwriting) erbracht habe. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel benutzt. Für den Fall, dass die Arbeit zusätzlich auf einem Datenträger eingereicht wird, erkläre ich, dass die schriftliche und die elektronische Form vollständig übereinstimmen. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.
independently and without illegitimate assistance from third parties (such as academic ghostwriters). I have used no other than the specified sources and aids. In case that the thesis is additionally submitted in an electronic format, I declare that the written and electronic versions are fully identical. The thesis has not been submitted to any examination body in this, or similar, form.

Aachen, July 13, 2021	
Ort, Datum/City, Date	Unterschrift/Signature

*Nichtzutreffendes bitte streichen
*Please delete as appropriate

Belehrung:
Official Notification:

§ 156 StGB: Falsche Versicherung an Eides Statt
Wer vor einer zur Abnahme einer Versicherung an Eides Statt zuständigen Behörde eine solche Versicherung falsch abgibt oder unter Berufung auf eine solche Versicherung falsch aussagt, wird mit Freiheitsstrafe bis zu drei Jahren oder mit Geldstrafe bestraft.
Para. 156 StGB (German Criminal Code): False Statutory Declarations
Whoever before a public authority competent to administer statutory declarations falsely makes such a declaration or falsely testifies while referring to such a declaration shall be liable to imprisonment not exceeding three years or a fine.
§ 161 StGB: Fahrlässiger Falscheid; fahrlässige falsche Versicherung an Eides Statt
(1) Wenn eine der in den §§ 154 bis 156 bezeichneten Handlungen aus Fahrlässigkeit begangen worden ist, so tritt Freiheitsstrafe bis zu einem Jahr oder Geldstrafe ein.
(2) Straflosigkeit tritt ein, wenn der Täter die falsche Angabe rechtzeitig berichtigt. Die Vorschriften des § 158 Abs. 2 und 3 gelten entsprechend.
Para. 161 StGB (German Criminal Code): False Statutory Declarations Due to Negligence
(1) If a person commits one of the offences listed in sections 154 through 156 negligently the penalty shall be imprisonment not exceeding one year or a fine.
(2) The offender shall be exempt from liability if he or she corrects their false testimony in time. The provisions of section 158 (2) and (3) shall apply accordingly.

Die vorstehende Belehrung habe ich zur Kenntnis genommen:
I have read and understood the above official notification:

Aachen, July 13, 2021	
Ort, Datum/City, Date	Unterschrift/Signature

Contents

List of Figures

1 Introduction

With the technological progress and the rising amounts of data being created, it gets more and more important to process this data for humans as well as machines. In times of big data, an emerging issue is creating data swamps, data warehouses, or data lakes that are not easy to understand. The data is often not enhanced with metadata and loses its processability and value. Therefore, the Semantic Web [BHLa01] was created which allows humans as well as machines to process the data much easier. One key element for describing and connecting data inside the Semantic Web are ontologies. Ontologies are mostly used to describe a certain domain (e.g., biology, government, sports). Depending on the precision of the ontology specification, the notion of this ontology contains several data and conceptual models, including, sets of terms, classifications, database schemas, or fully axiomatized theories [ShEu13]. The specification of an ontology is done by using terms and relationships between these terms, which are agreed upon inside this domain. Editors like Protege [NSD*01] or Neologism 2.0 [LGC*20, LGC*21] can be used to create ontologies. The use of ontologies has many advantages, some of which are listed below.

- Ontologies support semantic annotation such that other persons, who are not familiar with the ontology domain, can understand its meaning.
- Relationships to terms inside an ontology or other ontologies are helpful to understand the context of the data.
- Ontologies generally facilitate knowledge exchange between humans and machines.
- The (syntactical) validation of data is enhanced with the information provided in ontologies, such as the hierarchy, data types, and relations.
- The facts and rules in ontologies support the generation and derivation of new knowledge.

1.1 Motivation

A general best practice is not to create something that already exists, which also applies to ontologies. Therefore, ontology reuse is very beneficial when creating new ontologies, as plenty of terms and properties already exist [MJO*17]. Some hurdles have to be overcome when creating an ontology. One of them is the ontology creation process that can be simplified using an ontology editor like e.g., Neologism 2.0 [LGC*20, LGC*21]. The basic functionalities of an editor are CRUD (create, read, update, delete) functionalities of nodes and relations. One problem that may occur during ontology creation is spelling errors. Spelling errors provide wrong or misleading information and change the meaning of the created node or relation. Another issue when creating an ontology is that the ontology is only available locally and cannot be processed by other persons or machines.

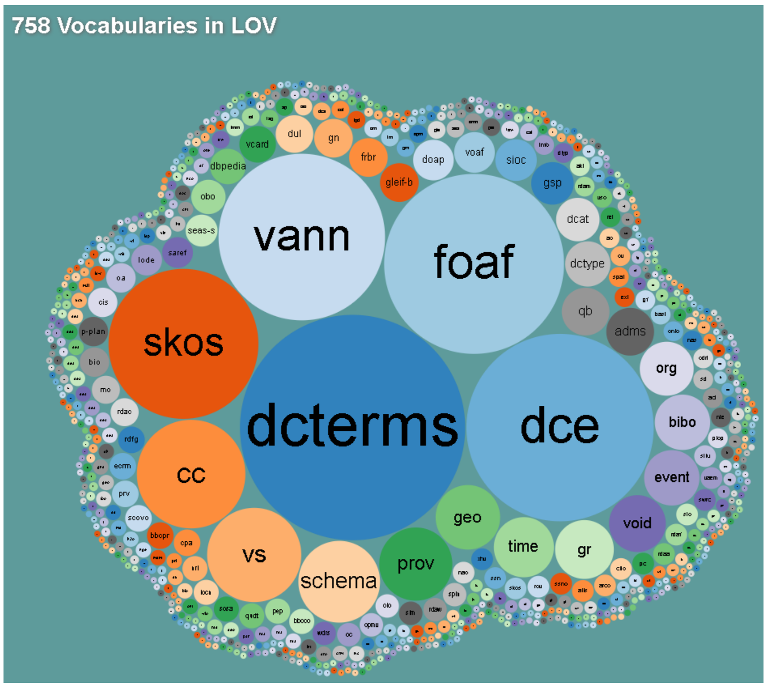


Figure 1.1: An overview of all currently available ontologies inside LOV in a cloud-like view [VAPV11].

Another hurdle is the specification of metadata for each node and relation. This hurdle can be overcome by reusing terms from existing ontologies. An issue here is that the research for existing ontologies that may have the desired meaning of the ontology to be created can be very time-consuming. The research for ontologies can be done using a search engine for ontologies. However, the obtained results may include recommended terms that do not match the meaning of the query. This problem is generally hard to solve, as the interpretation of suggestions is very subjective and there is no clear definition of what *good* suggestions are. A vivid example is the results we obtain from the Linked Open Vocabularies (LOV) API [VAPV17] when we query certain terms. The LOV API is offering a REST service for querying the largest open-source dataset for ontologies online [VAPV11]. An overview of the ontologies in LOV can be seen in Figure 1.1. The size of a bubble describes the popularity or usage rate of an ontology i.e., the larger a bubble the more common an ontology is used by datasets or other ontologies. Querying the input keyword *test* or *rain* the LOV API returns the terms ***RouteStop*** and ***Train***, respectively. Both recommended terms have a different meaning but include the queried terms *test* and *rain* as a substring.

The decision for a search engine can be hard as it may be domain-specific. Therefore, it can be a time-consuming task to find the *correct* search engine to retrieve the best fitting terms for the desired ontology. A combination of different search engines for specific domains could improve the results. There is currently no solution to combining the knowledge of domain experts and ontology experts to create an overall framework for recommendations of ontologies.

Most users would create ontologies from scratch as the mentioned effort for researching existing ontologies is high [CrCu05]. Creating a new ontology can be done using your ontology or as mentioned using existing ones. Nevertheless, it may occur that an existing domain ontology that was found does not fully cover the concept that you want to describe. This can be solved by combining multiple existing ontologies exploiting the specific metadata and relationships that fit your needs. With this best practice, the processing of ontologies gets much easier, as creating a new ontology out of different ontologies connects these ontologies in a certain way specified by the user.

Another problem occurs when domain experts want to create ontologies, but do not have proper knowledge in the field of creating ontologies in general. These hurdles may slow down or even block the progress of a project. That is why the purpose of this thesis is to create a batch recommender for ontologies that are created from scratch. The idea is that a user can create their domain ontology and after doing so, they get a full recommendation of terms from existing ontologies that they can use for their created ontology. The aim is to provide a fast prototyping experience with minimal effort for the user and still being able to make use of existing ontologies, which may cover the created concept.

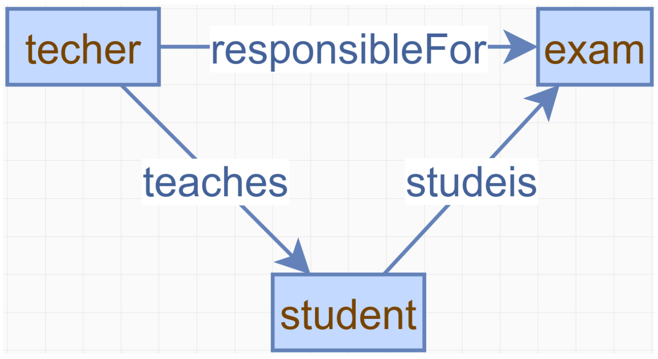


Figure 1.2: Example education ontology including two spelling errors created with Neologism 2.0.

The example in Figure 1.2 shows an ontology created with Neologism 2.0 [LGC*20, LGC*21]. Neologism 2.0 is called Neologism in the further course of this thesis. Neologism is an ontology editor for creating and publishing ontologies and allows other users to access them. Notice that in the example spelling errors occurred during the creation of the *techer* node and the *studeis* relation. Sending each of these terms to a search engine would lead to wrong recommendations, as the terms were not correctly specified. This would slow down the process of obtaining *good* recommendation results for the created ontology. We aim at providing recommendations for each of the nodes and relations that fit the users' needs in a good way, while still being able to cover any domain. This is a specific example for the education domain, so we need a database with a variety of ontologies to provide general information for any kind of domain. For this purpose, we have chosen the LOV [VAPV17] service, as it offers the biggest open-source data collection for ontologies, which is also available online [VAPV11]. The LOV service offers a simple API to access and search for ontologies.

The aim of this thesis is to create a customizable batch recommender framework and integrate this into the structure of an ontology editor with its functionalities and visualization. We aim at providing the option to not only create an ontology from scratch but to link it to terms from existing ontologies that are enhanced with metadata. With that we lift the created ontology to the Semantic Web i.e., replacing created nodes and relations with classes and properties that already exist. This fosters a better description of the created ontology and a better possibility of processing it. To this end, we provide a framework to easily add other recommenders to the batch recommender to receive user-requested recommendations that are not covered by LOV. The lifting process should be customizable as we believe that the recommendation area of ontologies is use-case and user-dependent. This is why the user should be able to influence the received results.

1.2 Thesis Goals

We solve the problems described in the previous section with the following approach. We provide a fast prototyping experience where the user creates an ontology, which is then lifted to the Semantic Web. As a basis, we use an ontology editor, which provides CRUD operations for ontologies. The lifting process is done, by mapping chosen terms, which already exist, from a recommendation list to the terms inside the created ontology. For the recommendations, we put special focus on the LOV service, as it provides the largest open-source dataset for ontologies. We still offer easy integration of other recommenders and local ontologies into the batch recommender framework. This ensures that the user-specific and application-specific recommendations can be adjusted as needed. The questions to be asked in this thesis to create a batch recommender for fast prototyping of ontologies are the following:

RQ 1: **How to integrate proper semantics in the fast ontology prototyping process in a simple way?**

RQ 2: **How does the infrastructure have to look like?**

RQ 3: **How to provide good recommendations?**

RQ 4: **How to integrate a batch recommender into an ontology editor?**

The goal is to create a customizable batch recommender framework, which expands the structure of an ontology editor, that does the following. We continuously support the user with possible correctly specified terms for their nodes, while they are creating or editing nodes or relations in the ontology. After the user created the ontology, we give the user the option to run a batch recommendation process. The results of this process are recommendations for each node and relation within the created graph, which are displayed to the user. The recommendation results are based on a ranking, so it supports the user in creating the ontology with its desired meaning. The ranking is based on specific metrics used to address certain issues. The results are displayed in a simple and readable manner, such that the user can choose the best fitting term for every node. After performing the lift operation based on user decisions, the terms within the graph are connected to terms in existing ontologies

The thesis is structured as follows. Chapter 2 presents literature related to our approach. Chapter 3 and Chapter 4 present the developed concept and realization with the integration of the batch recommender into an existing editor. We present an evaluation of our work in Chapter 5 before we conclude the thesis in Chapter 6. The goal is to create a batch recommender for ontologies, which is a standalone solution. We integrate the recommender into an existing ontology editor.

2 Related Work

We consider the evaluation of recommendations to be very subjective as many different people may have different opinions on which ontology terms are the best to describe things. In addition, the visualization may also be an issue, which can influence the user experience, especially considering the evaluation process of this thesis. In the following sections, we describe the approach we have chosen and the reasons why we choose it. Regarding the specified research questions in Section 1.2, this thesis could include a variety of topics. We distinguish our approach from other approaches and explain the limitations of this thesis. There are areas, which are important, but are not part of this thesis, like:

- Ontology search
- Ontology evaluation
- Ontology visualization

This is outlined in more detail in Section 2.1. Other areas, which are closely related to our thesis where we analyse the state-of-the-art are the following:

- Ontology creation
- Ontology ranking
- Ontology recommendation
- Ontology matching
- Semantic annotation

The areas are explained in Section 2.2, Section 2.3, and Section 2.4, respectively. As ontology ranking and recommendation are closely related, we discuss these topics together. This applies also to ontology matching and the semantic annotation tools we found. During our research we discovered that all the mentioned areas are very closely related and can overlap e.g., search engines may use ontology matching, semantic annotation, or evaluation strategies to determine scores for rankings. On the other hand, also recommenders may integrate search engines.

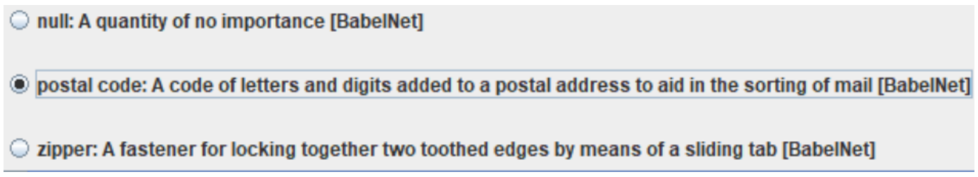
2.1 Searching, Evaluating, and Visualizing Ontologies

In this section, we give a brief overview of ontology search engines and describe evaluation and visualization techniques for ontologies. Ontology search engines return users a ranked list of either ontologies or terms of ontologies for a given query. With that, it helps users to find and reuse existing knowledge on the web, which benefits communities by establishing consensus on domain conceptualizations [KVKL20]. Ontology evaluation is the task of measuring the quality of an ontology using specific metrics to compare the quality of different ontologies [Vran09, TaAr07]. The ontology visualization area analyses the most effective ways to display the information of ontologies to the user [KHL*07].

Currently there exist many search engines for ontologies like e.g., Swoogle [DFJ*04], OntoSearch [TASB05, ZVSl04], Ontokhoj [PSLP03], OntoSelect [BuEi08] and Linked Open Vocabularies (LOV) [VAPV17]. Swoogle is a search engine that focuses on search functionality. It offers an advanced search, where the user can specify constraints for SQL queries, for faster searching. OntoSearch is based on observations by ranking ontologies focusing on the overlap between query terms and index terms, the ratio of class vs. property definitions, and the level of integration between ontologies. While OntoKhoj applies ontology classification in form of text classification algorithms and tools. The classifier determines whether a new ontology belongs to a particular topic using confidence. The ontologies are retrieved by crawling the web. OntoSelect uses the GoogleApi to crawl for ontologies and ranks the results based on specific metrics. Most of these metrics focus on the structure of ontologies. The LOV is not a search engine like the others but enables searching for vocabulary terms (class, property, datatype). It is a service that offers the largest open-source database for ontologies on the web, currently covering 723 ontologies [VAPV11]. All these search engines focus on the search criterion, but also include specific ranking mechanisms. Some of them use ontology matching or classification, while others perform SQL queries specified by the user. Yet, they do not provide functionalities in regard to the batch recommendation and ontology creation aspect, as the focus is on providing search functionalities. As the LOV service offers the largest open-source dataset of ontologies, which also offers an API for requests, we build our recommender using LOV.

The topic of ontology evaluation is covered broadly, as many authors proposed many different metrics to evaluate ontologies [Lant16, IvPo20, Verm16, MVMS16, BBEI16, Degb17]. This also applies to ontology ranking and therefore we do not create new ranking methods. Based on the information, we can get from LOV we create recommendations for each node and relation inside a created ontology. Considering the ontology reuse aspect, we offer the user the usage of existing ontologies, that may fit the specified terms. The option of not accepting the provided recommendation remains. Finally, the created ontology should be published with the used ontologies providing more value, processability, and new knowledge.

The aspect of visualization is also a very important one, as there is a lot of work considering visualization techniques [DLSP18, KTV*08, LPKM17]. This area is also out of the scope of this thesis and may be considered for future work, where different



○ null: A quantity of no importance [BabelNet]

[● postal code: A code of letters and digits added to a postal address to aid in the sorting of mail [BabelNet]]

○ zipper: A fastener for locking together two toothed edges by means of a sliding tab [BabelNet]

Figure 2.1: Visualization of a recommendation containing a label, a description, and a [BabelNet] tag. This visualization concept can be used for displaying information of a recommendation [LPKM17].

visualization techniques can be evaluated. We choose a simple technique provided in [LPKM17], which can be seen in Figure 2.1. This visualization technique fits our needs and can be easily integrated into the provided structure.

2.2 Ontology Creation Tools

Ontology creation is an area, where we focus on improving the quality of created ontologies. Ontology creation is the task of creating an ontology with its corresponding classes and properties and organizing these in a meaningful way [CrCu05]. There are many different ontology editors like e.g., Protege [NSD*01], Neologism [LGC*20, LGC*21], NeonToolkit [HLS*08], OntoEdit [GFCo06] or Swoop [KPS*06]. In Protege [NSD*01], rdfs properties can be used for a better description of terms. Neologism [LGC*20, LGC*21] offers a fast and simple way of creating and managing ontologies. The NeonToolkit [HLS*08] combined with the Watson Plugin (`http://kmi.open.ac.uk/technologies/name/watson/`) offers at least to search for descriptions inside the editor. It allows the user to reuse ontologies provided by Watson. OntoEdit [GFCo06] is focusing on a clear structured process in three phases for creating ontologies, which needs more effort from the user and therefore is more time-consuming than Neologism. In addition, it offers collaborative work on ontologies. In Swoop [KPS*06], you have a search engine for terms. Yet, no editor offers a recommender for possible ontologies covering the complete created ontology. As Neologism is suitable to create and manage ontologies in a simple and quick manner, it makes sense to integrate the recommender into this editor.

2.3 Ranking Algorithms and Recommendation Tools for Ontologies

We aim to have a simple design for our batch recommender offering a modular framework. The modularity supports the integration of other recommenders as well as the adjustment of existing ones. It also offers the integration and adjustment of metrics to achieve an individual ranking of recommendations. This is explained in Chapter 3 in more detail. In this section, we compare state-of-the-art recommendation and ranking concepts and discuss the different approaches. Ontology ranking is the process of ranking different ontologies or terms of ontologies based on a set of metrics or analytical measurements that are applied on a set of available ontologies [SASi11]. Ontology recommendation uses ontology ranking but focuses on providing terms or ontologies that may fit a given context [MJO*17]. As far as we know, there is currently no batch recommender for ontologies available that focuses on enhancing the ontology creation process. The available batch recommenders are either domain-specific or vary in focus. In the following, we give an overview of different ranking algorithms.

A ranking algorithm is introduced in [Glei20]. The algorithm is focused on entities, so it is not applicable in the model case yet. For future work, it could also be considered if the approach could be transformed to be used on ontologies. Another work [SASi11] reviews different ranking algorithms, which are briefly described in the following:

- **AktiveRank** is based on different analytical measures using the graph structure of ontologies.

- **The content-based ontology ranking algorithm** obtains a list of ontologies from a search engine. The received ontologies are ranked according to the number of concept labels in those ontologies, which match a set of terms extracted from WordNet.

- **The Ontology Structure Rank Algorithm** is ranking the ontologies using class names, semantic relation, and ontology structure. The weights of measures can be adjusted according to the users' needs and the importance of the application.

- **The Semantic-aware importance flooding algorithm** also converts the ontology into a directed graph and performs an iteration fixpoint computation to calculate the importance of nodes.

The paper states the disadvantages of increased time complexity as well as difficulties retrieving suitable results, based on different input terms. This is something that could be used for future work, as these algorithms consider the internal structure of the ontology (ontology matching), which we do not focus on.

As we need to rank recommendations, we create specific metrics based on the recommender providing the recommendations, the domain specified by the user, a prefix and suffix matching, a LOV popularity measuring, and a common vocabulary metric. These default metrics are addressing certain issues specified in Chapter 3. The metrics and their weights can be adjusted or removed, and new metrics can be added. The emphasis in creating these metrics is simplicity. More extensive ranking methods can be added, as the metric system is designed in a modular way.

In the following, we present some recommender systems, which also include ranking systems. The ranking algorithms discussed previously were mainly focused on ranking. There is no clear classification, where to put which work, as the areas overlap. There exist different recommender systems, but currently none of those covers a batch recommendation for ontologies focusing on the ontology creation process. Either the work is domain-specific, has a different focus, or needs a lot of data.

The authors of [TaAr07] rank ontologies based on their content and their relevance to a set of keywords as well as user preferences. The aim is to evaluate ontologies using their instances and schemas. The input is based on a keyword and the populated ontology, which then is expanded using WordNet for related keywords and forwarded to the search engine Swoogle. The related ontologies are then evaluated and ranked based on specific metrics.

In [ANS*07] the authors perform query expanding via search engines like Wikipedia and Google. They try to identify the most relevant terms based on a specific topic and then query the ontology repository. They rank the results based on how many times each query term appears in the labels of classes, labels of properties, and in property values for datatype properties (e.g., string-values properties). In addition, normalization is performed, to consider ontologies that cover a variety of terms more important than ontologies that cover the same terms very often.

In [LPAl11] the authors focus on architecture for semantic recommenders for semantic datasets. Therefore, they transform datasets into a graph structure and assign weights to nodes and edges for clustering. They use two different recommender approaches, a memory-based recommender, and a model-based recommender. The first one is focused on calculating paths starting from a set of given input entities. Reachable entities from the input are ordered according to a similarity rating that is based on edge weights. The second approach focuses on creating a model from the dataset and clustering similar entities. The recommender strategy is based on the respective scenario and query. The results are then aggregated in a unified result list.

The authors of [GiWi09] focus on the biomedical area and need a large amount of user specification considering the biology domain. The concept of [Zieg04] focuses on similarity and neighbourhood measurements. Creating specific trust and rating functions.

ESKAPE focuses on creating semantic recommendations for dataset input [PPP*18, PLMe19]. They extended it with a structural recommendation for a term, i.e., for a given keyword it suggests a graph, with terms that have specific relationships to that keyword. The aim is to describe the keyword properly with its surrounding relationships [PLMe19]. This approach has a different focus, while we keep the developed concept of the user and enhance it with existing ontologies, ESKAPE focuses on enlarging the concept or creating a semantic model based on a given dataset. In ESKAPE the recommended concepts are lists that are not bound to ontologies. We focus on recommending existing terms from ontologies for the created ontology, that fit the desired meaning of the user. In addition, we want to provide a modular environment to simply adjust the recommendation result received.

BioSS [MVPP14] and BioPortal [MJO*17] are both approaches, which offer multiple keyword searches showing ontology sets that could cover a certain amount of the input keywords. Currently, our approach is focused on general ontologies, with an option to specify a domain, and therefore, these recommenders would not lead to sufficient results considering general terms. In contrast to BioPortal and BioSS, our approach focuses on improving the quality of ontology creation and recommender integration, while BioSS and BioPortal focus on annotation, searching, and recommending in general.

The approach of BioPortal or BioSS could be integrated into the batch recommender in two ways. One way would be the realization of the metrics of BioPortal and BioSS and analyse if and how they improve the ranking of recommendations. The second way would be to integrate these two recommenders as sub-recommender for our batch recommender. With this, we could improve the usage of the domain that can be specified by giving possibly more accurate results regarding the biological domain.

2.4 Ontology Matching and Semantic Annotation

In this section, we give an overview of state-of-the-art ontology matching and semantic annotation tools. Ontology matching focuses on finding correspondences between semantically related entities of ontologies. These correspondences may stand for equivalence or relations, such as consequence or disjointness between ontology entities [ShEu13, EuSh07]. Semantic annotation related to ontologies is the task of tagging ontology class instance data and map it into ontology classes [ReHa05]. It can be considered as the task of linking entities to their semantic description [PKK*03]. The mapping of terms from existing ontologies on a created ontology can also be considered as semantic annotation, which is done in the BR. We enrich terms from a created ontology with the metadata from existing terms. In addition, we give the user the opportunity to choose the best fitting terms and adjust the retrieved terms.

The ranking of the recommendations could be improved through ontology matching as well. We do not focus on ontology matching considering the internal structure of an ontology, as it is much more complex and out of the scope of this thesis. Integrating the structure of ontologies into the ranking may be problematic in regard to custom relations that are not frequently used. Matching unknown custom relations that are not frequently used to existing ones is a hard task to do due to missing information. In addition, it is not clear, if it leads to better results, as semantic meanings are very subjective. It needs to be evaluated, in which cases ontology matching could lead to better results. In this section, we distinguish our batch recommender from semantic annotation tools and list current problems of ontology matching attempts.

Ontology Matching

Performing ontology matching is a challenging task. Matching systems are typically evaluated against a reference alignment. One example of a matching tool is proposed in [HPPa20], which is mainly based on the ontology structure to create matches. The inputs may be two URLs of the ontologies to be matched together with an URL referencing an input alignment. The tool focuses on matching different versions of the same ontology. They use matchers and filters, where matchers create specific results and filters attach confidence to these results and extract them based on certain thresholds. For this procedure training data is needed. They measure their tool based on the Ontology Alignment Evaluation Initiative (OAEI, which is a coordinated international initiative to forge a consensus of evaluating schema or ontology matching methods), which provides different datasets for measurement. Entities, which do not appear in the reference alignment, cannot be judged.

The authors of [ASPT10] use an RDF graph to exhibit the structure of ontologies for the purpose of entity verification. The aim is to find common entities across different ontologies in the same domain. They also consider e.g., the distance between ontologies using entropy-based distribution. The work of [GLCh05] is based on a weight vector matching algorithm. The algorithm takes a benchmark ontology and an evaluating ontology as input. These ontologies are compared to each other, based on a score vector for each concept (node) within the ontology graph.

A broad variety of tools for instance and schema matching can be found on the OAEI website `http://oaei.ontologymatching.org`. As we do not focus on instance or schema matching, it could be considered for future work. This area remains a big research topic, where the advantages and disadvantages of using the ontology structure need to be discovered. It needs to be evaluated, whether the further complexity taking into account different structural measures, leads to better semantic recommendation results. Currently, there are problems with matching tools, which we describe in the following.

The authors of [Hopf20] introduce a gold standard dataset for ontology matching. Therefore, they focus on the schema of large, automatically constructed, less well-structured KGs based on DBpedia and NELL. For this, they manually screened both databases to create a mapping of similar terms between the databases. The developed gold standard can be used for testing KG matching to gain a deeper understanding and discovery in this domain. The authors discovered that the major limitation of the current benchmark is their lack of representation of real-world KGs. The authors state that the need for specialized matching tools remains significant, to tackle the problem of KG matching. Three problems were distinguished of the current ontology matching tools:

1. Current tools can produce high-quality results for well-formed ontologies, such techniques are not as well-performing when applied on KGs that lack textual descriptions.

2. Many ontology matching systems utilize structural knowledge available in well-structured ontologies to refine their alignments. Structural-based techniques can be difficult to apply when lacking schematic information.

3. Matching strategies used when two resources are from a specific domain setting are not applicable for domain-independent settings where classes contain information about real-world entities described with different terminologies.

Semantic Annotation

The area of semantic annotation is closely related to this thesis, as we annotate terms with terms from existing ontologies. These existing terms propose metadata and descriptions, which may fit the created model that is not annotated yet. There are many annotation tools described in [ReHa05] and in [UCI*06], which focus on annotating documents or data instances. The authors of [EMSS00] focus on semi-automatic engineering of ontologies from text. The tool allows the annotation of facts within any document by tagging parts of the text and semantically defining its meaning while focusing on a selected instance. It supports the choice of the most specific concept for the selected instance. Annotators like [LiDi05] and [PBS*06] focus on processes and multimedia content, respectively. Both proposed concepts have a general ontology or core ontology, which is used for annotating the specific domain.

Our goal is to gather a created ontology and create a proper recommendation based on all the nodes and relations of the ontology and not just parts of it (considering the example in Figure 1.2). The described annotators do not offer the functionality we provide in regard to improving the quality of an ontology using terms from existing ontologies. Our recommender focuses on replacing terms of a created ontology and not a data instance, with fitting terms from existing ontologies to the created ones.

Generally, we think that the evaluation of recommendations is subjective, which is why we keep the recommendation procedure simple. We do not perform ontology matching in detail, as we do not consider the graph structure of ontologies, this is out of the scope of this thesis. We do not know, if this would yield better recommendations, too. We use the LOV [VAPV17] service presented in this chapter to retrieve recommendations for various domains. For our batch recommender, we provide a modular environment for customization (adjusting metrics and recommenders and adding local ontologies) based on the desired needs. Our batch recommender is integrated into the ontology creation tool Neologism [LGC*20, LGC*21], which we presented in this chapter. For this, we use a similar visualization technique as in [LPKM17] to display the information of terms properly. The focus is on improving the quality of the ontology creation process and thus the overall quality of the ontologies created.

3 Concept

In this chapter, we present the concept of the batch recommender, in the following called BR. We first show the general framework of the BR and then explain the BR architecture in detail. These two sections tackle RQ2. After this, we show the workflow of the BR and how we integrate the BR into an ontology editor with the chosen visualization technique. Finally, we explain the formulas of the BR and the metrics for ranking the different recommendations. The goal is to give a user without knowledge of ontologies the possibility to use existing solutions and to adapt the desired result based on their needs.

To understand the following sections and chapters, we call the *classes* and *properties* of an ontology that has not been processed by the BR, *nodes* and *relations*. A processed ontology graph lifts nodes and relations to classes and properties from existing ontologies. Generally, we treat properties like classes. There are also the following definitions to simplify understanding:

Definition 1 *A **keyword** is the name of a node or relation of the input model.*

Definition 2 *A **label** is the name of a class or property.*

Definition 3 *A Uniform Resource Identifier (**URI**) is a unique identifier of a class or property.*

Definition 4 *A **comment** contains a description of a class or property.*

Definition 5 *A **recommendation** contains a list of labels, a list of comments, the URI, and the creator of the recommendation.*

Definition 6 *A **sub-recommender** is a function, which returns a list of recommendations for each keyword of the input ontology.*

Definition 7 *A sub-recommender can be the **creator** of a recommendation.*

Definition 8 *The **batch recommender** returns the top recommendations (lists of classes and properties) based on the integrated sub-recommenders for each node and relation of the input ontology.*

3.1 Recommender Framework

In this section, we explain the overall framework of the BR. Although we have a variety of possible use cases, we focus on ontology editors as the main purpose of this thesis is to enhance the ontology creation process. The BR can be addressed via an interface to allow easy use by other programs, humans, or machines. The input of the interface is the ontology created by the user, while the output is a list of possible classes or properties for each node or relation inside the input ontology, respectively. The BR is not supposed to be a new domain recommender since there are many recommenders already available online (see Chapter 2), but a customizable recommender that integrates existing approaches based on the user's needs. The aim is to enhance the ontology prototyping process and create a simple and fast way to gather good recommendations for ontologies that were created from scratch. The framework is shown with an ontology editor to enhance its understanding in Figure 3.1.

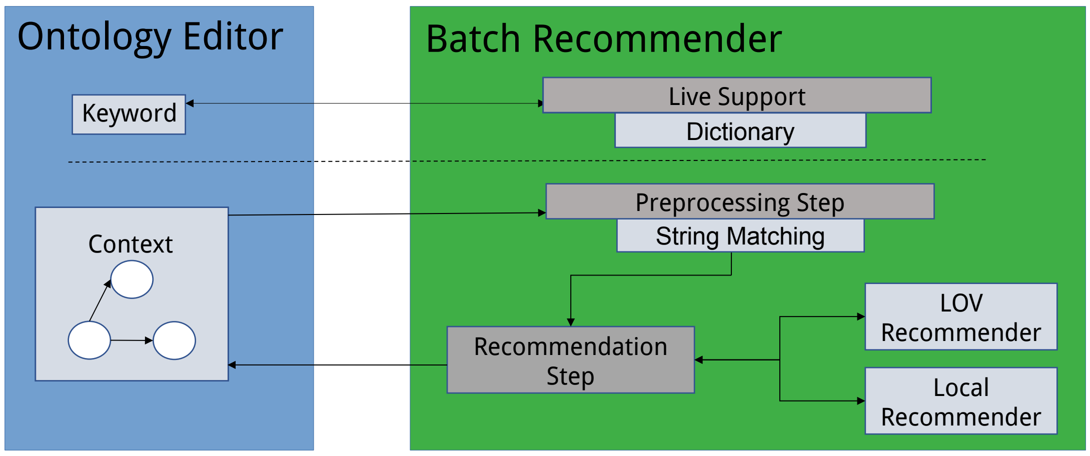


Figure 3.1: The standalone BR framework with example use of an ontology editor. The dashed line separates the two phases.

The BR is divided into two phases, which are explained below. The phases are superimposed because the first phase supports the outcome of the second phase. The two phases deal with RQ1:

1. *Context-free* phase: Give the user *Live Support* for correcting spelling errors, while creating an ontology.
2. *Context-development* phase: This phase is divided into two steps:
   a) *Preprocessing Step*: After the ontology creation is complete, each node and relation keyword is put into a consistent format through string operations.
   b) *Recommendation Step*: Create an overall recommendation of classes and properties for the entire the input ontology whose nodes and relations were transformed in the first step.

The general BR framework is shown in Figure 3.1. The *Live Support* is responsible for phase 1 and directly communicates with the ontology editor, while phase 2 is divided into two steps. These two steps are the preprocessing and the recommendation step. In the second phase, the input is received in the preprocessing step, while the output is generated in the recommendation step. Inside the recommendation step all sub-recommenders are used, which are two by default:

- The **LOV Recommender** recommends ontologies using the LOV API [VAPV17].
- The **Local Recommender** recommends from a predefined ontology set.

The framework allows the user to add or remove sub-recommenders based on their specific needs as well as to change preferences for each sub-recommender separately. In addition, the user can add custom ontologies to the predefined ontology set. This is explained in detail in Section 4.1.2.

3.2 Architecture and Modules

In this section, we explain the BR architecture in detail. We explain the modules and their responsibilities during the preprocessing and recommendation step. Generally, we offer the user integration of metrics, sub-recommenders, and ontologies of their choice, by exploiting existing approaches. For this, we create several modules that can be adjusted by the user based on their specific needs. Figure 3.2 shows the overall architecture of the BR. We have modules serving different purposes, which we explain in the following.

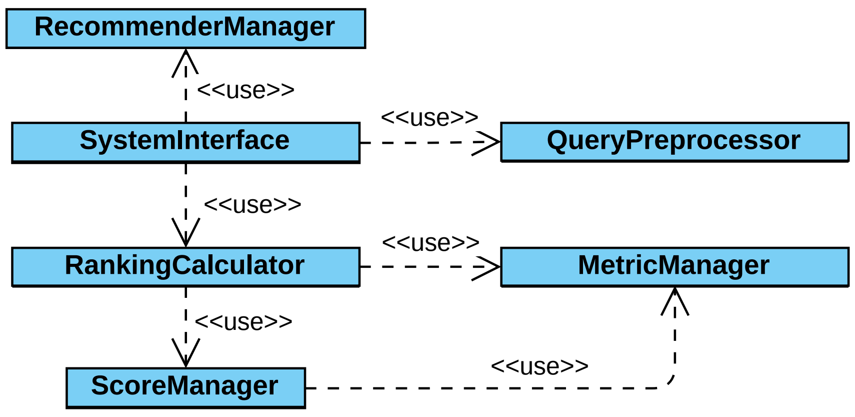


Figure 3.2: BR architecture with the different modules and their connections.

The *SystemInterface* is the starting point of the BR. It receives the input, which are the nodes and relations of an ontology, and dictates the flow of the BR. It communicates directly or indirectly with all other modules that are explained in the following.

The *QueryPreprocessor* applies a clean-up process to the node and relation keywords of the input ontology and is the second point through which the BR flow passes by. This module is responsible for the preprocessing step described in the previous section. This is done because of problems that can occur when creating an ontology from scratch. These problems are explained in detail in Section 3.3.1.

The next module is the *RecommenderManager*, which is focused on managing the sub-recommenders. It serves the purpose of providing recommendations from all sub-recommenders for all nodes and relations of the input ontology. As explained in Section 3.1 we have the *LOVRecommender* and the *LocalRecommender* implemented by default. The RecommenderManager is also the reference point for integrating new sub-recommenders that should be used inside the BR or disabling the existing ones.

The *RankingCalculator* is focused on calculating the scores for the recommendations provided from the RecommenderManager. For the calculation, the *MetricManager* and the *ScoreManager* are used. While the MetricManager manages all the metrics used for the calculation and their corresponding weights, it is also the reference point for integrating or deactivating the metrics used for the score calculation. The ScoreManager, on the other hand, manages the scores calculated by the various metrics specified in the MetricManager. The ScoreManager also uses the MetricManager to obtain the weights of the metrics required for the final calculation of the score.

3.3 Workflow and Integration

In this section, we explain the workflow of the BR with its integration into an ontology editor of our choice. We also explain the visualization technique and its integration into the ontology editor. With the adjustment of the workflow, we tackle RQ1 here.

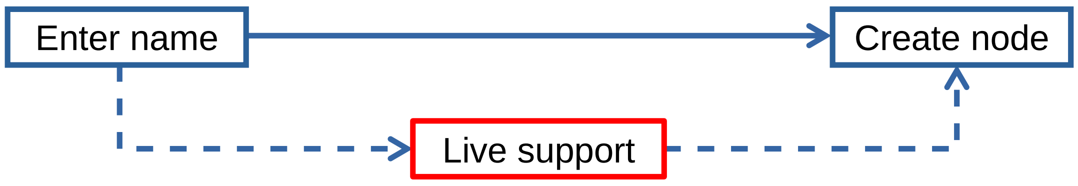


Figure 3.3: The enhanced process of creating a node in an ontology editor. The red box shows our added contribution.

The first starting point of integration into an editor is to adjust the process of how nodes and relations are created and edited with the BR. At this point of the ontology creation process, the first phase is applied. Figure 3.3 shows the adjustment of the node creation process, which also applies to relations. Live Support is added to help during the creation and editing process of nodes and relations within an ontology. It is helpful for the batch recommendation process as it prevents and corrects spelling errors. The results may be worse if the user provides an ontology with spelling errors, as this will lead to incorrect recommendations. With the correct specification, the query process is improved for the second phase.

The adjustment of the ontology creation process can be seen in Figure 3.4. During the ontology creation, the node creation from Figure 3.3 is repeated multiple times. After the user finished creating an ontology, the communication process with the BR interface can be started. The BR returns a list of recommendations for each node and relation separately. These lists of recommendations are then displayed inside the ontology editor and allow the user to choose from the provided results. Finally, the user can update the ontology with the selected classes and properties from the provided recommendations or keep their nodes and relations. This final step performs the lifting process of the ontology, where the ontology which was created from scratch is lifted to the semantic web, by replacing created nodes and relations with existing classes and properties.

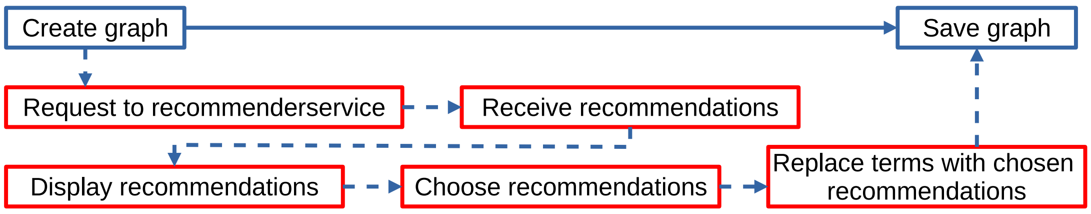


Figure 3.4: The process of the improved ontology creation process with batch recommendation. The red boxes show our added contributions, which are the communication with the BR interface and the interactions needed for applying the results.

We have multiple steps between the request and receive phase, which we briefly explain in the following. The missing steps are the steps of the context-development phase, which are inside the workflow of the BR and can be seen in Figure 3.5. After receiving the request, the preprocessing step starts, which assists the recommendation step. In the preprocessing step we address several issues that may arise when creating an ontology or a dataset in general (see Section 3.3.1). These issues are solved, by transforming the input words of the nodes and relations via string matching. The results worsen, if the user provides an ontology with terms including characters with no meaning, as it leads to wrong recommendations. With the preprocessing in the first step, the query process in the second step is enhanced.

The preprocessing step is followed by the recommendation step, where we receive recommendations from all existing sub-recommenders based on the nodes and relations of the input ontology. After gathering the recommendations, the final score for each recommendation is calculated based on the existing metrics and their corresponding weights. Each metric calculates its score separately. Finally, the scores are normalized and the top recommendations for each node and relation are returned. The nodes and relations can then be lifted to classes and properties from existing ontologies.

In the following we tackle RQ3. For the results of the batch recommendation, we aim to provide the possibility for the user to choose the best fitting term for a node. The terms are displayed in a specific order based on a score for each recommendation. This score is calculated via various metrics, which can be adjusted using their corresponding

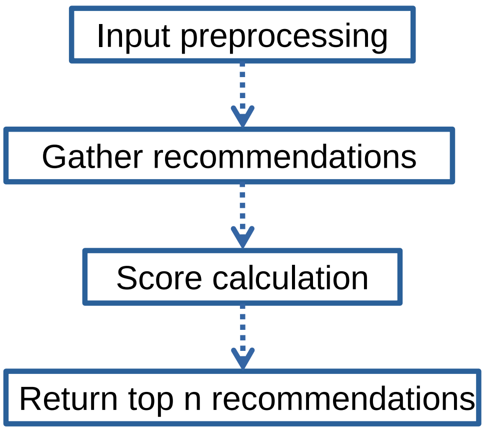


Figure 3.5: The BatchRecommender process overview.

weights. This score calculation is the key for ranking the recommendations in the BR. As we think that it is very subjective if a recommendation fits the desired meaning, we provide the functionalities to improve the recommendations based on certain interests or use-cases, by changing or adding metrics and their weights. This customization can also be done by adding local ontologies and sub-recommenders. The developed metrics and used rankings are explained in Section 3.4. In addition, the user can save time, if they just intend to accept the best recommendation provided from the BR as we preselect the first recommendations. In regard to visualizing the recommendations properly, we choose a visualization concept similar to Figure 2.1. This decision is based on the research from Section 2.1. This visualization offers the user the option to see the results directly. The information of each recommendation is displayed by replacing the BabelNet tag with the ontology URI and adding the calculated score, keeping the label and description of the recommendation.

After the user decided, which recommendations they want to keep based on the given recommendations, they can directly lift the created ontology to the Semantic Web. Concretely, replacing nodes and relations with the chosen classes and properties from existing ontologies and thereby lifting the ontology to the Semantic Web. It is also possible to choose a better fitting term from the list or even keep the term the user entered (e.g., if the results are not sufficient, or there are no recommendations available).

We additionally give the user the option to specify a certain domain to get better results. It is optional but may lead to increased user satisfaction, as the user already knows in advance what kind of ontology they intend to create. We implement a metric that uses the domain, if it has been specified by the user, for score calculation. We leave the opportunity for future work, to use the domain tag for further improvement of the recommendations. It could be used to integrate other domain-recommenders. One example may be the integration of BioPortal [MJO*17] for the biological domain.

3.3.1 Input Preprocessing and Live Support

In the following we tackle RQ3. As mentioned beforehand we perform preprocessing on the input ontology which serves as the query input for the BR. We use the keywords from nodes and relations of the ontology as input. There are ways to specify these keywords during ontology creation, which may lead to several problems. Regarding our own experience and the analysis of different datasets in [PLP*18], we decide to tackle some of the issues that can occur to provide better recommendations. The authors of [PLP*18] define different problem classes where a label can belong to. As many of the classes are challenging and out of the scope of the thesis, we focus on problem classes, which have a big influence and can be solved in a simple way. In the following, we show the excerpt of the problem classes we distinguish for our use case from [PLP*18] and afterwards describe the solutions we choose for these classes.

- **Misspelling:** The person who labelled the data made a simple mistake. Examples are *Acess Point*, *Telehphone Number*, etc.
- **Splitting Characters:** Beside white spaces in labels, some labels contained special characters (e.g., '_' or '-') to split words.
- **Camel Case Input:** Similar to splitting characters, some persons tend to split words using the camel case syntax (e.g., StreetNumber).

We solve the problem classes of *Splitting Characters* by replacing the splitting characters with spaces (e.g., *telephone_number* is transformed into *telephone number*). For the *Camel Case Input* problem class, we distinguish all upper-case letters (except the first one) and put a space in front of all upper case letters (e.g., *telephoneNumber* is transformed into *telephone Number*). The transformation of a class or property keyword is done by applying the following formula:

$$f : String \mapsto String$$

The function f transforms a keyword $k \in K$ into a keyword k', which is neither in the Splitting Characters problem class nor in the Camel Case Input problem class.

$$\begin{aligned}
&f(k_i) = k_i', \forall k_i \in K,\\
&\text{where } K = \mathrm{C} \cup \mathrm{P},\\
&\text{and } C = \text{class keywords}\\
&\text{and } P = \text{property keywords}
\end{aligned}$$

Regarding the *Misspelling* problem class, we use a dictionary and let the user decide whether to use the suggestions provided from the dictionary (see Section 4.2). We choose this way, as we do not want to abbreviate the input strings from the user more than correcting the last two problem classes but give the user the choice to correct mistakes themselves.

3.4 Formal Recommender and Metrics Description

In this section, we describe the recommender formulas of the BR, which also tackle RQ3. These formulas are divided into two categories:

- The general BR formula with the ontology keywords as input and the recommendation lists as an output as well as the corresponding formula for any sub-recommender.
- The metrics that we developed to create a better ranking of the recommendations.

3.4.1 Recommender Formulas

In the following, the BR and the sub-recommender formulas are explained. While the BR formula explains the overall input and output of the BR, the sub-recommender formula explains the underlying concept of the sub-recommender functions.

Batch Recommender Formula

Consider $K=$ the set of keywords from the nodes and relations of the input ontology. $T =$ the set of terms (all classes and properties from existing ontologies). A class or property $t_i : U \times L \times C \times R$, where $t_i = (u_i, l_i, c_i, C_i)$ and $u_i \in U, U =$ set of URIs, $l_i \in L, L =$ set of labels, $c_i \in C, C =$ set of comments, and $C_i \in R, R =$ set of available sub-recommenders (creators), $i \in \mathbb{N}$, then we have the BR formula as following:

$$f : \overbrace{(k_1, \dots, k_m)}^{K^m} \mapsto \overbrace{(t_{11}, t_{12}, \dots, t_{1n}), \dots, (t_{m1}, t_{m2}, \dots, t_{mn})}^{(T \times \mathbb{R})^{m^n}}$$

$\#m$: number of keywords

$\#n$: number of final recommendations per keyword

The number of final recommendations is $\leq n$ as there may be cases with less than n or no recommendations. The batch recommender formula maps each of the m keywords on $\leq n$ sorted recommendations with their corresponding scores. The recommendations are sorted based on the score.

Sub-Recommender Formula

The sub-recommender formula is used for calculation by each sub-recommender $r \in R$, where R is the set of all sub-recommenders. Thereby each sub-recommender may have a different number of recommendations available for a keyword $k \in K$. The sub-recommenders do not assign a score to the recommendations but retrieve the recommendations. We define the formula of a sub-recommender as following:

$$f : \overbrace{(k_1, \ldots, k_m)}^{K^m} \mapsto \overbrace{\{(t_{11}, t_{12}, \ldots, t_{1n_1}), \ldots, (t_{m1}, t_{m2}, \ldots, t_{mn_m})\}}^{T^p},$$
$$\text{where } p = \sum_{i=1}^{m} n_i$$
$$\#n_i : \text{number of recommendations of keyword } k_i$$
$$\#p : \text{number of all recommendations of sub-recommender}$$

3.4.2 Metric Formulas and Final Score Calculation

We have different metrics to calculate the score for each recommendation. The metrics provided are the basis for ranking the different recommendations, with which we combine the input ontology so that the recommendations are:

- domain related, which means we integrate the domain specified by the user into the ranking, checking label and comment of the recommendation for the domain (`DomainMetric`).
- related in such a way that simple semantic errors not intended by the user do not occur (`PreSufMetric`).
- description related, which means recommendations with a better description through labels and comments are preferred (`DescriptionMetric`).
- recommender related, which means that we take into account the user's preferences regarding the specified recommenders (`CreatorMetric`).
- related in such a way, that the set of different ontologies inside the recommendations for each keyword is minimized or maximized (`CommonVocabMetric`).
- LOV related, which means we use the metadata provided by LOV for the ranking (`LOVMetric`).

These metrics serve for the calculation of the overall score. For this calculation we need to declare the following:

$$s = \sum_{r \in R} p_r$$
$$\#s : \text{number of all recommendations}, \forall r \in R$$

The calculation of s is needed to cover the number of terms, which are used inside the metrics. Note that p is reused from the previous section marked with the corresponding recommender $r \in R$ as the index. The default metrics available are listed below:

DomainMetric

Let $d \in D$ be a chosen domain, where D is the set of all available domains. Then the `DomainMetric` formula is defined as following:

$$f : T^s \times D \mapsto \mathbb{R}^s,$$

where we assign each recommended term $t \in T^s$ a value from $\mathbb{R}$ based on the chosen domain $d \in D$.

$$\begin{aligned}
&f(t_1, \dots, t_s, d) = (x_1, \dots, x_s)\\
&\text{and } x_i = \omega_\alpha * g(l_i, d) + \omega_\beta * g(c_i, d),\\
&\text{where } g : S \times S \mapsto \{0, 1\},\\
&\text{and } g(a, b) = \begin{cases} 1 & \text{if a contains b} \\ 0 & \text{otherwise} \end{cases},
\end{aligned}$$

where the weights ω_α and ω_β focus on the preferences of the occurrence of the domain either inside the label or the comment section of the recommendation.

CreatorMetric

Let $r \in R$ be a sub-recommender. Then the `CreatorMetric` formula is defined as following:

$$f : T^s \mapsto \mathbb{R}^s,$$

where we assign each recommended term $t \in T^s$ a value from $\mathbb{R}$ based on the sub-recommender preferences.

$$\begin{aligned}
&f(t_1, \dots, t_s) = (x_1, \dots, x_s)\\
&x_i = \sum_{\forall r \in R} \omega_r * g(r, C_i)\\
&\text{where } g : R \times R \mapsto \{0, 1\},\\
&\text{and } g(a, b) = \begin{cases} 1 & \text{if a equals b} \\ 0 & \text{otherwise} \end{cases},
\end{aligned}$$

where the sub-recommender preferences are specified by the weight ω_r for each sub-recommender, respectively.

CommonVocabMetric

The following metric is based on calculating the number of common vocabularies within the recommendations for each keyword separately. Therefore, the `CommonVocabMetric` formula is defined as following:

$$f : T^s \mapsto \mathbb{R}^s,$$

where we assign each recommended term $t \in T^s$ a value from $\mathbb{R}$ based on whether the ontology appears in the recommendation lists of each keyword.

$$\begin{aligned}
&f(t_1, \ldots, t_s) = (x_1, \ldots, x_s)\\
&G_k = u_1, \ldots, u_j,\ j \in \mathbb{N},\ k \in K\\
&G_k : \text{recommendations of keyword k },\forall r \in R\\
&x_i = \omega_c \sum_{\forall k \in K} g(u_i, G_k),\\
&\text{where } g : U \times U^j \mapsto \mathbb{N},\\
&\text{and } g(a, B) = |a \cup B|.
\end{aligned}$$

Consider the following example: We have three keywords, and an ontology appears in one or multiple recommendations in the recommendation lists of two keywords, then the score assigned to each recommendation from this ontology would be 2. Adjusting the weight ω_c leads to the following results. The lower or higher this weight, the lower or higher are the preferences of maximizing or minimizing the diversity of ontologies inside the recommendations, respectively.

PreSufMetric

The following metric enhances the semantic meaning of the keywords. Therefore, the `PreSufMetric` formula is defined as following:

$$f : T^s \mapsto \mathbb{R}^s,$$

where we assign each recommended term $t \in T^s$ a value from $\mathbb{R}$ based on the keyword occurrence inside the label of the term.

$$
\begin{aligned}
&f(t_1, \dots, t_s) = (x_1, \dots, x_s),\\
&x_i = \omega_{ps} * g(l_i, k_i) + \omega_e * h(l_i, k_i) + \omega_b * p(l_i, k_i), i \in \mathbb{N}, k_i \in K,\\
&\text{where } k_i \text{ is the keyword that belongs to the term } t_i,\\
&\text{and } g, h, p : L \times K \mapsto \{0, 1\},\\
&\text{where } g(a, b) = \begin{cases} 1 & \text{if b is prefix or suffix of a} \\ 0 & \text{otherwise} \end{cases}\\
&\text{where } h(a, b) = \begin{cases} 1 & \text{if b equals a} \\ 0 & \text{otherwise} \end{cases}\\
&\text{where } p(a, b) = \begin{cases} 1 & \text{if b is a substring that is no prefix or suffix of a} \\ 0 & \text{otherwise} \end{cases}
\end{aligned}
$$

The weight ω_{ps} describes the preference of the keyword to occur either as a prefix or suffix of the label of the recommendation. Note that prefix and suffix here, need to have a space separation to the following part or the part beforehand inside the label. Consider this example: Let *player* be the keyword, the label ***football*** *player* would get a higher score than the word *ball**player***. This way we support a better semantic value improving results for the *rou**TeStop*** and ***train*** example in Section 1.1. The weights ω_e and ω_b describe the preference of the keyword to be equal or a substring (not prefix or suffix) of the label of the recommendation.

DescriptionMetric

The following metric prefers recommendations, which offer more value through metadata. Basically, preferring recommendations that have a label as well as comment enhancing the meaning of a node or relation. Therefore, the `DescriptionMetric` formula is defined as follows:

$$f : T^s \mapsto \mathbb{R}^s,$$

where we assign each recommended term $t \in T^s$ a value from $\mathbb{R}$ based on the presence of a label and a comment.

$$
\begin{aligned}
&f(t_1, \dots, t_s) = (x_1, \dots, x_s),\\
&x_i = \omega_d * g(l_i, c_i),\\
&\text{and } g : L \times C \mapsto \{0, 1\},\\
&\text{and } g(a, b) = \begin{cases} 0 & \text{if a or b is empty} \\ 1 & \text{otherwise} \end{cases}
\end{aligned}
$$

The weight ω_d describes the preference of the recommendation to have a label as well as a comment. This improves the description of the node or relation through metadata.

LOVMetric

Let $t' \in T$ be defined as an extension of the regular t defined before, where $t'_i = (t_i, o_i, e_i, s_i)$ and $o_i =$ the number of the *occurrences in datasets* parameter, $e_i =$ the number of the *reused by datasets* parameter, and $s_i =$ the score from LOV. Note that o_i, e_i, and s_i are metadata parameters from the LOV service. Then the `LOVMetric` formula is defined as follows:

$$f : T^s \mapsto \mathbb{R}^s$$

where we assign each recommended term $t \in T^s$ a value from $\mathbb{R}$ based on the metadata from the LOV service. This metric can be seen as a popularity metric, taking into account how *popular* an ontology is in regard to LOV.

$$\begin{aligned}
&f(t_1, \ldots, t_s) = (x_1, \ldots, x_s),\\
&x_i = \omega_s * g(s_i) + \omega_e * h(e_i) + \omega_o * p(o_i)\\
&\text{where } g, h, p : \mathbb{R} \mapsto \{0, 1\},\\
&\text{and } g(a) = \begin{cases} 1 & \text{if a} \geq \gamma_s \\ 0 & \text{otherwise} \end{cases}\\
&\text{and } h(a) = \begin{cases} 1 & \text{if a} \geq \gamma_\alpha \\ 0 & \text{otherwise} \end{cases}\\
&\text{and } p(a) = \begin{cases} 1 & \text{if a} \geq \gamma_\beta \\ 0 & \text{otherwise} \end{cases}
\end{aligned}$$

Note that this metric is only used for recommendations obtained from LOV. The parameters γ_s, γ_α γ_β specify the thresholds for the score from LOV, the *reused by datasets* parameter obtained from LOV, and the *occurrence in datasets* parameter obtained from LOV. Adjusting the weights ω_e and ω_o leads to the following results. The lower or higher these weights are, the lower or higher are the preferences of ontologies that are frequently reused. The weight ω_s describes the preference of the score obtained from LOV itself and thus the preference of the metrics defined by the LOV service.

Finalized Score Formula

We return the top n recommendations sorted based on the score. For the sorting, we need to merge the scores for each recommendation, from each metric. The finalized score for each recommended term $t \in T^s$ is applied by the formula:

$$\begin{aligned}
&M : \text{set of all metrics}\\
&\#j : |M|\\
&v_{tm} : \text{the score of recommendation } t \in T \text{ by metric } m \in M\\
&f : \mathbb{R}^{s^j} \mapsto \mathbb{R}^s,\\
&\text{then } f(\{v_{11}, \ldots, v_{1j}\}, \ldots, \{v_{s1}, \ldots, v_{sj}\}) = (x_1, \ldots, x_s),\\
&\text{where } x_i = \sum_{\forall m \in M} v_{im} * \omega_{im} \text{ and } \forall i \in T^s
\end{aligned}$$

All in all, the presented modular architecture offers the integration of new recommenders and metrics as well as the customization of the existing ones. As we already mentioned there are some reference points where the user can integrate several things to transform the default BR into their desired BR. In the following chapter, we explain these reference points as well as the architectural implementation details of the BR and how we integrate the BR into an existing ontology editor.

4 Realization

In this chapter, we describe the implementation of the recommender architecture from Chapter 3 in detail. For this, we explain interfaces, classes, and the internal structure. We also describe the instructions needed for the customization process regarding how to add or remove metrics, recommenders, or ontologies inside the recommender. In addition, we demonstrate how we integrate and use our recommender inside Neologism by showing the workflow of the BR in combination with the editor.

4.1 Architecture Implementation and Customization

In the following, we explain the architecture of Figure 3.1 in more detail. The following sections particularly tackle RQ2 and RQ4. The BR was implemented with the Java SDK version 1.8. We reused the recommender project of Neologism that was already existing (`https://github.com/Semantic-Society/Recommender`). The BR is built on top of the existing recommender. We reused as much as possible but adjusted and created additional classes based on the created concept.

4.1.1 Architecture and Modules Implementation

The following explanations of our data types and interfaces enhance the understanding of the following sections:

- The `Recommendation` and `Recommendations` classes are reused from the existing part of the recommender. The `Recommendations` class contains a list of `Recommendation` instances and the name of the creator as a `String`. The `Recommendation` class contains a list of comments, a list of labels, a URI, and the ontology name.
- The `BatchRecommendations` class is an extension of `Recommendations`, with the corresponding keyword (class or property) in `String` format.
- `LOVRecommendation` extends the `Recommendation` class to use the metadata provided from the LOV service. The metadata includes the parameters *occurrencesInDataset* and *reusedByDataset* as `int`, and *score* as `double`.
- The `RatedRecommendation` class extends the `Recommendation` class with a score specified as `double`.
- The `BatchQuery` class is the representation of the query in the recommender (see Listing 4.6). We have a list of class keywords, a list of property keywords, a domain as `String`, and an `int` for the recommendation limit.

- The `BatchRecommender` interface (see Figure 4.1) is the interface that needs to be implemented by all *sub-recommenders*, which are used for obtaining recommendations.

- The `Metric` abstract class (see Figure 4.2) is the class that needs to be extended by all *metrics*, which are used for calculating the score of the recommendations.

- The `Builder` class is used to create cleaned instances of recommendations when using the cleaning process to obtain only English recommendations. This class is reused from the existing part of the recommender.

In addition, the word *keyword* stands for the label specified inside the input ontology for any node or relation i.e., the labels for all nodes and relations of the input ontology are unified in *keywords*.

RESTController and QueryPreprocessor

In the following, we explain Figure 3.2 and its workflow by starting with the *SystemInterface*, which is replaced by the `RESTController`. It dictates the flow and forwards the `BatchQuery` to the `QueryPreprocessor`. The `QueryPreprocessor` executes the preprocessing step as described in Section 3.3.1. The original and transformed keyword are saved inside a map in the `QueryPreprocessor`. The resulting transformed `BatchQuery` is forwarded by the `RESTController` to the `RecommenderManager`. The collected recommendations for each keyword from the `RecommenderManager` are passed to the `RankingCalculator` in a `Map<String, List<Recommendations>>` format. The `String` describes the keyword, while the `List<Recommendations>` contains the collected recommendations from each sub-recommender related to that keyword. The final result is then received from the `RankingCalculator` as a `List<BatchRecommendations>`. The keywords of the result are then transformed back to the original keywords using the map of the `QueryPreprocessor` described above. The final result is then returned by the `RESTController` as a response.

RankingCalculator

The `RESTController` uses the `RankingCalulator` to collect the final recommendation lists for all keywords. The `ScoreManager` is used to add the `MetricScore` calculated by each metric for the `Map<String, List<Recommendations>>`, that was created by the `RecommenderManager`. The defined metrics, which are used for calculation, are obtained from the `MetricManager`. Instances of the `RatedRecommendation` class are used to specify the final score that was calculated by the metrics for each keyword. The formula used for the final score calculation is explained in Section 3.4. We return a `List<BatchRecommendations>` that contains the amount of the top n `RatedRecommendations` for each keyword, which is the response returned from the `RESTController`.

ScoreManager

Inside the `ScoreManager` the `keywordMetricScores`, as well as `keywordFinalScores` maps, are specified. While the first contains `MetricScore` instances, the second contains `Score` instances, both using a keyword in `String` format as the key. The difference is that the `MetricScore` class is bound by the `MetricId`, which indicates the `Metric` used to calculate the score for the recommendation. The `keywordFinalScores` map is created, when all metrics finished their calculations, by using the corresponding weights for each `Metric` defined inside the `MetricManager` for the final calculation (see Section 3.4). The `Score` instances inside the `keywordFinalScores` map contain the finalized score for a specific recommendation and are used by the `RankingCalculator` to create `RatedRecommendation` instances for each recommendation.

RecommenderManager and MetricManager

The `RecommenderManager` is responsible for several tasks, which are explained in the following:

- Managing the sub-recommenders that are used to collect the recommendations based on the input model.
- Managing the domain that is specified inside the input model as a `String`.
- Receiving the recommendations of all sub-recommenders based on the nodes and relations from the input model.
- Unifying the `Recommendations` from all different sub-recommenders inside a `Map<String, List<Recommendations>>`.

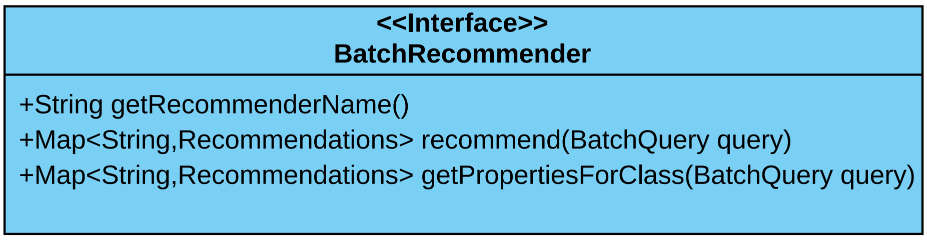
```java
private RecommenderManager() {
    recommenders.add(new LovBatchRecommender());
    recommenders.add(LocalVocabLoader.PredefinedVocab.
        DUBLIN_CORE_TERMS);
    recommenders.add(LocalVocabLoader.PredefinedVocab.DCAT);
}
```

Listing 4.1: Excerpt of the `RecommenderManager` class with three sub-recommender specifications. Two LocalRecommenders, using the DCTerms and DCAT ontologies, and the LOVRecommender.

Inside the `MetricManager` the used metrics and their corresponding weights are specified (see Listing 4.5). The current metrics implemented are the `DomainMetric`, the `CreatorMetric`, the `LOVMetric`, the `PreSufMetric`, the `DescriptionMetric`, and the `CommonVocabMetric`.

4.1.2 Customization of Sub-Recommenders, Ontologies, and Metrics

The user can add and remove sub-recommenders, local ontologies, and metrics. In addition, the user can easily adjust the weight of a metric inside the `MetricManager`. In the following, we explain the steps on how to perform the customization with regard to the previously mentioned aspects. Currently, the `LOVRecommender` and several `LocalVocabLoader` instances are in use as sub-recommenders for the BR. The process of adding a new sub-recommender is similar to the process of adding a sub-recommender for a local ontology, but the latter requires more steps.



<<Interface>>
BatchRecommender

+String getRecommenderName()
+Map<String,Recommendations> recommend(BatchQuery query)
+Map<String,Recommendations> getPropertiesForClass(BatchQuery query)

Figure 4.1: The `BatchRecommender` interface. Any recommender that is supposed to be used to collect recommendations, must implement this interface.

Note that the `LocalVocabLoader` was already implemented, we reused this class and implemented the `BatchRecommender` interface (see Figure 4.1). With this implementation, instances of the LocalVocabLoader can be used as local sub-recommenders. The `cleanAllExceptEnglish()` (see Listing 4.2) method is reused for retrieving only recommendations in English. This method must be adjusted when adding a customized recommendation (like the LOVRecommendation for the LOVRecommender).

```java
public Recommendations cleanAllExceptEnglish() {
    ...
    if (original instanceof LOVRecommendation) {
        b.addLOVParams(((LOVRecommendation) original).
           getScore(), ...);
    }
    ...
```

Listing 4.2: The `cleanAllExceptEnglish()` method with adjustments based on the `LOVRecommendation`. The `addLOVParams` method is executed if the recommendation is a `LOVRecommendation`.

For adding a sub-recommender that is not a local ontology, but an external recommender (e.g., BioPortal) the following steps need to be executed.

1. Create a recommender class (e.g., BioPortalRecommender) that implements the `BatchRecommender` interface of Figure 4.1.

2. Optional: If the recommender provides specific metadata that you want to use, you need to create a subclass of the `Recommendation` class, which specifies the metadata (like the `LOVRecommendation` class).

   - For this, the three methods specified in Figure 4.1 must be added to the recommender. The first method returns the name of the recommender, while the second and the third focus on the retrieval of the class and property recommendations, respectively.

```
1  public static class Builder {
2          ...
3          private Double score;
4          private boolean isLOVRecommendation;
5
6      public Builder addLOVParams(double score, int
          occurrenceInDatasets, int reusedByDatasets) {
7          this.score = score;
8          ...
9          this.isLOVRecommendation = true;
10         return this;
11     }
12
13     public Recommendation build() {
14         if (isLOVRecommendation) {
15             return new LOVRecommendation(URI, ontology,
                  ImmutableList.copyOf(labels), ImmutableList.
                  copyOf(comments), score, ...);
16         }
17         ...
```

Listing 4.3: Builder adjustment based on the `LOVRecommendation`. Showing a subset of the parameters needed for the `LOVRecommendation`, its corresponding `addLOVParams` method, and the adjustment of the `build` method.

   - In addition, the `Builder` must be adjusted like in Listing 4.3.

   (1) Add the parameters which are needed for the metadata like e.g., the *score* for the `LOVRecommendation`.

   (2) Add the method, to set these parameters as in lines 6-11 in Listing 4.3. Note that you should add a `boolean` for your specific recommendation like the *isLOVRecommendation* needed for the final build method.

   (3) Adjust the build method returning your specific recommendation, if the previously specified `boolean` parameter like *isLOVRecommendation* is `true`.

   (4) Finally, inside the `cleanAllExceptEnglish()` method check the type of the recommendation for your specific recommendation type and run the method created in step 2 like in Listing 4.2.

3. An instance of the recommender needs to be added into the `RecommenderManager` like in Listing 4.1.

In the following we explain the procedure for adding a sub-recommender for a local ontology using the `LocalVocabloader` class:

1. Copy your ontology into the resource folder `src\main\resources`.
2. Create the specific `LocalVocabLoader` instance by using the load function like in Listing 4.4. Note that the name of the instance should be integrated into the `CreatorMetric` if this sub-recommender should have a specific preference. The parameters of the load function are described in the following:
   (1) describes the filename of the resource (e.g., *dcat.ttl*).
   (2) describes the language of the file (currently only TURTLE is available).
   (3) describes the ontology name (e.g., dcat).
   (4) describes the common prefix inside the turtle file (e.g., in the case of `dcat:keyword` the common prefix would be `dcat`).
3. The new `LocalVocabloader` instance needs to be added into the `RecommenderManager` as in Listing 4.1 using the name of the instance specified in step 2) before.

```
public static class PredefinedVocab {
     public static final LocalVocabLoader DCAT = load("dcat
        .ttl", Lang.TURTLE, "DCAT", "dcat");
     public static final LocalVocabLoader DUBLIN_CORE_TERMS
        = load("dcterms.ttl", Lang.TURTLE, "DCTERMS",
            "dcterms");
}
```

Listing 4.4: Excerpt of the `PredefinedVocab` class with local ontologies, that are loaded as `LocalVocabLoader` instances. The input parameters of the `load` method are the ontology file as *.ttl*, the language (currently only `Lang.TURTLE`), the ontology name, and the ontology prefix.

Adding, Adjusting, and Removing Metrics

A user can add, adjust, and remove metrics and their corresponding weights. Note that the use of specific data of specific recommendations e.g., `LOVRecommendation` requires a distinction between the different types of recommendations within the metric. In the following we explain the process of adding a new metric with its corresponding weights:

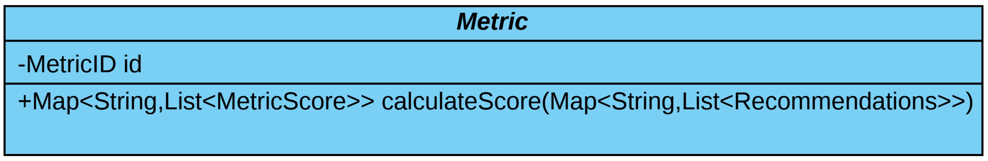

Metric
-MetricID id
+Map<String,List<MetricScore>> calculateScore(Map<String,List<Recommendations>>)

Figure 4.2: The abstract class `Metric`. Any metric that is to be used to calculate the score calculation must extend this class.

1. Create a metric that extends the abstract class `Metric`, which can be seen in Figure 4.2.
2. Specify a new `MetricId` for the created metric.
3. Add the created metric to the `MetricManager` using the specified `MetricId` as in Listing 4.5.
4. Add the weight for the created metric in the `MetricManager` using the specified `MetricId` as in Listing 4.5.

```java
private MetricManager() {
  metrics.add(new CreatorMetric(MetricId.CREATOR));
  metrics.add(new DomainMetric(MetricId.DOMAIN));
  metrics.add(new CommonVocabMetric(MetricId.COMMONVOCAB));

  metricWeights.put(MetricId.CREATOR, 1.0);
  metricWeights.put(MetricId.COMMONVOCAB, 1.0);
  metricWeights.put(MetricId.DOMAIN, 1.0);
}
```

Listing 4.5: Excerpt of the `MetricManager` class with metrics, which are used for calculating the scores for each recommendation, and their corresponding weights.

4.2 Integration into Neologism 2.0

Currently, Neologism offers the creation and publication of ontologies. In this section, we explain the integration of the BR into Neologism and how we extend the functionalities that support the user during the ontology creation process. More precisely, we explain how we integrate the context-free phase in Neologism and how we communicate with the BR to perform the context-development phase. With this we tackle RQ2. The context-free phase is directly integrated into Neologism and is part of the Neologism

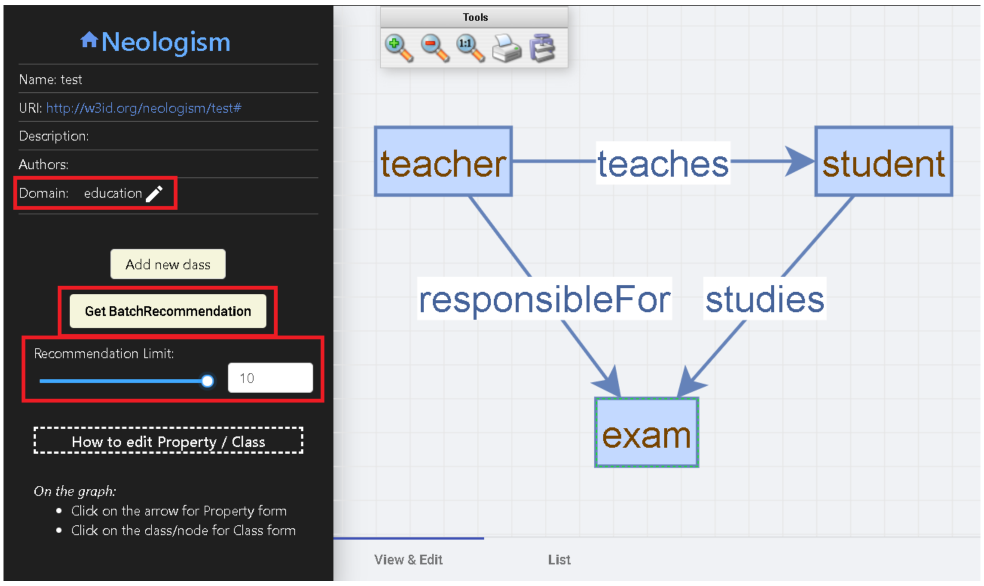

Name: test
URI: http://w3id.org/neologism/test#
Description:
Authors:
Domain: education
Recommendation Limit: 10

Figure 4.3: The ontology from Figure 1.2 specified correctly with the use of the Live Support from Figure 4.4. On the left are basic Neologism functionalities and highlighted in red are the integrated *Get BatchRecommendation* button, the regulator for the number of recommendations, and the optional domain specification.

architecture, which is implemented in Angular2 and Typescript. The context-free phase is performed by using the Live Support. Together with the Live Support, the optional domain specification is integrated directly into Neologism. To make use of the `DomainMetric` (see Section 3.4.2) the domain needs to be specified by the user. For the context-development phase, we explain the communication with the BR, where we expand the Java code of the existing recommender inside Neologism. Figure 4.3 shows an overview of Neologism with the improved example ontology from Figure 1.2.

4.2.1 Live Support and Optional Domain Specification

In this section, we approach RQ2, by explaining the Live Support and the optional domain specification. The input for the Live Support is the keyword entered by the user in `String` format while creating or editing a node or relation. The Live Support provides a list of suggestions for the user while typing. As mentioned, the Live Support is responsible for the *context-free* phase. With the Live Support, we focus on the prevention and correction of spelling errors. As the frontend of Neologism is implemented in Angular2, we use a compatible spellchecker. A tool that provides functionalities to give suggestions for an input keyword in Angular2 is the `ngx-spellchecker`. We integrated the `ngx-spellchecker` in Neologism in the following way. When the user creates or edits a node or relation, a dropdown-list of suggestions for the typed keyword is presented as in Figure 4.4. The Figure shows the suggestions for the wrongly specified *techer* keyword from Figure 1.2. In this case, selecting the second element of the suggestion list would lead to the desired meaning and therefore to a node with a correctly specified keyword. For the `ngx-spellchecker`, we used version 1.0.5 with the `normalized_en-US.dic` dictionary, as we focus on the English language.

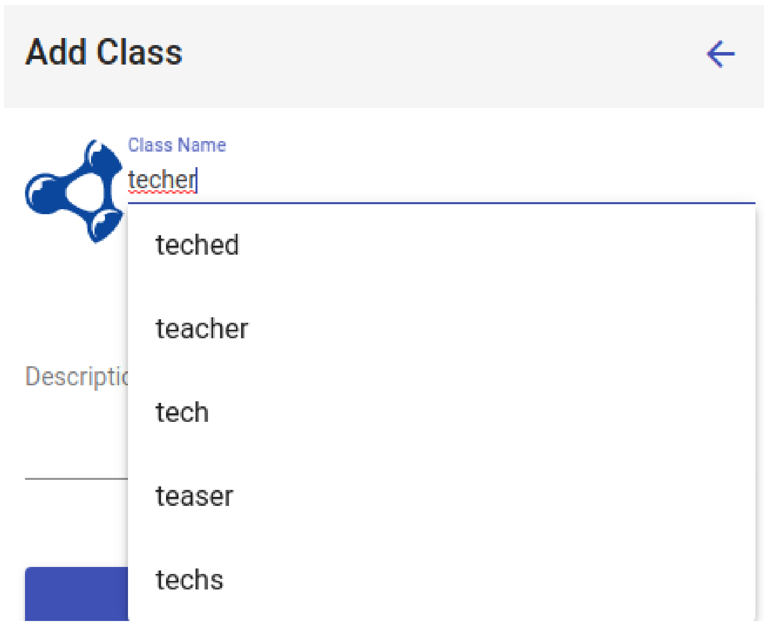


Figure 4.4: A list of suggestions using the `ngx-spellchecker` for the keyword *techer* displayed. The second suggestion shows the desired keyword.

As mentioned in Chapter 3, we offer the option for the user to specify a domain. The specified domain tag is used in `String` format to calculate the `DomainMetric` specified in Section 3.4.2. Another use-case for the domain tag can be the usage of specific recommenders for their corresponding domain. As mentioned in Chapter 2 BioPortal could be used to gather recommendations for the biological domain. With an extension of the `DomainMetric` recommendations from *domain-recommenders* can get a higher score and therefore, a higher preference than recommendations from other recommenders. The integration of domain-recommenders could be a customization step based on the use-case and user preferences. The domain can be specified by pressing the icon next to the *Domain:* field in Figure 4.5 and enter a domain as it can be seen on the right. After typing the new domain, the user must confirm their change by pressing the *Enter* key.

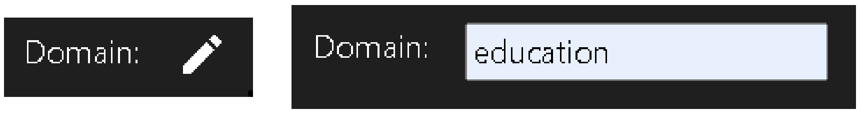


Figure 4.5: The optional domain specification. The left side shows the domain label with an edit button. The right side shows a sample input during editing.

4.2.2 Preprocessing and Recommendation Step

In contrast to the Live Support, the preprocessing and recommendation step are performed inside the BR architecture by certain modules. The BR implementation is set up as a REST-based service, such that Neologism or other programs can use this service. The service can be queried via the following URL: `http://localhost:8080/recommender/batchRecommender`, using the body structure that can be seen in Listing 4.6 defining the domain, a recommendation limit, classes, and properties of the input model.

```
{
 "domain": "education",
 "classes": ["teacher", "exam", "student"],
 "properties":["responsibleFor", "teaches","studies"],
 "limit":10
}
```

Listing 4.6: Request for the BatchRecommender based on the improved example of Figure 1.2 using the Live Support. The *domain* is specified as `String`, the *limit* is specified as an `Integer`, and *classes* and *properties* as `List<String>`.

The *context-development* phase starts when the user marks their ontology draft as final by clicking the *Get BatchRecommendations* button, which can be seen in Figure 4.3. A loading screen (see Figure A.1) is shown to confirm that the request is being processed. We use the created nodes and relations of the user as input. For this, we transform the nodes and relations into lists of `Strings`, containing the node and relation keywords. We create a request containing the lists and the domain (see Figure 4.5) as in Listing 4.6. The limit (see Listing 4.6) of recommendations that are received from the BR can be set by using the integrated regulator in Figure 4.6. The regulator functionality was integrated as a result of the evaluation in Chapter 5. After execution of a query against the BR the preprocessing step begins. In the preprocessing step, we mainly perform `String` matching operations, which are used to solve the problem classes specified in Section 3.3.1 with their corresponding solution. The recommendation step receives the output of the preprocessing step as input. For the recommendation step, we use the implemented sub-recommenders to receive all possible recommendations. We put a special focus on using the LOV API, as it provides the largest open-source dataset for ontologies available online. Therefore, we

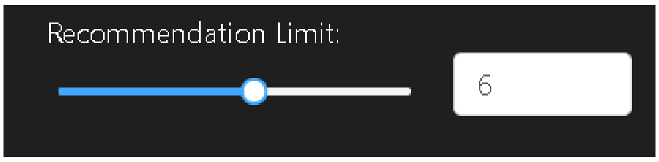
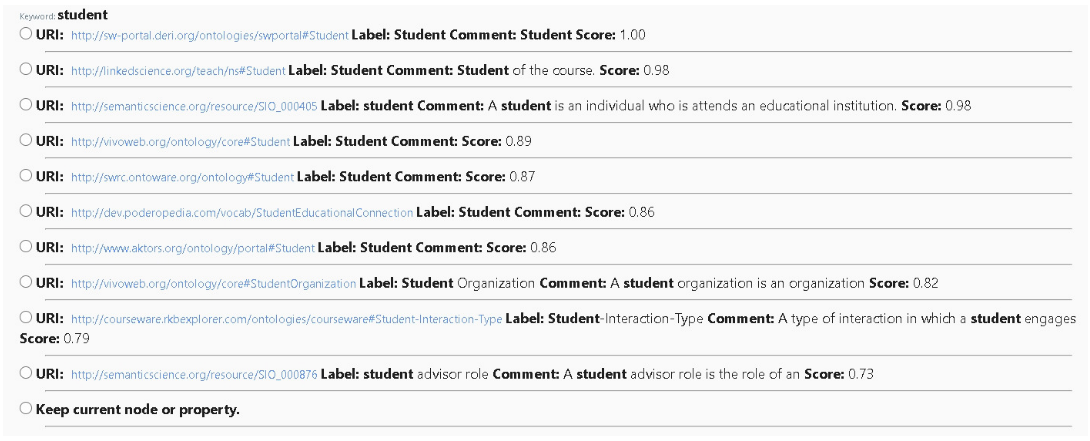


Figure 4.6: The regulator with which the recommendations limit can be specified.

mainly focus on explaining the LOV recommender for this step. Each `String` of the `classes` list and `properties` list of the body specified in the request (see Listing 4.6) is forwarded to the LOV API. The documentation of the LOV API can be found here [VAPV11]. The recommendations obtained from the LOV API are then used together with the recommendations obtained from the local recommenders. The local recommenders use each `String` of the lists to perform a search through the existing local ontologies.



Keyword: **student**

○ **URI:** http://sw-portal.deri.org/ontologies/swportal#Student **Label: Student Comment: Student Score:** 1.00

○ **URI:** http://linkedscience.org/teach/ns#Student **Label: Student Comment: Student** of the course. **Score:** 0.98

○ **URI:** http://semanticscience.org/resource/SIO_000405 **Label: student Comment:** A **student** is an individual who is attends an educational institution. **Score:** 0.98

○ **URI:** http://vivoweb.org/ontology/core#Student **Label: Student Comment: Score:** 0.89

○ **URI:** http://swrc.ontoware.org/ontology#Student **Label: Student Comment: Score:** 0.87

○ **URI:** http://dev.poderopedia.com/vocab/StudentEducationalConnection **Label: Student Comment: Score:** 0.86

○ **URI:** http://www.aktors.org/ontology/portal#Student **Label: Student Comment: Score:** 0.86

○ **URI:** http://vivoweb.org/ontology/core#StudentOrganization **Label: Student** Organization **Comment:** A **student** organization is an organization **Score:** 0.82

○ **URI:** http://courseware.rkbexplorer.com/ontologies/courseware#Student-Interaction-Type **Label: Student**-Interaction-Type **Comment:** A type of interaction in which a **student** engages **Score:** 0.79

○ **URI:** http://semanticscience.org/resource/SIO_000876 **Label: student** advisor role **Comment:** A **student** advisor role is the role of an **Score:** 0.73

○ **Keep current node or property.**

Figure 4.7: The recommendations list for the *student* keyword of Figure 4.3. Displaying the URI, the label, and comments (if available), and the score calculated by the BR.

The last part of the recommendation step is the calculation of the score and ranking of the recommendations for each keyword. For this, we use the metrics explained in Section 3.4. After developing a context between the results and thereby ranking the results, we return a list of the top recommendations (the number is based on the specified limit) for each node and relation of the input ontology. In the list, we focus on the top recommendations and display the URI, the label, the description, and the score calculated by the BR of each recommendation for each term. The key is that we do not show too much information, but cover the most important information that is provided from the ontology as shown in Figure 2.1. The specific way the visualization is integrated into Neologism can be seen in Figure 4.7. The Figure shows the recommendations for the *student* node. The label and the comment are retrieved using the `rdfs:label` and `rdfs:comment` properties. Generally, the first recommendation of each list is preselected, so that the user can directly lift the ontology, based on the recommendations with the highest score. If no recommendations were found for a keyword, the *Keep current node or property* option is preselected. This list of recommendations is displayed for each node and relation separately. The user can also choose to *Keep current node or property* if the desired meaning is not listed within the recommendations.

4.2.3 Ontology Lifting

After the two phases are completed, the final transformation must be performed, mainly lifting the created ontology based on the selected recommendations to the Semantic Web. As mentioned before, we preselect the first recommendation for the user. This is done because we assume that most users tend to select the recommendation with the highest score. With the pre-selection, a user can directly press the *Lift ontology* button, which appears on the sidebar of Neologism like the *Get BatchRecommendation* button in Figure 4.3. The *Get BatchRecommendation* button is disabled during the selection of the user. The selection is the process by which the user chooses the recommendation from the lists for each keyword that corresponds to the desired meaning. In addition, a *return* button is available, which returns the user to the editing graph view without applying the recommendations. When pressing the *Lift Ontology* button each node and relation is replaced with the selected classes and properties.

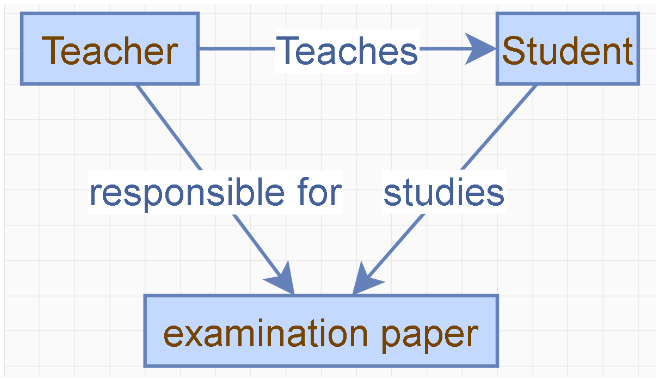
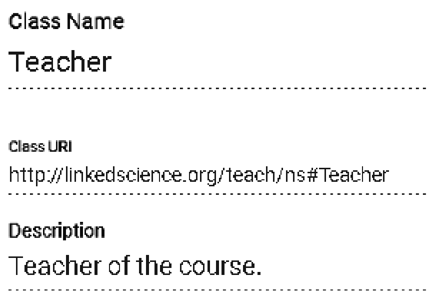


Figure 4.8: The ontology from Figure 1.2 after the lifting process. The originally specified terms are transformed based on the chosen recommendations.

An example of a lifted ontology is shown in Figure 4.8. All labels of the nodes and relations are replaced by the selected labels of the classes and properties. This example is the lifted and improved result of Figure 1.2. Note that if no label from the recommendation is specified, the original label specified by the user is used. In addition, the URI is replaced with the URI from the selected recommendation. The same applies to the description, using the comment from the selected recommendation if it is present.

An example of a node that was lifted to a class can be seen in Figure 4.9. It shows the lifted *teacher* keyword with its metadata of the chosen recommendation. The label, URI, and description are replaced by the corresponding selection of the user. The overall functionalities integrated in Neologism are summarized in the following.

- **Live Support**: This includes displaying suggestions of keywords within a dropdown-list, while the user creates or edits nodes or relations as described above.


Class Name
Teacher

Class URI
http://linkedscience.org/teach/ns#Teacher

Description
Teacher of the course.

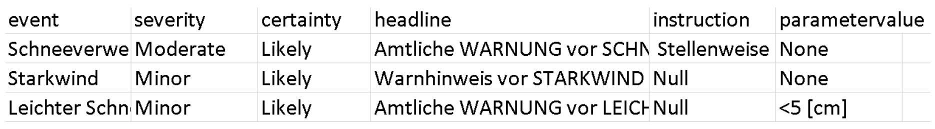
Figure 4.9: The *teacher* keyword and its corresponding metadata (Label, URI, and description) from the chosen recommendation.

- **Standalone Recommender**: The recommender is implemented with the connection to the LOV API and the possibility of integrating local ontologies or other sub-recommenders and using customizable metrics for ranking different recommendations.
- **Optional Domain Specification**: Optional field for the user, where they can specify a certain domain tag. This tag can be used for domain-specific sub-recommenders.
- **Recommendation Limit Regulator**: We have a regulator for the user, where they can specify the number of recommendations they want to receive.
- **Get BatchRecommendation Button**: Clicking this button triggers the communication with the BR.
- **Visualization**: The visualization is implemented in Neologism like Figure 2.1 and can be seen in Figure 4.7.
- **Lift Ontology Button**: Clicking this button applies the transformation of original terms in the created ontology to the chosen recommendations from the list (see Figure 4.7).

The contributions to the BR as well as its integration into Neologism are merged via pull-requests to the respective projects and are therefore now included in them. In the next chapter, we are going to evaluate the implemented version of the BR with its integration into the ontology editor Neologism. This evaluation is based on a user study.

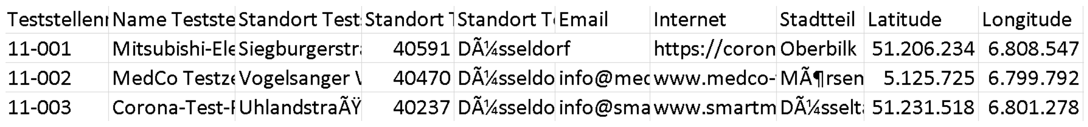
5 Evaluation

In this section, we evaluate the developed BR and its integration into Neologism. We choose a user-centered evaluation method. This is due to the difficult comparability to other BRs as there does not exist such a general recommender, which is focused on ontology creation, or a dataset for evaluation. We use the example of Figure 1.2 as input for the BR with which we determine suitable default weights of the metrics inside the BR for this evaluation. The chosen weights for each metric can be seen in Table A.1. It is generally difficult to determine *perfect* weights for all models created by users, as we already mentioned (see Chapter 2) the meaning of terms can be very subjective. Different people may prefer different definitions of terms considering the context of those terms.



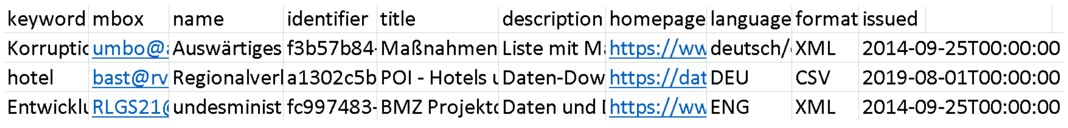
event	severity	certainty	headline	instruction	parametervalue
Schneeverwe	Moderate	Likely	Amtliche WARNUNG vor SCHN	Stellenweise	None
Starkwind	Minor	Likely	Warnhinweis vor STARKWIND	Null	None
Leichter Schn	Minor	Likely	Amtliche WARNUNG vor LEICH	Null	<5 [cm]

(a)



Teststellenn	Name Testste	Standort Test	Standort T	Standort T	Email	Internet	Stadtteil	Latitude	Longitude
11-001	Mitsubishi-Ele	Siegburgerstr	40591	DÃ¼sseldorf		https://coron	Oberbilk	51.206.234	6.808.547
11-002	MedCo Testze	Vogelsanger V	40470	DÃ¼sseldo	info@mec	www.medco-	MÃ¶rsen	5.125.725	6.799.792
11-003	Corona-Test-F	UhlandstraÃŸ	40237	DÃ¼sseldo	info@sma	www.smartm	DÃ¼sselt	51.231.518	6.801.278

(b)



keyword	mbox	name	identifier	title	description	homepage	language	format	issued
Korruptic	umbo@	Auswärtiges	f3b57b84	Maßnahmen	Liste mit M	https://ww	deutsch/	XML	2014-09-25T00:00:00
hotel	bast@rv	Regionalverl	a1302c5b	POI - Hotels	Daten-Dow	https://dat	DEU	CSV	2019-08-01T00:00:00
Entwicklu	RLGS21(	undesminist	fc997483-	BMZ Projekt	Daten und I	https://ww	ENG	XML	2014-09-25T00:00:00

(c)

Figure 5.1: Excerpts of the three datasets used for evaluation. The *storm*, the *corona test station*, and the *datasets* dataset can be seen in a), b), and c), respectively.

For the user-centered evaluation, each user is given a document (see Figure A.5) for introducing the tasks they have to perform, as many persons did not have any prior knowledge regarding ontologies or data modeling in general. The total of 15 participants was divided into 3 groups with 5 members each. Ten of the participants were students of different backgrounds (e.g., economy, sports, mechanical engineering) and the other five were researchers of the *Mobility Data Space* [1] project, working in the area of mobility. The tasks are based on three different datasets, where each dataset is processed by five persons. Archived copies of all datasets and the data evaluation can be found in the git repository [2].

The three datasets cover the topics *storm*, *datasets description*, and (*corona*) *test stations*. Excerpts of the datasets can be seen in Figure 5.1. The corona test station dataset is obtained from `https://opendata.duesseldorf.de`. Note that there were encoding errors (e.g., with the word Düsseldorf), caused by umlauts in the German language, which we intentionally did not remove in a preprocessing step. The participants received the dataset as depicted in Figure 5.1. The storm dataset is created by using the datasets from the `https://maps.dwd.de/geoserver/web/`, which makes German weather data publicly available. The datasets description dataset is a combination of descriptions of different datasets of the `https://www.govdata.de/` website. We choose these three domains as they can be understood by any user and are covering a broad range of terminology to test the BR. The storm dataset is processed by the participants with the mobility background, while the other two datasets are processed by the students. We make this decision because the storm dataset is more related to mobility than the others. This allows us to compare how domain experts behave compared to novices.

For measuring the usability of the implementation, we use the System Usability Scale (SUS) questionnaire [Brok13] that can be seen in Figure A.6. It is a measurement commonly used to compare different systems based on usability. The individual scenario complexity for each of the users is measured using the After-Scenario questionnaire (ASQ) [Lewi91] that can be seen in Figure A.7. Both of these questionnaires are short and simple and do not go too much into detail but cover the surface of what we aim to interpret: The complexity of the scenario and the overall system usability of the BR. In addition, the questionnaires provide comparability to other scenarios and systems.

Our goal for the user study is to evaluate our BR in a proper way. We do not aim to present the user with a pre-selected model that already provides optimal recommendations through previous tests, but rather perform an overall evaluation of the recommender. Therefore, we decided to select different domains in which different users create different nodes and relations. This gives us a variety of different recommendations that we cannot predict and optimize, and therefore we can evaluate our recommender in a more general way.

As creating a gold standard for ontologies is very difficult we need another verification system. Therefore, we guide the user during the process of the tasks in a controlled environment where we verify the user's desired meaning for a node or relation and their thoughts during creation. Especially, if the user cannot find proper relations, we support the user in finding combinations of nodes, which are likely to have a more intuitive relation. With this guidance, we ensure alignment of the user's task

[1] `https://www.mobility-data-space.de/`

[2] `https://i5.pages.rwth-aachen.de/master-thesis-enis-zejnilovic-data`

understanding with its design intent and if the chosen terms make sense. We assume that the user makes the *best* possible decision. The evaluation process is as follows:

1. For a given dataset without the header column, find a term for each of the columns based on the data inside of these.
2. Create classes inside Neologism for each of the columns and a class tied to the specific dataset. The latter is used as a base for the participant to simplify finding relations.
3. Create relations between the created nodes. These relations are invented by the user.
4. Press the *Get BatchRecommendation* button and choose from the recommended terms the best fitting ones for each of the nodes and relations designed. If none is fitting keep your created term.
5. The ontology is lifted by clicking on the *Lift Ontology* button based on the chosen recommendations.
6. Fill the SUS and ASQ forms to describe your experience with the tool and the scenario. For the tool, focus on the Live Support and the Batch Recommendation. Qualitative feedback is integrated inside the questionnaires.

A large-scale evaluation can be considered for future work. In the current evaluation, we do not know the users' decisions if the top 10 ranks of LOV were provided. This would lead to better comparability but is not possible in practice. The issue is that we cannot perform multiple evaluations with the same person, because e.g., the order of the evaluation with or without metrics likely the results. In addition, an evaluation is a very time-consuming task where most users needed about 60 to 90 minutes. Maintaining focus even for an evaluation is a difficult task to accomplish.

5.1 Statistical Evaluation

In this section, we describe the difference between the LOV ranking and the metric-based ranking of the BR. We point out the differences, based on the top 10 rankings of LOV compared to the BR. In addition, we show the variation of LOV rankings of the recommendations in the BR chosen by the users.

During the evaluation, we obtain the chosen recommendations from each user. More precisely, we save the chosen BR rank and check the corresponding rank in LOV without our metrics. In total, we gathered 145 classes and 53 properties. Note that the number of properties is much lower than the number of classes, which is due to the difficulty of creating relations compared to nodes. The evaluation is done by comparing the results of the BR with the ranking obtained from LOV. This allows us to distinguish whether the standard ranking provided by LOV is more preferable to using our customizable metrics.

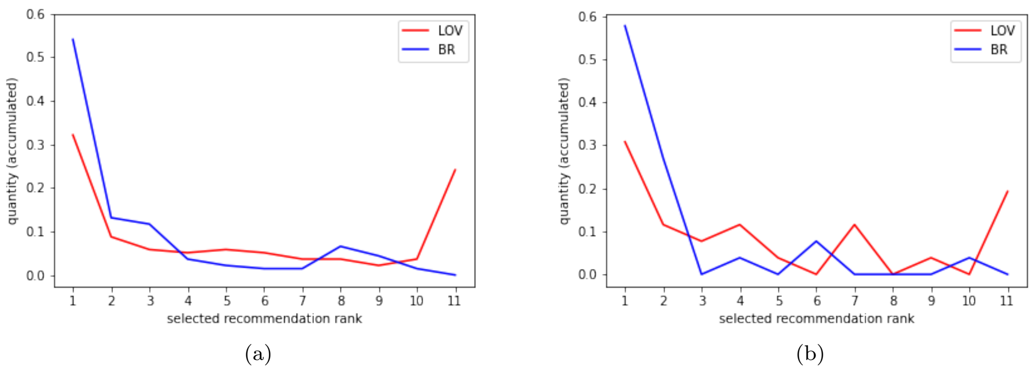


Figure 5.2: The distribution in % of class recommendation ranks (a) and property recommendation ranks (b) in the BR and LOV in. Rank 11 indicates all ranks >10.

Figure 5.2 shows the percentage of each rank of the recommendations chosen by the users. The total number of chosen classes is 137 and properties is 26, while in 8 and 27 cases the users kept their created node and relation, respectively. This corresponds to 5.6 % for the nodes and 51 % for the relations. We can see that rank 1 was chosen above 50 % of the time, which is the most by far, for the classes as well as properties. It is remarkable that the BR rank distribution is mainly at the top 3 for the classes and the top 2 for the properties, while there are still some outliers of about 10 % around rank 8 and 9 for classes. This implies that users do not tend to only choose between the upper ranks, but also take lower ranks into consideration. For the LOV rank, it is noticeable that rank 1 and ranks >10 almost have the same distribution with about 30 % for classes and properties. This shows a tendency for the LOV ranking to provide solid results with opportunities for improvement through our customizable metrics, as users have also selected results that have a much lower rank in LOV.

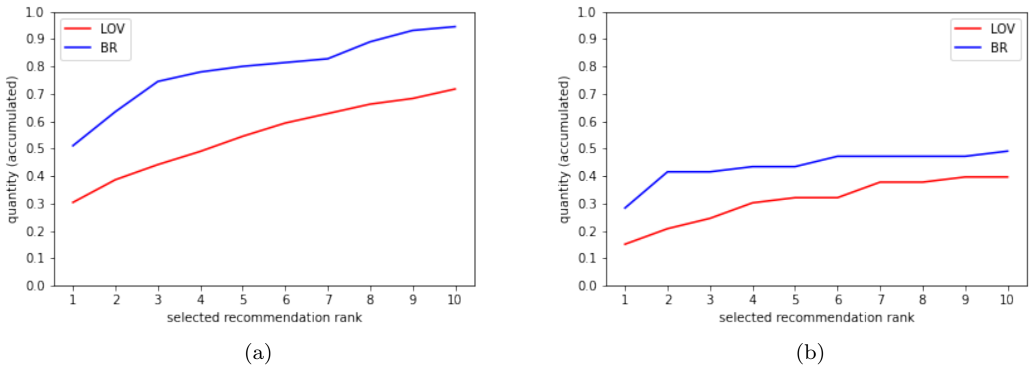


Figure 5.3: The cumulative distribution of chosen class (a) and property (b) recommendations. The difference of the curves indicates the number of recommendations, that can not be selected using the top 10 of the LOV rankings.

The improvement of the recommendation results by the BR can be seen in Figure 5.3. When we consider the ranks of the chosen recommendations for classes and properties focusing on the top 10 ranks, we can see the difference of about 20 % to 25 %. This difference indicates the ranks, which are above 10 assuming that the user would not have the possibility to choose another recommendation. In 1.38 % of the cases, which are 2 in total, there were no recommendations available because of the preprocessing of the recommender. The reason and solution for this will be explained later on. For properties, this difference is about 10 %, but as we can see in more than 50 % of the cases no recommendation was chosen.

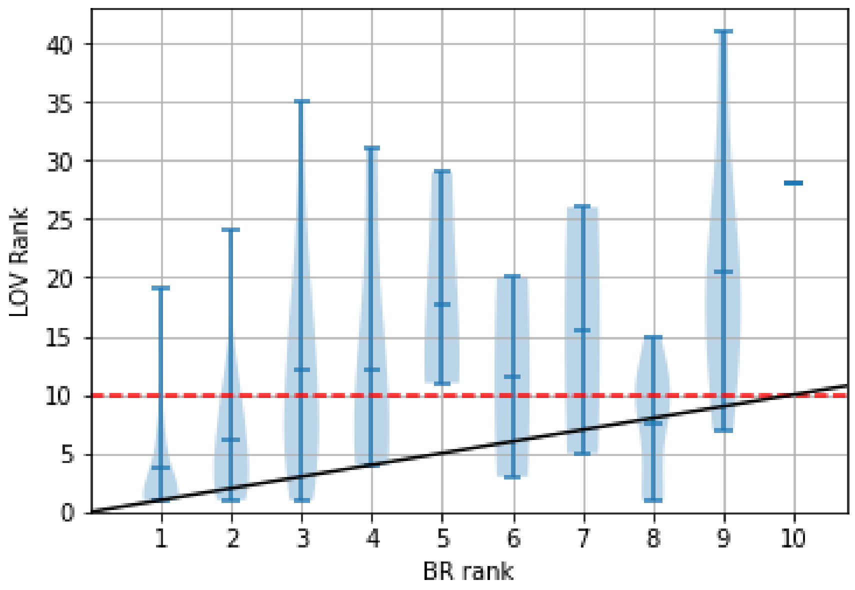


Figure 5.4: The distribution for all selected class recommendations of the BR ranks (top 10) based on their corresponding LOV ranks (top 50). The dashed red line indicates ranks above 10. The blue horizontal marks display the average LOV rank of each BR rank. The black line indicates the equivalence line of the LOV ranks and the BR ranks. All LOV ranks >10 would not have been visible to the user without the BR re-rankings.

Figure 5.4 shows the variation of the LOV ranks in regard to their corresponding BR rank only for classes. The (blue) average marker is above the equivalence line (black line) for every rank except for rank 8. It indicates that the BR re-ranking outperforms the default LOV rankings. We can see that when using the BR, users tend to choose a wide range of LOV ranks above rank 10. Although rank 1 occupies the most chosen class recommendations with more than 50 % it is striking that the BR managed to provide a variety of recommendations that are not in the top 10. The red dashed line indicates the threshold of the top 10. The ranks above this threshold cover about 30 % of the total chosen class recommendations (see Figure 5.2 rank 11). It is noticeable that the ranks above 10 are mostly below 35. Only for rank 9 in the BR is the highest LOV rank above 40 and still chosen by a user. Thus, we managed to cover a broad range of LOV ranks above rank 10 that were chosen by users. The correlation of the BR rank to the LOV rank considering the classes is 0.52 and 0.17 for the properties. In addition, the median of the LOV class ranks is 4 and the average class rank of LOV is 7.31, while the median of the BR class ranks is 1 and the average class rank of the BR is 2.66. These statistics show a tendency for the classes that the LOV ranking can be improved with our metrics. The plot for properties can be seen in Figure A.3. We distinguish between classes and properties, as the number of properties is low and in more than 50 % of the cases (see Figure 5.3) the users did not choose any recommendation.

5.2 User Questionnaires

In this section, we summarize the results of the SUS and the ASQ questionnaires. The SUS score provides comparability of the BR with other systems. The ASQ questionnaire describes the tendency of the different users in regard to the complexity of the given scenario.

Quantitative Evaluation

In the following, we explain the quantitative results of the SUS and ASQ questionnaires.

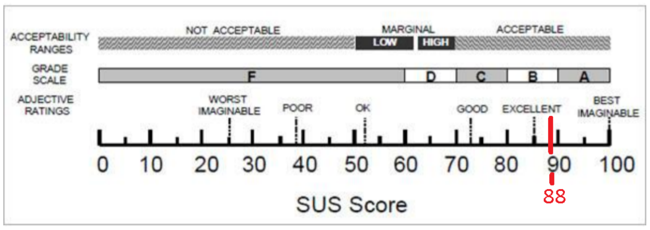


Figure 5.5: Grade rankings of SUS scores [Brok13]. The red mark indicates the achieved score (88) of the BR.

The SUS score is calculated in the following way. The value chosen for odd questions is subtracted from 5, while the value for even questions is subtracted by 1. We have calculated the average SUS score from all participants, which is 88. Figure 5.5 shows the grade ranking of SUS scores, which is based on „collected data on the use of SUS over more than a decade with a variety of different systems and technologies and have a pool of more than 3,500 SUS results“ [Brok13]. Based on this chart, we can grade the BR based on the calculated SUS score. The acceptability range is *acceptable*, which is the minimum requirement for the BR. The grading scale would be B while its adjective rating is *excellent*. With this grading, the BR can be compared to other systems. The results are satisfying and show the tendency that the BR is already in an *excellent* state, with possibilities of improvement.

For the ASQ questionnaire, we interpret score values from 1 to 3 as rather simple, 4 as neutral, and 5 to 7 as rather difficult as there are no standards to interpret the ASQ results rather than putting the score into its context. Figure 5.6 shows the answer distribution of the ASQ questionnaire. Note that the answers for Q3 are focused on the value 1 mostly, which shows that the users tend to perceive the material provided (see Figure A.5) and the support during the tasks as very helpful. This is also reflected in the average score, which is 1.3. We can see that the distribution of Q1 is distributed more than Q2 and especially more than Q3. This shows that there are users, for which the tasks were much more difficult than for others. For most of the users, the tasks were rather simple. This tendency can also be noted in the average score, which is 3.1. In addition, the distribution of Q2 shows that most users tend to be satisfied with the amount of time taken for the tasks but there are still users, which tend to perceive the time needed as longer than expected. The distribution is similar to the distribution of Q1, but with a slightly higher focus on the values from 1 to 3. The average score is 2.6, which shows the tendency of satisfaction with the time taken for the tasks.

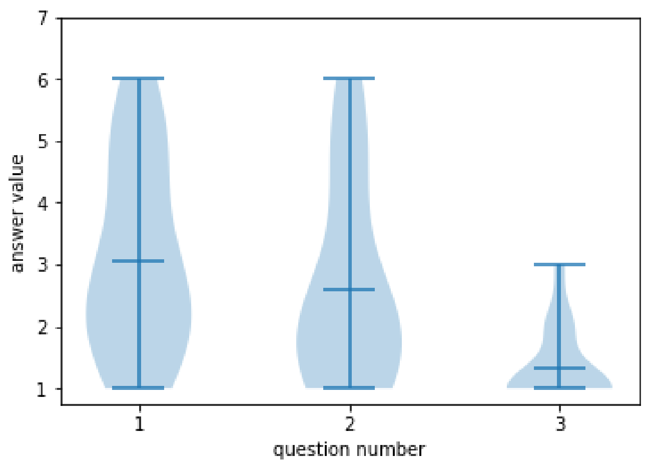


Figure 5.6: Answer distributions of the ASQ questionnaire (see Figure A.7). The mark displays the average answer value for each question. It shows that for Q1 and Q2 mostly 1 to 3 were chosen, while the distribution of Q3 shows that the max value is 3 and most users chose 1 here.

Generally, the users tend to see the system as beneficial, which can be derived from the SUS grading result. The given scenario is interpreted as rather simple with a tendency to neutral, based on our interpretations of the ASQ scores.

Qualitative Evaluation

In the following, we explain the qualitative results of the SUS and the ASQ questionnaires as well as the observations we made during the evaluation process. During the evaluation, we observed that the Live Support is helpful in many cases, as it prevents spelling errors and guides users to consistent use of spellings for the same word. An example is here the word *email*, where most users had their own writing style. The users, who commonly used the suggestions kept the same style. There were also situations where the Live Support did not show any suggestions. Especially for words that are more domain-specific and not commonly used, like *weather type* and *weather warning.* This is something that could be improved by adding domain-specific dictionaries or word sets.

Most users said that in some cases they had chosen other recommendations with a wrong meaning or done wrong node specifications if the support would not have been available. Some users prefer fewer recommendations, which is why the regulator was added after retrieving the evaluation results (see Chapter 4). In addition, one user mentioned the missing functionality of recommendations from domain ontologies, which is a functionality that is considered for future work e.g., integrating the BioPortal recommender for the biological domain (see Chapter 2).

Based on the quantitative feedback of the ASQ and the SUS questionnaires we found out that the tool is very helpful in improving the result of a draft ontology. The user does not have to do any research but can directly choose from a set of terms of already existing ontologies. Still, some issues occurred during the user study. Some issues were mentioned either in the qualitative feedback of the questionnaires or were given as oral feedback:

- The preprocessing is not optimized as in 1.37 % of the cases we did not receive a recommendation for classes because of the preprocessing. The camel-case solution has to be adjusted, as terms like *URL* or *ID* got separated into *U RL* and *I D*, which lead to unusable recommendations. This issue also shows how important the Live Support is, as wrongly spelled terms lead to wrong recommendations. We fixed this by only applying camel-case preprocessing to words, where directly after an upper-case letter at least 2 lower-case letters occur.

- Based on the user feedback and the observations we made, we can say that the language barrier is a major problem. All the persons are not native English speakers and therefore it was difficult to distinguish the correct word in English, even for a fluent speaker. This issue could be overcome by including synonyms and a metric, which assigns synonyms a higher score than words with a different meaning.

- The recommendations for the properties were in 51 % of the cases not accurate as only in 49 % of the cases a recommendation was chosen by users. The reason could be that the property specification varies much more than the specification of classes. It is difficult to use best practices for creating relations and selecting properties, especially for users who are not familiar with ontologies. In addition, the properties already available may be specified in another way than intended by the user [LGC*21]. In addition, it was difficult for the users to find relations between nodes in general, which is one reason why we have only about a third of the relations (53) compared to the nodes (145). These issues could be solved by a brief tutorial or introduction for users on how to use best practices to create properties between classes.

- The results of the LOV API sometimes do not provide the full description of a class or property. In some cases, the comment is cut off and therefore can not be interpreted correctly. By clicking on the URI link it is still possible to find the full description. This is something we contacted the LOV authors for to make improvements, such that the recommendations can be interpreted better and gain higher quality.

All in all, the evaluation shows, that the metrics tend to improve the ranking of the recommendations compared to the standard ranking from LOV. The BR improved the quality of the draft ontology by adding a description, a label, and the URI of an already existing class or property. We explained the main issues which occurred during the evaluation and possible ways how these issues could be solved. In the next section, we summarize the answers to the research question, the impact of this thesis, and give an outlook for future work.

6 Conclusion and Future Work

In this chapter, we summarize the results of the thesis and give an outlook for possible improvements and future work. We designed and implemented a flexibly extensible batch recommender for fast ontology prototyping, and extensively evaluated it and its integration into Neologism. Throughout the thesis, we answered all the research questions specified in Section 1.2. In Chapter 2 we described state-of-the-art concepts in various fields related to this thesis's topic. We pointed out the focus of this work, namely the development of a customizable recommender framework that enables the easy integration of different recommenders and metrics for computing the ranking of recommendations, focusing on the ontology creation process. We have shown that there is currently no such framework available. We summarize the answers to these research questions in the following. Note that the number of the answer is related to the number of the research question (RQ) from Section 1.2:

A 1: We developed a concept in Chapter 3, which is focused on a two-phase integration process of the BR into the ontology creation process. With the Live Support we aim at preventing spelling errors and therefore retrieving better recommendations with a more accurate semantic meaning. The preprocessing step focuses on simply using common design patterns, while the recommendation step mainly performs the calculation of the ranking and makes recommendations based on a set of simple metrics.

A 2: We managed to provide an infrastructure for a BR, which allows customization in various fields. This was outlined in Chapter 3 and Chapter 4. With this infrastructure, we allow adding and removing sub-recommenders. In addition, the BR provides the functionality of adding and removing metrics as well as adjusting their corresponding weights. All these specifications influence the BR and its ranking.

A 3: We created a default set of simple metrics, which are used to provide a useful ranking of available recommendations. Interpreting recommendations as *good* is very subjective, which is why we focused on rather simple metrics and more customizability rather than a higher complexity in the ranking calculation. Even with simple metrics, we managed to show that the integrated metrics offer better results than the default recommendation ranking from LOV (see Chapter 5). The different metrics are explained in Section 3.4, while its customizability in the realized BR is described in Section 4.1.2.

A 4: The easiest way was to use an already existing ontology editor (Neologism) and integrate the BR. We integrated the communication between the editor and the BR as well as a domain specification and a regulator to specify the desired number of recommendations. In addition, we provided the possibility to choose a recommendation for each node and relation of the designed ontology using a simple visualization technique for the recommendation results. Finally, we incorporated the functionality of lifting the ontology to the Semantic Web by using the chosen class and property recommendations and replacing the label, URI, and description of the corresponding node and relation, respectively. This was outlined in Section 4.2.

The results of this thesis impact the prototyping process of ontologies. We provide a two-phase approach of a BR, with customizable metrics and sub-recommenders that influence the ranking of recommendations. This offers the opportunity to customize the BR based on specific use-cases and domains and achieve optimal prototyping results. Starting from scratch is not needed as the information from existing ontologies is provided on demand. In addition, it improves the quality of the created ontology.

Four areas may be considered for future work. First, other visualization techniques that could improve the usability and other aspects of the recommendation visualization may be evaluated. Second, the ranking method can be expanded using e.g., ontology matching as explained in Section 2.4. Third, other domain-specific sub-recommenders like e.g., BioPortal could be integrated and bound to the domain using the optional domain specification of the BR. Finally, the description (label and comments) can be extended by using other common vocabulary terms (e.g., `skos:prefLabel`).

Bibliography

[ANS*07] Harith Alani, Natasha F Noy, Nigam Shah, Nigel Shadbolt, and Mark A Musen. Searching ontologies based on content: experiments in the biomedical domain. In *Proceedings of the 4th international conference on Knowledge capture*, pages 55–62, 2007.

[ASPT10] Neda Alipanah, Piyush Srivastava, Pallabi Parveen, and Bhavani Thuraisingham. Ranking ontologies using verified entities to facilitate federated queries. In *2010 IEEE/WIC/ACM International Conference on Web Intelligence and Intelligent Agent Technology*, volume 1, pages 332–337. IEEE, 2010.

[BBEI16] Judson Bandeira, Ig Ibert Bittencourt, Patricia Espinheira, and Seiji Isotani. Foca: A methodology for ontology evaluation. *arXiv preprint arXiv:1612.03353*, 2016.

[BuEi08] Paul Buitelaar and Thomas Eigner. Ontology search with the ontoselect ontology library. In *LREC*. Citeseer, 2008.

[BHLa01] Tim Berners-Lee, James Hendler, and Ora Lassila. The semantic web. *Scientific american*, 284(5):34–43, 2001.

[Brok13] John Brooke. Sus: a retrospective. *Journal of usability studies*, 8(2):29–40, 2013.

[CrCu05] Matteo Cristani and Roberta Cuel. A survey on ontology creation methodologies. *International Journal on Semantic Web and Information Systems (IJSWIS)*, 1(2):49–69, 2005.

[Degb17] Auriol Degbelo. A snapshot of ontology evaluation criteria and strategies. In *Proceedings of the 13th International Conference on Semantic Systems*, pages 1–8, 2017.

[DFJ*04] Li Ding, Tim Finin, Anupam Joshi, Rong Pan, R Scott Cost, Yun Peng, Pavan Reddivari, Vishal Doshi, and Joel Sachs. Swoogle: a search and metadata engine for the semantic web. In *Proceedings of the thirteenth ACM international conference on Information and knowledge management*, pages 652–659, 2004.

[DLSP18] Marek Dudáš, Steffen Lohmann, Vojtěch Svátek, and Dmitry Pavlov. Ontology visualization methods and tools: a survey of the state of the art. *The Knowledge Engineering Review*, 33, 2018.

[EMSS00] Michael Erdmann, Alexander Maedche, Hans-Peter Schnurr, and Steffen Staab. From manual to semi-automatic semantic annotation: About ontology-based text annotation tools. In *Proceedings of the COLING-2000 Workshop on Semantic Annotation and Intelligent Content*, pages 79–85, 2000.

[EuSh07] Jérôme Euzenat, Pavel Shvaiko, et al. *Ontology matching*, volume 18. Springer, 2007.

[GLCh05] Mingxia Gao, Chunnian Liu, and Furong Chen. An ontology search engine based on semantic analysis. In *Third International Conference on Information Technology and Applications (ICITA'05)*, volume 1, pages 256–259. IEEE, 2005.

[GFCo06] Asuncion Gomez-Perez, Mariano Fernández-López, and Oscar Corcho. *Ontological engineering: with examples from the areas of knowledge management, e-Commerce and the semantic web.* Springer Science & Business Media, 2006.

[Glei20] Lars C. Gleim and Schimassek. Schematree: Maximum-likelihood property recommendation for wikidata. In Andreas Harth, Sabrina Kirrane, Axel-Cyrille Ngonga Ngomo, Heiko Paulheim, Anisa Rula, Anna Lisa Gentile, Peter Haase, and Michael Cochez, editors, *The Semantic Web*, pages 179–195, Cham, 2020. Springer International Publishing.

[GiWi09] Maiga Gilbert and Ddembe Williams. A flexible biomedical ontology selection tool. *Strengthening the Role of ICT in Development*, 01 2009.

[HLS*08] Peter Haase, Holger Lewen, Rudi Studer, Duc Thanh Tran, Michael Erdmann, Mathieu d'Aquin, and Enrico Motta. The neon ontology engineering toolkit. *WWW*, 2008.

[Hopf20] Frank Hopfgartner. A gold standard dataset for large knowledge graphs matching. 2020.

[HPPa20] Sven Hertling, Jan Portisch, and Heiko Paulheim. Supervised ontology and instance matching with melt. *arXiv preprint arXiv:2009.11102*, 2020.

[IvPo20] Tatyana Ivanova and Miroslav Popov. Ontology evaluation and multilingualism. In *Proceedings of the 21st International Conference on Computer Systems and Technologies' 20*, pages 215–222, 2020.

[KHL*07] Akrivi Katifori, Constantin Halatsis, George Lepouras, Costas Vassilakis, and Eugenia Giannopoulou. Ontology visualization methods—a survey. *ACM Computing Surveys (CSUR)*, 39(4):10–es, 2007.

[KPS*06] Aditya Kalyanpur, Bijan Parsia, Evren Sirin, Bernardo Cuenca Grau, and James Hendler. Swoop: A web ontology editing browser. *Journal of Web Semantics*, 4(2):144–153, 2006.

[KTV*08] Akrivi Katifori, Elena Torou, Costas Vassilakis, Georgios Lepouras, and Constantin Halatsis. Selected results of a comparative study of four ontology visualization methods for information retrieval tasks. In *2008 Second International Conference on Research Challenges in Information Science*, pages 133–140. IEEE, 2008.

[KVKL20] Niklas Kolbe, Pierre-Yves Vandenbussche, Sylvain Kubler, and Yves Le Traon. Lovbench: Ontology ranking benchmark. In *Proceedings of The Web Conference 2020*, pages 1750–1760, 2020.

[Lant16] Birger Lantow. Ontometrics: Putting metrics into use for ontology evaluation. In *KEOD*, pages 186–191, 2016.

[LiDi05] Yun Lin and Hao Ding. Ontology-based semantic annotation for semantic interoperability of process models. In *International Conference on Computational Intelligence for Modelling, Control and Automation and International Conference on Intelligent Agents, Web Technologies and Internet Commerce (CIMCA-IAWTIC'06)*, volume 1, pages 162–167. IEEE, 2005.

[Lewi91] James R Lewis. Psychometric evaluation of an after-scenario questionnaire for computer usability studies: the asq. *ACM Sigchi Bulletin*, 23(1):78–81, 1991.

[LGC*20] J Lipp, L Gleim, M Cochez, I Dimitriadis, H Ali, C Lange, and S Decker. Quick ontology prototyping with neologism 2.0. `https://github.com/Semantic-Society/Neologism/`, 2020. [Accessed 02.12.2020].

[LGC*21] Johannes Lipp, Lars Gleim, Michael Cochez, Iraklis Dimitriadis, Hussain Ali, Daniel Hoppe Alvarez, Christoph Lange, and Stefan Decker. Towards easy vocabulary drafts with neologism 2.0. In *ESWC2021 Poster and Demo Track*, 2021. [to-be-published].

[LPAl11] Andreas Lommatzsch, Till Plumbaum, and Sahin Albayrak. An architecture for smart semantic recommender applications. In *11th International Conference on Innovative Internet Community Systems (I2CS 2011)*. Gesellschaft für Informatik eV, 2011.

[LPKM17] Johannes Lipp, André Pomp, Torsten Kuhlen, and Tobias Meisen. Applying external knowledge bases and user-assisted semantic modelling to evolve a semantic knowledge graph. Master's thesis, 11 2017.

[MJO*17] Marcos Martínez-Romero, Clement Jonquet, Martin J O'connor, John Graybeal, Alejandro Pazos, and Mark A Musen. Ncbo ontology recommender 2.0: An enhanced approach for biomedical ontology recommendation. *Journal of biomedical semantics*, 8(1):21, 2017.

[MVPP14] Marcos Martínez-Romero, José M Vázquez-Naya, Javier Pereira, and Alejandro Pazos. Bioss: A system for biomedical ontology selection. *Computer methods and programs in biomedicine*, 114(1):125–140, 2014.

[MVMS16] Kristína Machová, Jozef Vrana, Marián Mach, and Peter Sinčák. Ontology evaluation based on the visualization methods, context and summaries. *Acta Polytechnica Hungarica*, 13(4):53–76, 2016.

[NSD*01] Natalya F Noy, Michael Sintek, Stefan Decker, Monica Crubézy, Ray W Fergerson, and Mark A Musen. Creating semantic web contents with protege-2000. *IEEE intelligent systems*, 16(2):60–71, 2001.

[PBS*06] Kosmas Petridis, Stephan Bloehdorn, Carsten Saathoff, Nikos Simou, Stamatia Dasiopoulou, Vassilis Tzouvaras, Siegfried Handschuh, Yannis Avrithis, Yiannis Kompatsiaris, and Steffen Staab. Knowledge representation and semantic annotation of multimedia content. *IEE Proceedings-Vision, Image and Signal Processing*, 153(3):255–262, 2006.

[PKK*03] Borislav Popov, Atanas Kiryakov, Angel Kirilov, Dimitar Manov, Damyan Ognyanoff, and Miroslav Goranov. Kim–semantic annotation platform. In *International Semantic Web Conference*, pages 834–849. Springer, 2003.

[PLMe19] André Pomp, Johannes Lipp, and Tobias Meisen. Enabling the continuous evolution of ontologies for ontology-based data management. *International Journal of Robotic Computing*, pages 19–43, 10 2019.

[PLP*18] A. Paulus, Johannes Lipp, André Pomp, Lucian Poth, and Tobias Meisen. Gathering and combining semantic concepts from multiple knowledge bases. In *ICEIS*, 2018.

[PPP*18] Alexander Paulus, André Pomp, Lucian Poth, Johannes Lipp, and Tobias Meisen. Recommending semantic concepts for improving the process of semantic modeling. In *International Conference on Enterprise Information Systems*, pages 350–369. Springer, 2018.

[PSLP03] Chintan Patel, Kaustubh Supekar, Yugyung Lee, and E. Park. Ontokhoj: A semantic web portal for ontology searching, ranking and classification. pages 58–61, 01 2003.

[ReHa05] Lawrence Reeve and Hyoil Han. Survey of semantic annotation platforms. In *Proceedings of the 2005 ACM symposium on Applied computing*, pages 1634–1638, 2005.

[SASi11] R Subhashini, J Akilandeswari, and V Sinthuja. A review on ontology ranking algorithms. *International Journal of Computer Applications*, 33(4):6–11, 2011.

[ShEu13] P. Shvaiko and J. Euzenat. Ontology matching: State of the art and future challenges. *IEEE Transactions on Knowledge and Data Engineering*, 25(1):158–176, 2013.

[TaAr07] Samir Tartir and I Budak Arpinar. Ontology evaluation and ranking using ontoqa. In *International conference on semantic computing (ICSC 2007)*, pages 185–192. IEEE, 2007.

[TASB05] Edward Thomas, Harith Alani, Derek Sleeman, and Christopher Brewster. Searching and ranking ontologies on the semantic web. 2005.

[UCI*06] Victoria Uren, Philipp Cimiano, José Iria, Siegfried Handschuh, Maria Vargas-Vera, Enrico Motta, and Fabio Ciravegna. Semantic annotation for knowledge management: Requirements and a survey of the state of the art. *Journal of Web Semantics*, 4(1):14–28, 2006.

[VAPV11] Pierre-Yves Vandenbussche, Ghislain A Atemezing, María Poveda-Villalón, and Bernard Vatant. Linked open vocabularies (LOV) website. `https://lov.linkeddata.es/dataset/lov`, 2011. [Online; accessed 25-June-2021].

[VAPV17] Pierre-Yves Vandenbussche, Ghislain A Atemezing, María Poveda-Villalón, and Bernard Vatant. Linked open vocabularies (lov): A gateway to reusable semantic vocabularies on the web. *Semantic Web*, 8(3):437–452, 2017.

[Verm16] Amandeep Verma. An abstract framework for ontology evaluation. In *2016 International Conference on Data Science and Engineering (ICDSE)*, pages 1–6. IEEE, 2016.

[Vran09] Denny Vrandečić. Ontology evaluation. In *Handbook on ontologies*, pages 293–313. Springer, 2009.

[Zieg04] Cai-Nicolas Ziegler. Semantic web recommender systems. In *International Conference on Extending Database Technology*, pages 78–89. Springer, 2004.

[ZVSl04] Yi Zhang, Wamberto Vasconcelos, and Derek Sleeman. Ontosearch: An ontology search engine. In *International Conference on Innovative Techniques and Applications of Artificial Intelligence*, pages 58–69. Springer, 2004.

A Extra Figures, Resources, and Questionnaires

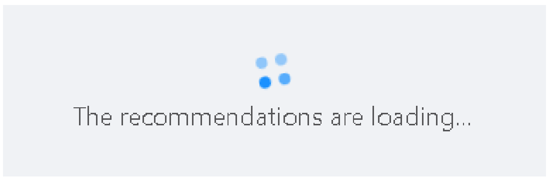

The recommendations are loading...

Figure A.1: The loading screen appears, when pressing the Get BatchRecommendation button.

Weight and Threshold List			
Metric Name	Name	Name in implementation	Value
DomainMetric	ω_α	LABEL_WEIGHT	0.5
	ω_β	COMMENT_WEIGHT	0.5
	ω_{Domain}	weightMap: MetricId.DOMAIN	1
PreSufMetric	ω_{ps}	PRESUF_WEIGHT	0.7
	ω_e	MATCH_WEIGHT	1
	ω_b	INFIX_WEIGHT	0.1
	ω_{PreSuf}	weightMap: MetricId.PRESUF	10
DescriptionMetric	ω_d	DESCRIPTION_WEIGHT	1
	$\omega_{Description}$	weightMap: MetricId.DESCRIPTION	1.5
CreatorMetric	$\omega_{r=LOV}$	LOV_WEIGHT	0.5
	$\omega_{Creator}$	weightMap: MetricId.CREATOR	1
CommonVocabMetric	$\omega_c = \omega_{CommonVocab}$	weightMap: MetricId.COMMONVOCAB	1
LOVMetric	ω_s	SCORE_WEIGHT	0.2
	ω_e	OCCURRENCES_IN_DATASET_WEIGHT	0.5
	ω_b	REUSED_BY_DATASET_WEIGHT	0.5
	γ_s	SCORE_THRESHOLD	0.5
	γ_α	REUSED_BY_DATASET_THRESHOLD	0.4
	γ_β	OCCURRENCES_BY_DATASET_THRESHOLD	0.4
	$\omega_{LOVOccurrences}$	weightMap: MetricId.LOVOCCURRENCES	1

Table A.1: Weights and thresholds table used for the calculation of recommendations during the evaluation process.

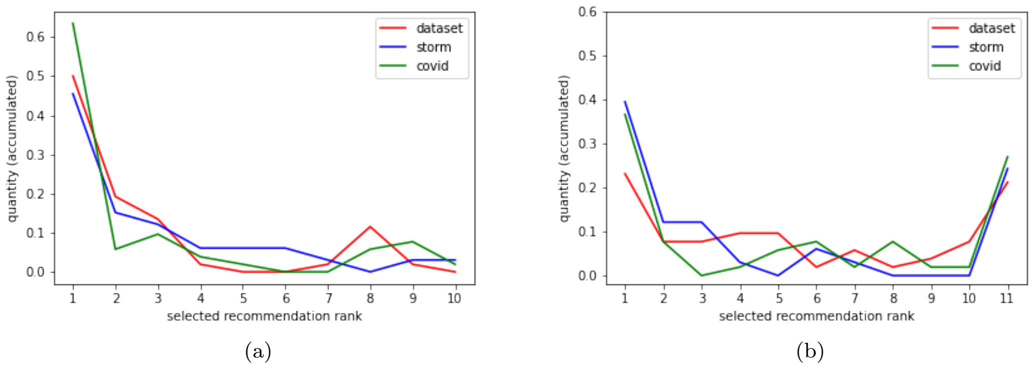


Figure A.2: The distribution of class recommendation ranks in BR (a) and LOV (b) by datasets. Rank 11 in (b) indicates all ranks which are greater than ten.

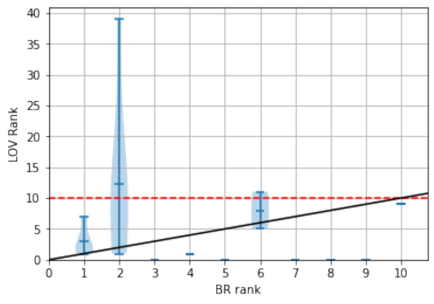


Figure A.3: The correlation of all selected property recommendations in the BR (top 10) and LOV (top 50). The blue horizontal marks display the average LOV rank of each BR rank. The black line indicates the equivalence line of the LOV ranks and the BR ranks. All LOV ranks >10 are not visible to the user without the BR re-rankings.

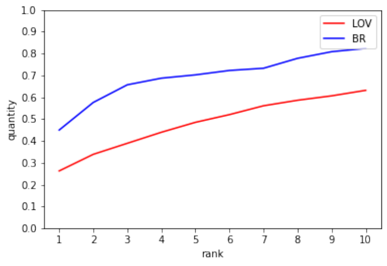
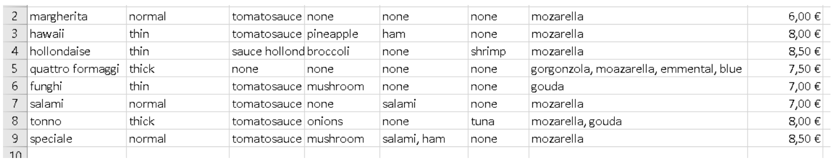


Figure A.4: The cumulative distribution of chosen class and property recommendations combined. The difference of the curves indicates the number of recommendations that can not be selected using the top 10 of the LOV rankings. In 17.7% of cases, no recommendations were chosen.

Ontology Prototyping Introduction

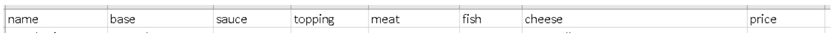
We support the understanding of ontology prototyping for users without experience as well. This is emphasized with examples in the following. The following example describes the process that is also used for the real evaluation process. We use a dataset without the header and give the user a general keyword like *pizza*, which describes where this dataset belongs to. The header is then to be modelled with relations inside Neologism using the integrated functionalities.

The following pizza dataset describes certain pizzas with different attributes. Note that we removed the header we have chosen to describe the dataset.



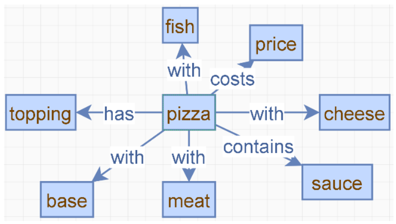
2	margherita	normal	tomatosauce	none	none	none	mozarella	6,00 €
3	hawaii	thin	tomatosauce	pineapple	ham	none	mozarella	8,00 €
4	hollondaise	thin	sauce hollond	broccoli	none	shrimp	mozarella	8,50 €
5	quattro formaggi	thick	none	none	none	none	gorgonzola, moazarella, emmental, blue	7,50 €
6	funghi	thin	tomatosauce	mushroom	none	none	gouda	7,00 €
7	salami	normal	tomatosauce	none	salami	none	mozarella	7,00 €
8	tonno	thick	tomatosauce	onions	none	tuna	mozarella, gouda	8,00 €
9	speciale	normal	tomatosauce	mushroom	salami, ham	none	mozarella	8,50 €
10								

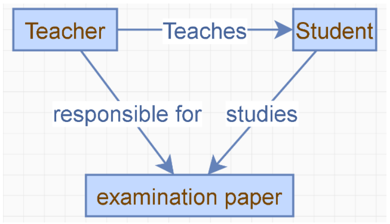
The headers you would use to describe these columns can not be *wrong* or *right.* Still, it is possible to be too precise or too general, but this is also very subjective. Next you can see the header we chose to describe this dataset.



name	base	sauce	topping	meat	fish	cheese	price

With this header, we modelled the dataset inside the ontology editor (without using the recommender process as we just want to emphasize the modelling process here).





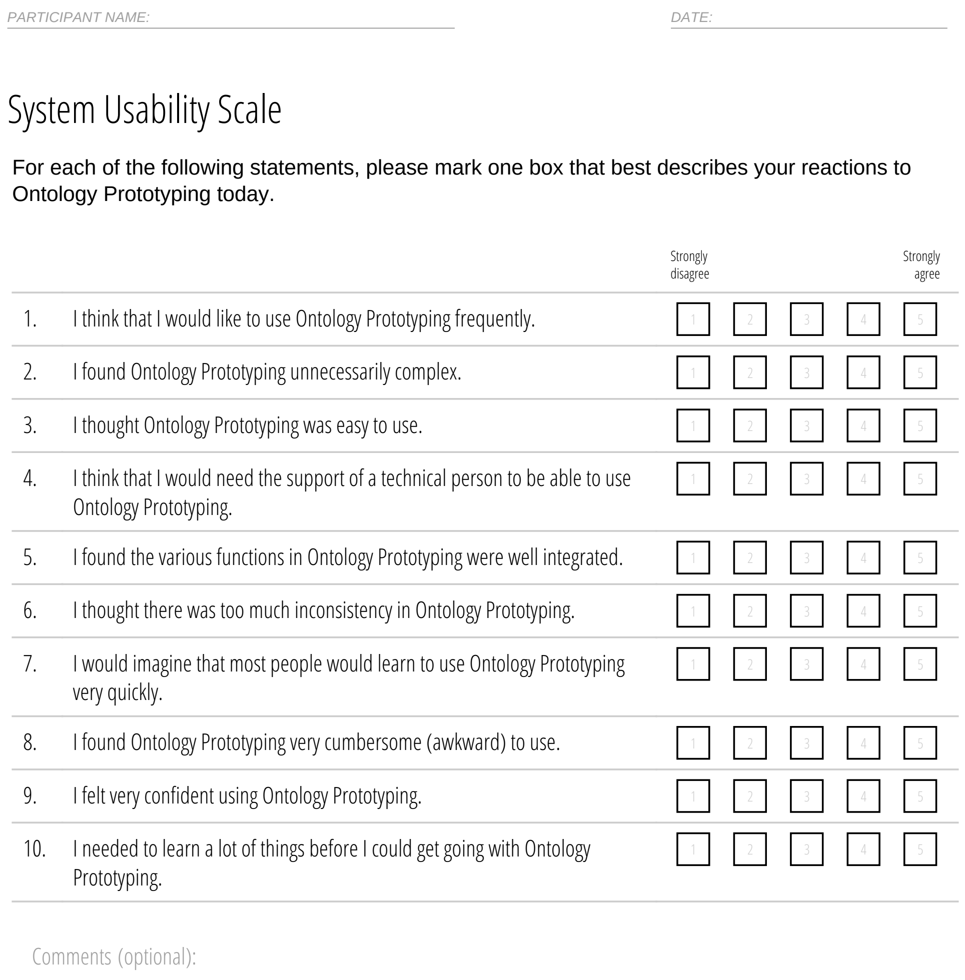
The figure on the left shows one possible solution for modelling the pizza dataset. The arrows indicate the relationship between each of the nodes (attributes of the dataset above).

To emphasize a broader view, we show another education example. The model on the right, shows a possible student-teacher-exam data-model (this data-model is a result using our recommender).

The aim of this evaluation process is to focus on the developed functionalities. After you finished the task, you will get two questionnaires to describe your experience with the functionalities and the results. During your usage of the editor, you should focus on the following functionalities:

- Domain specification (you can specify a domain inside Neologism that could enhance the ranking of recommendations)
- Live Support (During node and property creation, you will get suggestions for keywords that you could use while typing)
- Batch Recommender (When you finish the modelling process you can press a button to get recommendations from already existing ontologies)

Figure A.5: The ontology introduction document, which each participant received. It gives an overview of the task that has to be performed and examples.



PARTICIPANT NAME: DATE:

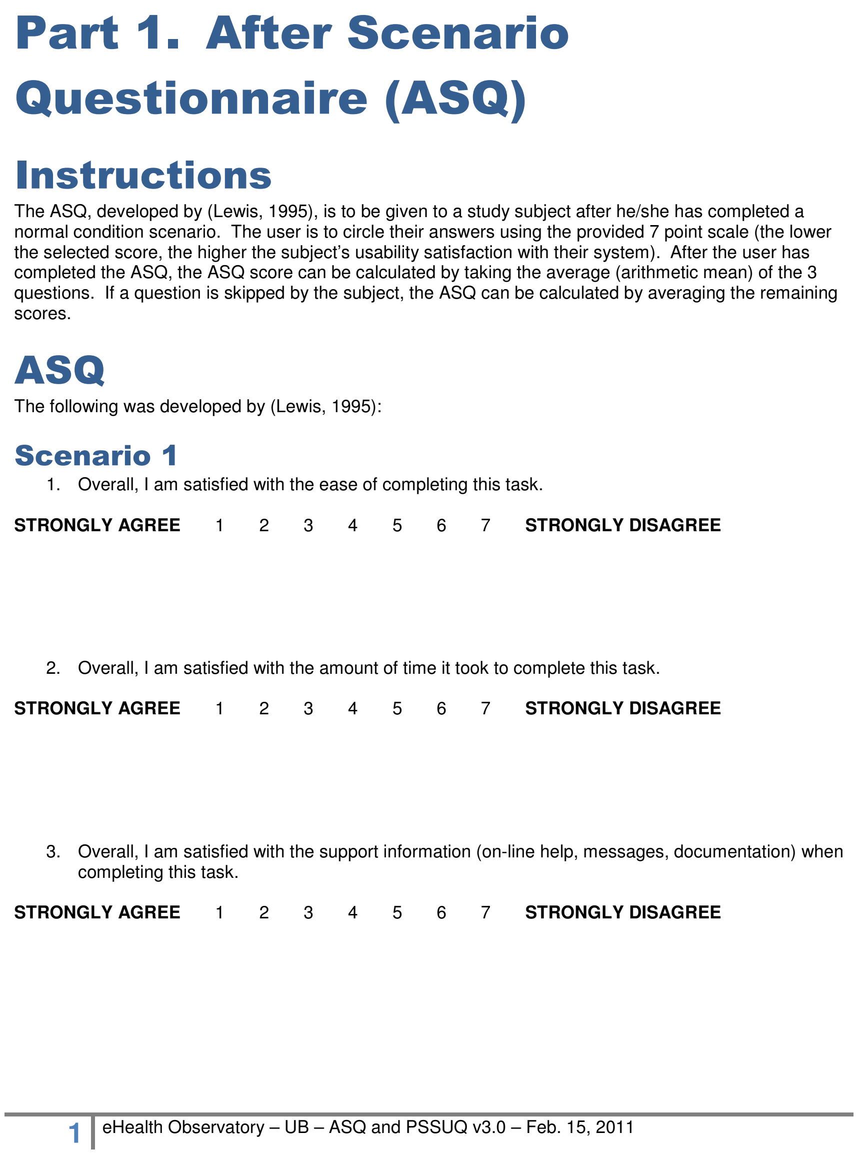
System Usability Scale

For each of the following statements, please mark one box that best describes your reactions to Ontology Prototyping today.

		Strongly disagree				Strongly agree
1.	I think that I would like to use Ontology Prototyping frequently.	1	2	3	4	5
2.	I found Ontology Prototyping unnecessarily complex.	1	2	3	4	5
3.	I thought Ontology Prototyping was easy to use.	1	2	3	4	5
4.	I think that I would need the support of a technical person to be able to use Ontology Prototyping.	1	2	3	4	5
5.	I found the various functions in Ontology Prototyping were well integrated.	1	2	3	4	5
6.	I thought there was too much inconsistency in Ontology Prototyping.	1	2	3	4	5
7.	I would imagine that most people would learn to use Ontology Prototyping very quickly.	1	2	3	4	5
8.	I found Ontology Prototyping very cumbersome (awkward) to use.	1	2	3	4	5
9.	I felt very confident using Ontology Prototyping.	1	2	3	4	5
10.	I needed to learn a lot of things before I could get going with Ontology Prototyping.	1	2	3	4	5

Comments (optional):

Figure A.6: The used System Usability Scale questionnaire.



Part 1. After Scenario Questionnaire (ASQ)

Instructions

The ASQ, developed by (Lewis, 1995), is to be given to a study subject after he/she has completed a normal condition scenario. The user is to circle their answers using the provided 7 point scale (the lower the selected score, the higher the subject's usability satisfaction with their system). After the user has completed the ASQ, the ASQ score can be calculated by taking the average (arithmetic mean) of the 3 questions. If a question is skipped by the subject, the ASQ can be calculated by averaging the remaining scores.

ASQ

The following was developed by (Lewis, 1995):

Scenario 1

1. Overall, I am satisfied with the ease of completing this task.

STRONGLY AGREE 1 2 3 4 5 6 7 **STRONGLY DISAGREE**

2. Overall, I am satisfied with the amount of time it took to complete this task.

STRONGLY AGREE 1 2 3 4 5 6 7 **STRONGLY DISAGREE**

3. Overall, I am satisfied with the support information (on-line help, messages, documentation) when completing this task.

STRONGLY AGREE 1 2 3 4 5 6 7 **STRONGLY DISAGREE**

1 eHealth Observatory – UB – ASQ and PSSUQ v3.0 – Feb. 15, 2011

Figure A.7: The used After-Scenario questionnaire.