Semantic Embeddings for OWL 2 Ontologies

Ole Magnus Holter

Thesis submitted for the degree of
Master in Programming and Networks
60 credits

Department of Informatics
Faculty of mathematics and natural sciences

UNIVERSITY OF OSLO

Spring 2019

Semantic Embeddings for OWL 2 Ontologies

Ole Magnus Holter

Semantic Embeddings for OWL 2 Ontologies

http://www.duo.uio.no/

Printed: Reprosentralen, University of Oslo

Abstract

While semantic technologies and the use of ontologies is one of the proposed solutions to the heterogeneity problem in database systems, the independent generation of ontologies leaves us with another heterogeneity problem. We need to match ontologies to allow active co-operation of different systems. This matching can often not be done by human effort alone because of the size and complexity of the ontologies. Thus automatic ontology matching systems are indispensable.

We evaluate the usefulness of embeddings, more specifically ontology embeddings both for analytical tasks and for ontology alignment tasks. We further propose OWL2Vec as a framework to create such embeddings.

The embeddings are created by projecting the ontology to an RDF-graph, conducting a series of walks on the graph and collect them in a walks document. Natural language embeddings systems such as word2vec are used to train the embeddings. To leverage ontology matching, we propose to create joint embeddings in the same vector space for two (or more ontologies). The user must then provide the system with anchors.

The results are promising for analytical tasks such as clustering and partitioning of the ontology into related concepts. We demonstrate that by using the embeddings and the provided anchors, it is possible to find new mappings between two ontologies. We also demonstrate that the structural similarities provided by the embeddings can give ontology matching systems an additional similarity score that could help such systems to decide if a potential mapping is valid.

Contents

List of Figures

List of Tables

Preface

Working with this thesis has been both a challenging and exciting journey. There are still many open questions and topics to explore. In order to limit the scope, I have concentrated my efforts on the creation of a generic system for the unsupervised generation of embeddings for OWL 2 classes and leave up to the reader the application of such a system to solve real-world problems related to ontology analysis and matching.

I would like to dedicate my master's thesis to my youngest child, Christian Nathaniel, who was born just after finishing the first semester of the master's program. Thanks go to my wife Neorlenis. I am very grateful for your love, support, your sacrifices, and your immense help. Also to my oldest son, Isak Magnus. I am grateful for your pacience, sacrifices, support, and understanding. I have invested countless weekends and after work hours in this thesis, and it has undoubtedly been, at times, both frustrating and intense for my family. Besides, I want to thank my extended family for keeping us company, for helping out with the children, and with the chores.

I would also like to thank my employer, Harald A. Møller, including its leadership and my fellow employees for being flexible and supportive all the years I have been studying and in particular, the time I have been working on this thesis.

My thanks also go to Ernesto Jimenez-Ruiz, my supervisor for excellent advice and support. Without your help, this work would not have been half of what it is.

Part I

Preliminaries

Chapter 1

Introduction

With the advances and constant improvements in the field of information and communication technology, we see an exponential increase in the amount of data available from an expanding variety of unequal sources, both structured and unstructured, employing many different technologies. In order to take advantage of the available information, we need to handle massive amounts of data from dissimilar sources with all the challenges that this poses. Semantic technology is one of the proposed solutions to this problem, and it is the technology explored in this thesis.

Matching of knowledge bases involves two main steps and many possible challenges. First, a matching must be performed at the schema level. In the field of Semantic Technologies, this is known as ontology matching. When the schema of one knowledge base differs from the schema of another knowledge base, it can be impossible to extract and compare the data between the datasets without some form of initial mapping. The field of Ontology Alignment has been extensively researched over the last decades, and vast improvements have been made. The last few years, however, have seen a slight decline in the overall improvements in the field [88].

The second step is the matching at the instance level. This task, entity matching, or link discovery in linked data, is to discover which instances in two or more separate datasets refer to the same real-world objects. Entity matching is also a mature field being widely studied due to its usefulness for data warehouses, data mining and duplicate detection. The advent of Linked Data, however, has led to a renewed interest in the field with a focus on the discovery of links between datasets.

The Machine Learning-community is also on the quest for semantics, and increasingly mature tools for capturing the semantics of natural languages are available. The use of word semantic tools such as word2vec [50, 51] has shown promising results on improving ontology matching tools [97].

Recently, numerous studies have investigated the embedding and training of neural network models using an increasing variety of different sources. These sources include RDF knowledge graphs [17, 80, 108]. However, to date, scant attention has been paid to the application of such models on the embedding of OWL 2 ontologies. Relatively little is understood about how the semantics of large ontologies can be embedded,

used to train neural network models, and applied to the task of ontology similarity in general and ontology matching in particular.

This thesis investigates the usefulness of recent neural network models to the process of matching large ontologies. Its main contribution is OWL2Vec a generic framework for the creation of semantic embedding for OWL 2 ontologies.

The thesis comprises of 7 chapters. This chapter (Chapter 1) gives a brief introduction. Chapter 2 on the facing page provides a theoretical introduction to relevant fields while Chapter 3 on page 27 gives a reasonably comprehensive overview of relevant frameworks. Then, Chapter 4 on page 51 introduces OWL2Vec, a framework for the creation of OWL 2 embeddings before Chapter 5 on page 69 presents experiments with OWL2Vec. We discuss the findings in Chapter 6 on page 91. Conclusions and suggestions for future work are found in Chapter 7 on page 101.

Chapter 2

Background

This chapter intends to provide the reader with the necessary theoretical background and set the thesis in context. It starts by introducing several broad topics relevant to the project before it narrows down on the creation of embeddings in general and embeddings of OWL 2 ontologies in particular. Details about specific systems and approaches used directly or indirectly in the project are described in chapter 3 on page 27.

The chapter is divided into nine sections. Section 2.1 deals with the Semantic Web in general. In section 2.2 on page 8, we introduce ontologies. The task of ontology matching is discussed in section 2.3 on page 10, while section 2.4 on page 17 provides a brief introduction to the task of entity matching. Section 2.5 on page 19 is dedicated to machine learning, followed by a discussion on word embeddings in section 2.6.1 on page 22. The word embeddings section paves the way for three subsequent sections on embeddings. Graph embeddings in section 2.6.2 on page 23, embedding RDF-graphs in Subsection 2.6.3 on page 24 and the specific challenges on embedding ontologies 2.6.4 on page 25.

2.1 The Semantic Web and semantic technologies

Semantic refers to the meaning of words. It comes from the French word *sémantique* which comes from the Greek word *σεμαντικος* (semanticos), *significant*, from *σεμαινειν* (semainein) meaning *to show by sign* [86].

Two central topics to the Semantic Web (SW) are the addition of abstract models of knowledge and the idea of computing with knowledge. The SW aims to extend the World Wide Web (WWW), not replacing it, but employing abstract models to create order and understanding, and to enable machines to reason and draw conclusions from the knowledge [68].

To buy something on the web today, one would need to enter different stores' web pages, compare prices, the availability of the item of interest and the delivery conditions, or use some semi-functional price comparing web-page. After deciding where to buy, one would have to enter that store, create an account and enter all the details before the purchase can be completed. With an operational Semantic Web, one could give a specification to a digital agent. The agent would seamlessly search the web on its own and, according

to the specifications, not only find the store with the best prices and delivery conditions but also perform the actual purchase [8].

Adding explicit semantics to the web is not a new idea. The idea has been around since the beginning of the WWW itself and is closely related to the WWW. The WWW's inventor and director of the W3C, Tim Berners-Lee was the first to coin the term "The Semantic Web." He has been active and inspired much of the research in this area [68, Chapter 1]. Berners-Lee envisions a web filled with machine understandable structured information that will let us construct computer programs able to reason and draw conclusions from the knowledge on the web, and shared his vision in his book *Weaving the Web*:

> *I have a dream for the Web [in which computers] become capable of analyzing all the data on the Web – the content, links, and transactions between people and computers. A "Semantic Web," which makes this possible has yet to emerge, but when it does, the day-to-day mechanisms of trade, bureaucracy, and our daily lives will be handled by machines talking to machines. The "intelligent agents" people have touted for ages will finally materialize [101]*

The goals of the WWW and the Semantic Web (SW) are overlapping. The purpose of both is the exchange of information. Both technologies aim to make data available to everyone by using interlinked resources and enabling advanced applications to search and browse them. In the newsgroup *alt.hypertext*, Berners-Lee outlined his purpose for the WWW in 1991:

> *The World Wide Web (WWW) project aims to allow links to be made to any information anywhere [. . .] The WWW project was started to allow high energy physicists to share data, news, and documentation. We are very interested in spreading the web to other areas and having gateway servers for other data. Collaborators welcome! [101]*

The ideas of the Semantic Web did not gain significant attention until the publication of the article, "The Semantic Web" in *Scientific American* in 2001 [8], where Berners-Lee presents his visions to the general public. Later, a considerable amount of research has been done on the subject. W3C has published many standards for the Semantic Web including RDF, OWL/OWL 2, and SPARQL. The availability of queryable data on the Web using the Linked Data principles, such as the Link Open Data Cloud (LOD) is increasing [55]. Some applications such as RDF Site Summary (RSS 1.0) are using the new standards, while other applications provide ad hoc solutions to exchange semantic information. The semantic data formats are used extensively on the web, and the term "web of data" has emerged to describe the part of the Semantic Web that focuses on data exchange. The more expressive semantic technologies, however, are used mostly outside the web where complex knowledge can be managed more easily [68, Chapter 1].

2.1.1 Current technology

Most of the underlying technology of the Semantic Web is already standardized under W3C. These technologies include Resource Description Framework (RDF), Web Ontology Language (OWL and its revision OWL 2) and SPARQL Protocol and RDF Query Language (SPARQL) [87].

RDF is a model for data interchange organizing data using triples ("subject"-"predicate"-"object"). The subject and predicate are usually "Universal Resource Identifiers" (URIs), and the object can be either a URI or a literal. Both the subject and the object can be represented as blank nodes enabling the construction of more complex structures and the storing of data where some information is unknown. RDF datasets are often referred to as graphs of data because of the linking through the use of URIs [77].

OWL/OWL 2 is a family of languages designed to express knowledge about concepts and relations between concepts. Its use in the Semantic Web is to create ontologies (see Section 2.2 on the following page). It is designed to express complex relationships but limited in such a way that automated reasoning is feasible [65][68, Chapter 4]. OWL 2 is a revision of OWL 1.1 that became a W3C recommendation in 2009. OWL 2 defines three sublanguages or profiles to provide the end-user with more flexibility. The OWL 2 EL is designed to be lightweight and provides polynomial time reasoning. OWL 2 QL is designed to give easier access to traditional databases, and the OWL 2 RL is designed to interact with rule-based reasoners. OWL 1.1 can also be seen as a sublanguage of OWL 2. OWL 2 formal semantics can be expressed in terms of description logic, and we assume the reader to be familiar with the basics of this field. An introduction to description logic is outside the scope of this thesis. Hence, the interested reader is referred to [43].

SPARQL is a query language for RDF datasets. From its standard 1.1 it is also able to perform insertions, deletions, and updates on the dataset. It aims to fulfill the same purpose for RDF-datasets as SQL does as a query language for relational databases.

With a knowledge base, we understand a collection of knowledge (i.e., facts) in a domain, possibly including a schema, intended to be used by a computer system to analyze and solve problems. A knowledge graph is a "large network of entities, their semantic types, properties and the relationships between the entities" [39]. Commonly, it refers to an RDF dataset.

2.1.2 Current uses

Well known usages of the Semantic technology on the web are the Linked Open Data Cloud (LOD) [100], DBpedia, Wikimedia and Google's Knowledge Graph. LOD aims to be the seed of Berners-Lee's Semantic Web, and DBpedia aspires to create a linked data knowledge base from all Wikimedia projects [15]. Wikidata [105] is an editable knowledge graph that provides structural data for all Wikimedia projects. Google's Knowledge Graph [33], introduced in 2012, integrates structural data into Google search and other Google services. When searching, it provides direct access to

relevant information, and this information is presented to the user in a box or a carousel. These initiatives are rapidly transforming the web from being purely document-oriented into a distributed database [22].

The more expressive technologies are, however, easier to exploit in closed, controlled applications than on the web due to several unresolved challenges (section 2.1.3). Semantic technology is, for instance, used extensively in biology and medicine. SNOMED CT [92], National Cancer Institute Thesaurus (NCI) [53], the UMLS Metathesaurus [102] and the Foundational Model of Anatomy (FMA) [21] are prominent examples.

2.1.3 Challenges

We are still facing some challenges before the semantic web can emerge as envisioned by Berners-Lee.

One challenge is the interaction of applications using different vocabularies (ontologies). Consequently, we need effective and efficient methods for matching ontologies [88]. Furthermore, we need to face the issue of trustworthiness. When anyone can express anything, we will need some way of locating relevant data and ensuring that it comes from reliable sources. We will also need to be able to reason why a given result is trustworthy. Another current challenge is the lack of links between datasets already on the Linked Open Data Cloud. According to [21] 44 % of available Linked open datasets are not connected to other datasets. The lack of links is partly because the linking of datasets can be an overwhelming task. The size of the datasets makes it virtually impossible to do this manually, thus the need for effective and efficient methods for automatic link discovery [55].

In [22], Gandon, Sabou, and Sack describe the current challenges of the Semantic Web as "the many S of the Semantic Web"; scalability, storage, search, semantics, security, streaming. The vast scale of the web makes scalability, storage, and search, topics of active research. Other topics such as access control, version management, and long-term preservation are crucial to provide reliable and trustworthy services.

2.2 Ontologies

The word ontology comes from the Latin word *ontologia*, a metaphysical science, which itself is a compound word from ancient Greek $ο\nu\tau ος$ (ontos), *being*, genitive of present participle of $\epsilon\iota\mu\iota$ (eimai) meaning *to be* and $\lambda ο\gamma ος$ (logos), *a spoken word* from the root $\lambda\epsilon\gamma\epsilon\iota\nu$ (legein) *to speak* [62].

In metaphysics, ontology is the study of existence and being. The essence of what it will say to be and what it means to be a thing. A tradition started by the Greek philosopher Platon (429–347) and expanded by his student Aristoteles (384–322) in the endeavor to define reality and the nature of things [68, Chapter 1].

From the 1970s, in the artificial intelligence (AI) field, it was argued that a conceptual, explicit model of the real world could prove to be invaluable [47]. This idea drew on the mathematical research on logic and automated

reasoning and inference. Researchers recognized that having such a model, AI systems could, "mathematically" infer new, implicit, knowledge based on its model of the real world.

In the 1990s, the ontology was increasingly seen as a separate layer in database systems. Gruber, in his paper "Toward Principles for the Design of Ontologies Used for Knowledge Sharing", where he describes how the creation of an ontology can be regarded an engineering art, was one of the first to define the ontology in computer science as an "explicit specification of a conceptualization".

With the increased interests in the Semantic Web seen in the early 2000s, numerous researchers began to focus on ontologies as a way to share definitions and rules in a domain of knowledge. RDFS became a W3C recommendation for ontologies in the early 2000s, OWL 1.1 the in 2004 and OWL 2 from 2009 [66, 76].

Ontologies can be seen as an abstraction in data modeling [28]. It serves the same purpose for data models in general as the database schema, and integrity rules do for a regular database, but the ontology operates at a higher level of abstraction, above implementation language and implementation details. In databases, a tuple in a table must conform with constraints defined in the schema. These constraints could be for example datatypes constraints, primary and foreign keys or check constraints. Any deviation from these rules would render the database inconsistent. The same way, a statement in any dataset using an ontology conforms to the axioms expressed in that ontology. The expressiveness of an ontology language is, in general, higher than data definition languages of regular databases (like SQL's DLL) and closer to first-order logic than its database counterpart. Therefore it is considered to be at the semantic level.

We use ontologies as a formal, explicit, description of knowledge about a domain of interest. The ontology provides a vocabulary, used by an application, for describing the domain of interest along with a specification of the meaning of the terms and the relationships between terms [88]. An ontology can be reused, and it can provide a shared vocabulary for multiple applications in the same domain of interest. The ontology typically deals with entities such as classes, individuals, relations, and data types [34, Chapter 2]. Today an ontology is commonly expressed in OWL/OWL 2, W3C's recommendation and the emerging standard for ontologies in semantic applications. Simple ontologies can also be expressed in RDFS (RDF Schema). Other languages for ontologies also exist [60][68, Chapter 1].

An OWL-ontology can be expressed using several different syntaxes. It can, among others, be expressed using RDF/XML-syntax [75], the turtle syntax [74], the Manchester OWL syntax [63], and the OWL/XML-syntax [64].

Some common OWL-axioms are described in Manchester syntax and its equivalent DL notation in Table 2.1 on the following page where C and D are two arbitrary OWL-classes and R and S two arbitrary object properties.

Semantic Technology handles semi-structured data using triplestores and can use ontologies to describe its vocabulary. An ontology is somewhat similar to a database schema in that it provides a vocabulary and constraints

OWL-axiom	DL-expression
C SubClassOf: D	$C \sqsubseteq D$
C DisjointWith: D	$C \sqcap D \sqsubseteq \bot$
C EquivalentTo: D	$C \equiv D$
C SubClassOf: R some D	$C \sqsubseteq \exists R.D$
C SubClassOf: R only D	$C \sqsubseteq \forall R.D$
R InverseOf: S	$R \equiv S^{-}$
R Domain: C	$\exists R.\top \sqsubseteq C$
R Range: C	$\top \sqsubseteq \forall R.C$

Table 2.1: Common OWL-axioms

the meaning of that vocabulary.

Ontologies today are one of the basic building blocks of the Semantic Web stack. They provide data model designers with reusable vocabulary at a high level of abstraction without regard to implementation details. This vocabulary is essential to facilitate the integration of databases and the interoperability of systems. Ontologies are also central to automated reasoning and inference in semantic applications. By reasoning and inference, machines can evaluate and draw conclusions based on the data and the resources and rules expressed in the ontology. An ability that can improve data quality and data completeness, for instance, by adding relationships between resources and discover inconsistencies in the dataset [32].

2.3 Ontology matching

Regardless of the domain of interest, two applications using different ontologies cannot be assumed to be able to interoperate directly. They would not be able to "understand" each other's vocabulary. Ontology matching aims to solve this problem. Thus it is part of the solution to the problem of handling data from heterogeneous data sources. It plays a vital role in ontology integration, data integration and data warehouses [88].

Ontology matching is the process of matching two (or more) ontologies. A mapping is a correspondence between two entities from different ontologies (E_1 and E_2) and the relation between them (R). More formally, the mapping $\langle E_1, R, E_2 \rangle$ (or simply $E_1\ R\ E_2$) states that E_1 and E_2 are R-related. R is usually an equality relation, but it can be another relation, for instance, `owl:subClassOf`. A mapping can include additional information such as a confidence score and a mapping id. The result of a matching process is a set of mappings. This set is called an *ontology alignment*.

Many different matching techniques and strategies and are used for ontology matching. From simple string comparison techniques to more advanced global similarity techniques such as comparing relations and

using semantic reasoners. Some systems also work with ontologies written in different ontology languages, but this is out of the scope of this thesis. We consider exclusively the matching of ontologies expressed in OWL 2.

Ontology matching is related to schema matching for relational databases. Schema matching has been researched since the early 1980s when distributed databases were starting to become available to corporations. Batini, Lenzerini, and Navathe describe the emerging challenge in the paper "A Comparative Analysis of Methodologies for Database Schema Integration" from 1986:

> *Database integration is a relatively recent problem that has appeared in the context of distributed databases.*
>
> *...*
>
> *With the increasing use of databases, we expect the integration problem to be more severe and pervasive. New technologies of networking, distributed databases, knowledge-based systems, and office systems will tend to spread the shared use of data in terms of number of users, diversity of applications, and sophistication of concepts. Design, manufacturing, and engineering applications are becoming centered around database management systems. The need for methodologies for integrating data in its diverse conceptual and physical forms is thus expected to increase substantially [6].*

In hindsight, we understand that this prediction was right on target. The issue of heterogeneity has not diminished but instead increased substantially. The database community is still researching ways to reduce this problem automatically and semi-automatically. However, it remains an open question [88].

Schema matching in databases is performed at a syntactical level. A system can compare strings and data types, keys and constraints. However, with the interest in ontologies in the semantic web community from the early 2000s, the heterogeneity challenge shifts to a higher level of abstraction. Now the challenge lays on a more semantical level, and less dug down in implementation details. The ontologies can, however, be very rich. They tend to be much larger and more complex than database schemas. This vastness and complexity make it challenging to create matchings manually.

2.3.1 A simple matching example

Let *ns* and *ex* be two arbitrary namespaces and $\mathcal{O}_1$ an ontology with the following concepts with the natural hierarchical relations: *(i)* Product, *(ii)* Book, *(iii)* CD, *(iv)* Person, and *(v)* Author. Let $\mathcal{O}_2$ be a second ontology with the following concepts with the natural hierarchical relations: *(i)* Monograph, *(ii)* Essay, *(iii)* Political essay, *(iv)* Biography, *(v)* Literature, *(vi)* Person, and *(vii)* Writer.

Applying string equality on the concept labels to match the two ontologies, we find only one mapping (*ns:person* $\equiv$ *ns:person*). If we enhance the vocabulary with synonyms, as shown in Table 2.4 on page 13, we find more mapping candidates:

1. ns:book ≡ ex:essay
2. ns:book ≡ ex:political_essay
3. ns:book ≡ ex:bibliography
4. ns:person ≡ ex:person
5. ns:author ≡ ex:writer

This result is probably not what we want. Some systems include reasoning to improve the candidate mappings. We can do this by first identifying a set, $\mathcal{M}$, of reliable mappings. For example by using very similar strings (e.g., person in $\mathcal{O}_1$ and person in $\mathcal{O}_2$), and see if there are some matches also from neighbor to neighbor. In this case we consider ns:person ≡ ex:person to be reliable. The reliable mapping is added to the final mapping set. Then, we can check the other candidate mappings for inconsistency against the growing set of final mappings before adding them. In this simple case, adding ns:book ≡ ex:bibliography after ns:book ≡ ex:political_essay would lead to inconsistency if ex:political_essay and ex:bibliography are declared to be disjoint. Structural techniques can help us avoid mapping ns:book with both ex:essay and its subclass ex:political_essay. A more sophisticated system could even find that all three candidate mappings with ns:book should be considered subclasses.

URI	label
ns:product	Product
ns:book	Book
ns:cd	CD
ns:author	Author
ns:person	Person

Table 2.2: The URIs and labels of $\mathcal{O}_1$ in the simple matching example

URI	label
ex:monograph	Monograph
ex:essay	Essay
ex:political_essay	Political essay
ex:bibliography	Bibliography
ex:person	Person
ex:writer	Writer

Table 2.3: The URIs and labels of $\mathcal{O}_2$ in the simple matching example

URI	Synonym
ns:product	Product
ns:product	Goods created
ns:product	Creation
ns:book	Book
ns:book	Essay
ns:book	Fiction
ns:book	Novel
ns:cd	CD
ns:cd	Album
ns:author	Author
ns:author	Writer
ns:author	Creator
ns:person	Person

Table 2.4: The URIs and labels of $\mathcal{O}_1$ in the simple matching example enhanced with synonyms

2.3.2 Element based matching strategies

Element-level techniques are the most straightforward techniques, and consider the entities or instances in isolation without taking into account the relations with other entities or instances. Most of these techniques consider strings as tokens, while others use language and statistical approaches [34].

One of the simplest and most important strategies for finding corresponding entities between ontologies is comparing the labels. In the example in Subsection 2.3.1 on page 11 the comparison of labels finds the mapping `ns:person` $\equiv$ `ex:person`. This strategy has, however, many shortcomings. We would not find concepts described by different synonyms, and homonyms would provide false mappings. The creators of one ontology could have preferred "author" over "writer" which would is not possible to find using string comparison. It could also be possible that the string "name" in one ontology refers to article name and the string "name" in another ontology refers to the author's name and can thus provoke a mismatch.

Another challenge with string comparison is that ontology labels can be in different natural languages. The word "name" in English would be "nombre" in Spanish and "navn" in Norwegian. There are multiple strategies for solving these problems. These include normalization of strings, language normalization, and the use of external resources such as lexicons, dictionaries, and thesauruses. Many systems use WordNet for this, but it seems possible to improve upon synonyms and multilingual ontologies using word embeddings [97].

Some ontologies provide more lexical information than just the labels. The URIs could have lexical meaning. There could be useful information

stored as comments. There could be synonym fields and other annotation information available. Matching systems could take advantage of these sources of lexical information as well. The XMap-system, for instance, aggregates the id, label, and comment and applies string based measures on the resulting string [42].

There are many ways of determining string similarity. Often the strings must first be normalized in some way. For instance, substituting all capital letters with lowercase letters, accents may be removed, whitespace normalized and punctuation removed. Other techniques include stemming and removing stop words. It is possible to calculate a similarity score using one (or several) of many methods from simple string equality measures to more elaborate token based distance measure methods [34].

2.3.3 Structure-based matching strategies

Another matching strategy is to consider the structure of ontologies and entities. A matching system can compare the definition of an entity along with its properties such as data types, values, and related entities. It can also use the relational structure. One way is seeing the dataset as a graph and trying to find similar subgraphs. A second way is to exploit the `subClassOf`-relations. A third strategy is using semantic-based techniques. Semantic-based techniques use reasoning to improve alignments and find inconsistencies [34, Chapter 4]. It can involve the use of other ontologies as background ontologies to help understand the relations between terms.

While the structure-based techniques are useful, they are usually not sufficient to identify most matches. Global structure-based techniques are therefore mostly used in combination with other matching strategies such as string similarity measures.

2.3.4 Semantic-based matching strategies

Semantic techniques are used to ensure completeness and consistency. It finds correspondences that would lead to inconsistencies in the alignment. They are model-theoretic techniques used to justify results, and they do not perform well alone on ontology matching task. The matching system or the user must provide semantic-based approaches with anchors, entities that are declared to be equivalent. Semantic methods then act as amplifiers of these seed alignments.

Two semantic-based approaches have been used in ontology matching; Propositional Satisfiability (SAT) techniques and description logic (DL) techniques [34].

2.3.5 Other strategies

Current systems employ a wide range of other matching strategies as well. Some examples are matching with background knowledge, user involvement in the matching process and social and collaborative matching [88]. To be able to match large ontologies, it is necessary to reduce

the search space. Testing the cartesian product of classes and properties for equality is very ineffective. Strategies to reduce search space include blocking and filtering strategies. One could, for example, create blocks from the ontology where each block contains classes with labels with equal substrings and only test for equality only within each block.

2.3.6 Matcher requirements

Applications in need of ontology alignments can put very different requirements on the alignments systems. Some alignments are created on design time of an application, for example in data warehousing. The effectiveness of the alignment system is most important, and the efficiency is of less importance as it is performed only once. In other cases, the alignment is needed at runtimes, such as in peer-to-peer systems and web services. In this context, efficiency is essential, and there must be a trade-off between effectiveness and efficiency. In some circumstances, it is crucial that the alignment is complete (it must discover all the links), but it can discover some mappings without causing too much trouble. Other applications need it to be the other way around. Precision is the most important, and false mappings could be devastating.

The process of matching ontologies can, on small datasets that need ontology alignment on design time, be done manually. In real-world problems, however, the ontologies can often contain thousands of classes and complex relationships. The task of manually discovering thousands of correspondences is virtually impossible. Consequently, there is an emerging need for automatic and semi-automatic systems capable of aligning ontologies both effectively and efficiently.

2.3.7 Evaluation of systems

The Ontology Alignment Evaluation Initiative (OAEI) is a coordinated international initiative for the evaluation of ontology matching [2]. Since 2004 they hold yearly events where manufacturers of ontology alignment tools can test their tools. OAEI aims to evaluate the different systems' and algorithms' performance and inspire overall improvements.

To be able to determine if one ontology matching tool is better than another, it is necessary to have some standard measures on performance. OAEI measures precision (equation 2.1 on the next page), recall (equation 2.2 on the following page) and F-measure (equation 2.3 on the next page) which is an aggregate of the two. To be able to get a clear picture of the different system's performance, a set of systematically generated test benchmarks is used. If the test suit is stable over time, the evolution of the systems can also be measured [88].

Precision [34] Given a reference alignment R, the precision of some alignment A is a function $P : \Lambda \times \Lambda \rightarrow [0\ 1]$ such that:

$$P(A,R) = \frac{\mid R \cap A \mid}{\mid A \mid} \tag{2.1}$$

Recall [34] Given a reference alignment R, the recall of some alignment A is a function $P : \Lambda \times \Lambda \rightarrow [0\ 1]$ such that:

$$R(A,R) = \frac{\mid R \cap A \mid}{\mid R \mid} \tag{2.2}$$

F-measure [34] Given a reference alignment R and a number α between 0 and 1, the F-measure of some alignment A is a function $F_\alpha : \Lambda \times \Lambda \rightarrow [0\ 1]$ such that:

$$F_\alpha(A,R) = \frac{P(A,R) \times R(A,R)}{(1-\alpha) \times P(A,R) + \alpha \times R(A,R)} \tag{2.3}$$

If $\alpha = 1$ the F-measure is equal to precision and if $\alpha = 0$, the F-measure is equal to recall. $\alpha = 0.5$ is a common value.

2.3.8 State of the art

After more than a decade of research on ontology matching, there is now a myriad of different ontology matching systems available, and they differ in many important ways. Some take only OWL-ontologies as input while others can take RDFS as well. Some systems are even able to match XML or relational databases. Most systems only find one to one equality relations, while more sophisticated systems can find other types of relationships as well. Some applications, come with GUI, but most are command line only. Many of the systems are domain specific and focus on finding alignments on, for instance, biological datasets and some are general purposes.

Jerome Euzenat and Pavel Shvaiko group ontology matching systems into four categories based on their main matching strategies [34]. The first category is schema-based systems, which depend mostly on schema-level input. The second category of matching strategies is instance-based systems, which depend mostly on the data indexed by the ontologies. Thirdly, we have the systems employing a mix of these two strategies. The fourth category is Meta-matching systems which combine other matching systems instead of performing the matching themselves.

Achichi *et al.* provide in [2] and [1] a thorough description of the tests and results of OAEI's events in 2017 and 2016 respectively. Jerome Euzenat and Pavel Shvaiko give an overview of the similarities and differences of many available ontology matching systems in [34]. Among the systems considered at OAEI's event in 2017 [2] we find ALIN, AML, CroLOM, DiSMatch, I-Match, KEPLER, Legato, LogMap, njuLink, ONTMAT, POMap, RADON, SANOM, Silk, Wiki2, XMap, and YAM-BIO. The results from 2018 are found in [4].

2.3.9 Challenges

Some of the current challenges in the field of ontology matching are the need to match even more large-scale ontologies and the need for more efficient systems for real-time applications. Few systems are taking advantage of available background knowledge, and that the systems are challenging to use for end-users [88].

Most systems focus on the detection of simple equivalence mappings [34, Chapter 8]. However, the detection of other relationships such as subsumption relationships is also useful. Other, more complex mappings could also be part of an alignment. For example, if $\mathtt{C}$, is a class of one ontology and $\mathtt{D}$ and $\mathtt{E}$ are classes of another ontology, $\mathtt{C} \sqsubseteq \mathtt{D} \sqcap \mathtt{E}$ is a possible mapping.

Most ontology matching systems depend primarily on lexical similarity measures to decide if a candidate mapping is valid. Even though they employ simple structural techniques to help the process, it would be desirable to capture the semantics of the entities. Vector space embeddings is a possible means to express the semantics of OWL 2 entities.

The role of the user in the matching process has received relatively little attention. A domain expert user could provide a matching system with valuable information and improve the quality of the resulting alignment. However, how to involve a user in the matching process without having to understand the technicalities of the ontology matching process is still poorly understood.

2.4 Entity matching

While ontology matching mainly focus on matching the schema of the knowledge base, entity alignment (or link discovery for linked data), focuses on matching the instances indexed by the ontology and resolves links between datasets.

An automatic entity matching tool takes as input two, or more, datasets to be linked, configuration parameters, and possibly background knowledge resources. The datasets can be RDF/OWL dumps or SPARQL endpoints [55]. We define a correspondence, or a link, between two entities e_1 and e_2 from different datasets as a tuple $\langle e_1, e_2 \rangle$. The tuple can include additional information such as a confidence score and an id. The output is a set of correspondences between the two datasets.

The Link Data Cloud is based on the link data principles. It depends on the interlinking of datasets with other datasets. In 2014, Schmachtenberg, Bizer, and Paulheim pointed out that 44 % of the LOD datasets are only the target of RDF links from other datasets or completely isolated [85]. They are not themselves connected to other datasets. This lack of links is a significant problem as the links are essential to many applications [22].

If we are to merge (or link) distinct datasets, we need to be able to detect which, if any, elements refer to the same real-world objects. The same challenge arises when one is trying to find duplicated instances in one

dataset. Entity matching proposes a solution to this problem. The goal of the process is to detect pairs of resources automatically. It usually identifies `owl:sameAs` relations [55].

LOD now consists of more than 10.000 datasets, and the creation of links between large datasets is a daunting process. Manually discovering links at this scale is not possible. Therefore an automatic or semi-automatic tool is needed to aid the process. This tool needs to be both highly effective and very efficient to be able to discover most of the links among large datasets. Consequently, link discovery and the related problems of entity resolution and object matching are being studied extensively [55].

Because knowledge bases typically are large and semantically different, it is challenging to solve this problem with both high effectiveness and efficiency. Although the problem of entity resolution has been studied extensively, the focus has mostly been on similar and relatively simple datasets. The resources described using Linked Data are, however, potentially very heterogeneous and interrelated with other datasets. Research on entity resolution has relied primarily on equality relations while link discovery can involve not only `owl:sameAs`-relations but also other relations. Linked data can also have an ontology describing vocabulary, resource properties and relations (see Section 2.2 on page 8) that can aid the process. However, the ontologies must be matched as well [55].

2.4.1 Requirements

As with ontology alignment, we need the entity alignment tools to be as effective as possible in discovering the positive links between datasets, and also in not identify links where such a relationship does not exist. There are different requirements in terms of efficiency between offline and online systems, but the vastness of many relevant datasets makes efficiency crucial. Other requirements for entity alignment tools can be a low manual effort for configuration and tuning and the support for online background data in the form of linked data [55].

2.4.2 Existing technology

Typically, a link discovery tool is capable of using multiple techniques for calculating similarity on the potential instance matches. As for ontology matching techniques (section 2.3 on page 10), we can categorize link discovery techniques as element-based or structure-based. Element-level matchers are the simplest and also the most common. Background knowledge can be used to help similarity calculation. The ontology can also be exploited, for instance by limiting the search to equivalent classes. Another approach is to use existing links to find new links. Using the transitivity of the `owl:sameAs`-relation it is possible, by composing several links to derive new `owl:sameAs`-links [55].

Link discovery tools have not been evaluated and compared as much over the last decades as having ontology alignment tools. However, interest in such systems is increasing.

Several systems capable of link discovery have been developed in recent years, we have for example ReMom, KnoFuss, AgreementMaker, Silk, CODI, LIMES, LogMap and more.

The systems differ in terms of functionality as well as in terms of input. Some take RDF-dumps while others also can take SPARQL endpoints as input. The systems' flexibility and configuration effort differ. Some offer post-processing, to check for data consistency and logical contradictions. The user interface also varies from simple CLI-tools to more sophisticated GUI-interfaces. Most tools, however, focus on simple property-based matching techniques and are not taking advantage of ontologies, existing links, and available background knowledge. The tools also need to take advantage of efficiency techniques such as blocking, filtering, and parallel processing [55].

2.4.3 Evaluation of systems

There is an effort, from 2009 by the OAEI to, in addition to evaluating ontology matching tools, also to evaluate link discovery tools [55]. The evaluation of link discovery systems and methods are, however, still immature, but able to give a reasonable indication of the relative effectiveness of comparable systems. OAEI has mainly focused on effectiveness. The efficiency of the systems, however, has not been evaluated. The participation, in the contests on link discovery, has been rather low, thus to date most Link Discovery tools have not been evaluated.

2.5 Machine learning

Samuel coined the term Machine lEarning in 1959. Machine learning is the field of computer science that gives computers the ability to "learn" without being explicitly programmed [84]. It started as a study of pattern recognition and computational learning theory in artificial intelligence. Machine learning creates algorithms that can learn from a large body of data and make predictions on the data. Machine learning is today applied to a wide range of tasks, such as email filtering, detection of networks, hacking detection, optical character recognition, learning to rank, and computer vision [67].

2.5.1 Machine learning and SW semantics

Discussing the future of the Semantic Web community, Gandon, Sabou, and Sack argue in [22] that with the growing complexity and scale of Linked Data, the community must address the development in a more multidisciplinary manner. The authors particularly mention the common interest in intelligently processing of data to the machine learning community and the Semantic Web community. The last few years we have seen an increased interest in hybrid approaches where description logic techniques and reasoning are combined with machine learning.

The Semantic Web-community has a very concise sense of "semantics" where RDF gives us a way to state facts. OWL gives us vocabularies to describe resources and the semantics of those resources (see 2.1.1). Semantic Technology uses triples to state facts in a way that both people and machines can understand. The machine learning community's interest in semantics is more about generalizing from data. In a discussion of the connection between these two "semantics" technologies, DuCharme argues that while "RDF-based models are designed to take advantage of explicit semantics", neural network models can "infer semantic relationships and make them explicit." [17].

One can imagine that modifications could be made to programs like word2vec so that they output OWL-triples, contributing to systems using ontologies. At the same time, machine learning models could be trained with instances along with declared classes in an ontology. Running this machine learning model on new data, it can do classifications that take advantage of the ontology. While machine learning systems are designed around unstructured data, they could take advantage of and do even more with structured data. In other words; the machine learning and the Semantic Web-community can take advantage of each others' "semantics." While the Semantic Web can use machine learning to infer explicit semantics from data, machine learning can use the explicit semantics of the semantic web to do more.

2.5.2 Machine learning techniques in ontology matching

Utilizing machine learning techniques, it is possible for an ontology alignment application to identify alignments between ontologies by first learning from some sample data with already established alignments. There are multiple possible learning strategies. The naive Bayes learner, WHIRL, Support vector machines, decision trees, and neural networks are some examples [34]. Two examples of ontology matching systems that have employed a machine learning strategy for matching ontologies are YAM++ [56] and GLUE [16].

The learner can exploit different types of information. From word frequency, format, positions or properties of value distributions. Results produced by various learners can be combined with the help of a meta-learner [34].

Some additional challenges arise when using machine learning for ontology alignment. First, the model has to be trained and tested with both positive and negative matches. Issues like overfitting and appropriate method parameter selection have to be addressed as well. A previously learned matcher can be reused [34].

2.5.3 Neural networks

In this thesis, the machine learning models explored are embedding models based on Artificial Neural Networks (neural networks).

The neural network is not a new idea [109]. Already in the early 1940s, it was shown that simple neurological networks could solve complex arithmetical problems of different kinds. In the 1950s several "neurocomputers" were constructed that were able to solve different proposed problems, but they were not able to use them to solve real-world problems. Quickly they were seen as an area that promised much and delivered little. So the enthusiasm diminished. In the 1970s little was heard from the community, but the community made considerable advances. Research on neural network saw a renewed interest from the mid-1980s. Today neural networks are used for a wide array of applications and are especially successful in pattern recognition such as image recognition. It has recently also been successful in other fields as well such as the unsupervised generation of word embeddings.

Neural networks are made up of nodes and weighted connections between them. The nodes are grouped into layers. The network has an input layer, an output layer and can have none, one or more hidden layers. Usually, each node in a hidden layer is connected to all nodes of the preceding and the following layer.

In ontology alignment, neural networks have been used for various tasks such as discovering correspondences via categorization and classification [20, 44] or learning matching parameters [34].

The work of [51] demonstrates that with the progress of neural network techniques in recent years, it is no longer only a choice between simple embedding techniques on vast amounts of data or complex models on smaller datasets. It has now become possible to train more complex models on much larger datasets.

2.5.4 Embedding models

One of the recent successes of the neural network models is the unsupervised generation of embeddings. To *embed* is defined as "to fix (an object) firmly and deeply in a surrounding mass" [18]. In machine learning, the term usually means representing objects in a real number vector space.

Embeddings reduce complex models to fewer dimensions while retaining the objects' characteristics. Moreover, machine learning models work mostly on numerical data, so having embeddings allows us to perform complex analysis tasks on new types of data.

Several recent systems (e.g., word2vec, `fastText`, StarSpace) demonstrate that it is possible to train high-quality word vectors using simple model architectures. Because of the lower computational complexity, it is possible to compute accurate high-dimensional word vectors from a much larger dataset. They also show that different characteristics easily can be combined into one representing vector [38, 51, 108].

Other works show that the techniques used to train word embeddings can be used to embed other data types as well. DeepWalk uses the same algorithm to create embeddings from social network graphs by creating walks on the graph and considering the nodes as words [72].

RDF2Vec applies the DeepWalk algorithm and creates embeddings from RDF graphs [80].

2.5.5 Vector similarity

An important property of the embeddings is that similar objects are close in the vector space. We, therefore, need a way to measure distance in vector space. A common way to measure vector similarity is using cosine similarity [14]. If A and B are two non-zero vectors, cosine similarity ($\cos(\theta)$) is defined as:

$$\cos(\theta) = \frac{A \cdot B}{\|A\|\|B\|} \tag{2.4}$$

The magnitude or length of a vector ($\|V\|$) with n dimensions and v_i is the value of the i-th element in the n-tuple. It is defined as its Euclidean length:

$$\|V\| = \sum_{i=1}^{n} v_i^2 = \sqrt{v_1^2 + v_2^2 + \cdots + v_n^2} \tag{2.5}$$

2.5.6 Visualizing the vector space

Even though we speak about word vectors as being low dimensional, they are usually in the range of 50 and 500 dimensions. Consequently, it is not possible to visualize them directly. We have to reduce the dimensionality to 2 or 3 dimensions to be able to get an impression of how the elements relate to each other. One way to do this is to use Principal Component Analysis (PCA). This technique reduces the number of dimensions by identifying the dimensions that contribute the most to the variation among the vectors [11].

2.6 Machine learning embeddings

2.6.1 Word embeddings

A word embeddings system aims to represent words as elements in a real numbered vector space. Representing words as vectors has a long history, but only recently it has become feasible to train high-quality word vectors on large amounts of data. Current techniques train word embeddings by maximizing the probability of the co-occurrence of words within a given context.

Word embeddings is a fundamental concept in the field of natural language analysis. It allows, not only to do analysis on texts, documents, and words but also capture the semantics of words.

The problem of calculating the probability distribution of words in a text is rendered intractable by the number of different words (variables) involved. Consider the problem of calculating the joint probability distribution of only 10 consecutive words. If we have a vocabulary of 100 000 words, there would be $10^{50} - 1$ potential free parameters[7].

In 2000, Bengio, Ducharme, and Vincent proposed to "fight the curse of dimensionality" by learning a "distributed representation for words" [7].

Mikolov *et al.* elaborate on these ideas and propose two improved, yet simpler models, in [51] and [50]. The Skip-gram and the CBOW models can train the embeddings on a much larger corpus. We discuss specific systems in more detail in Section 3.2 on page 29.

A word embeddings system takes as an input a preprocessed corpus of text. Depending on the application of the word embeddings, preprocessing techniques could include the removal of accents, commas, case normalization, removal of stop words and stemming.

The system outputs an n-dimensional vector for each word in the vocabulary. The embeddings would ideally conserve each word's characteristics in relation to other words in the corpus in such a way that semantically similar words have similar vectors.

2.6.2 Graph embeddings

Much knowledge is naturally expressed as graphs. One example is the hierarchical relationships between species in biology. Another is social graphs as relationships in social networks. Being able to perform analysis on graph data, is essential to find patterns and discover unknown, potentially useful, information. Discovery of patterns can also extend and improve upon already existing graphs. Some typical applications of graph analysis are data mining, node clustering, and classification.

A graph embedding technique aims to represent a graph or elements in the graph as elements in a vector space. Usually, unsupervised machine learning techniques are applied to generate embeddings. Ideally, the embeddings should preserve all the graph elements' properties and at the same time reduce the dimensionality of the representation of these properties.

Traditional graph analysis methods [12] are computationally expensive and problematic to scale. Graph embeddings intend to solve this problem and make possible the analysis of larger graphs. We can consider the embedding of a graph as the compression of the structural information in the graph into a few dimensions. Embedding the graph is essential to provide access to most machine learning systems. Few ML systems are designed with graphs in mind but operate on numerical data such as vectors.

The input of a graph embedding system is a graph, and the output is one or a set of low dimensional vectors. The vectors could denote either the entire graph, subgraphs or the individual nodes in the graph. It can also be extended to include the edges in the graph.

There are three main strategies for embedding graphs. *(i)* The first approach uses matrix factorization. These systems typically capture some relationship like the similarity of nodes in a matrix and factorize the matrix in order to find the individual node embeddings. *(ii)* The second approach is to use deep learning methods on the graph. This approach samples the graph using random walks, or it uses the entire graph as input before using a neural network to train the embeddings. *(iii)* The third approach is to directly optimize the probability of the existence of an edge (i.e., a

relationship between two nodes). These strategies have different advantages and use cases. Matrix factorization techniques can embed the entire graph with all its relationships. Using deep learning with random walks can be very scalable because one decides the number of and size of walks on the graph independent of the number of relationships and nodes in the graph. Edge optimizing makes a good fit for knowledge graph completion and graph repair. In this thesis, we will focus on deep learning using random walks.

There are two very different similarity measures that the embeddings can conserve. One measure of similarity is to regard that closely connected entities should have similar embeddings (homophily). The other measure of similarity is to regard that similar structures should have similar embeddings (structural equivalence). Matrix factorization approaches are unable to embed structural equivalence [25]. Random walk approaches embed a combination of the two. It is possible to define which will be most prominent by changing the walk generation strategy.

Random walks and the Skip-gram (or a similar) algorithm as a way to embed graphs was introduced by Perozzi [72] and is known as the DeepWalk algorithm. This algorithm has given rise to a family of systems using similar approaches. The node2vec expands on this system using, not only random walks along connected edges but introduces the notion of a node's neighborhood and second-order similarity. It uses two variables (p and q) to bias the walks either to focus on a node's nearby neighborhood or to perform deeper walks like the DeepWalk algorithm.

2.6.3 Embedding RDF data

There are several proposed algorithms for embedding RDF-data. The Neural Tensor Network (NTN) [93] applies a bilinear tensor product to predict the relationship between two entities. The idea is that any triple $\langle h, r, t\rangle \in D$ should receive a higher score than any triple not in the dataset. TransE [10] is a model where, for a triple $\langle h, r, t\rangle$, the relation r is seen as a function transforming the embedding of h to the embedding of t. So that $h + r \approx t$ whenever $(h, r, t) \in D$ where D is the dataset. Another model HolE [57] combines the logic of TransE with the tensor product using *circular correlations* of the embeddings to represent pairs of entities.

Many embeddings systems focus on the creation of word embedding from documents in natural language. Some of these systems have been used successfully to create embeddings from other sources (including graph embeddings). RDF data lacks the linear structure that natural language texts have. By nature, RDF data has a graph structure. To be able to embed RDF using such systems, we first need to create something that a linear system will understand.

Ristoski and Paulheim introduce Skip-gram to create embedding from RDF-data in the system RDF2Vec [80]. This system uses all entities that are instances of `owl:Thing` as a source for the walks, and it includes both the nodes and the edges in the walk document. Ristoski and Paulheim evaluate two general approaches. The first is creating graph walks. Starting

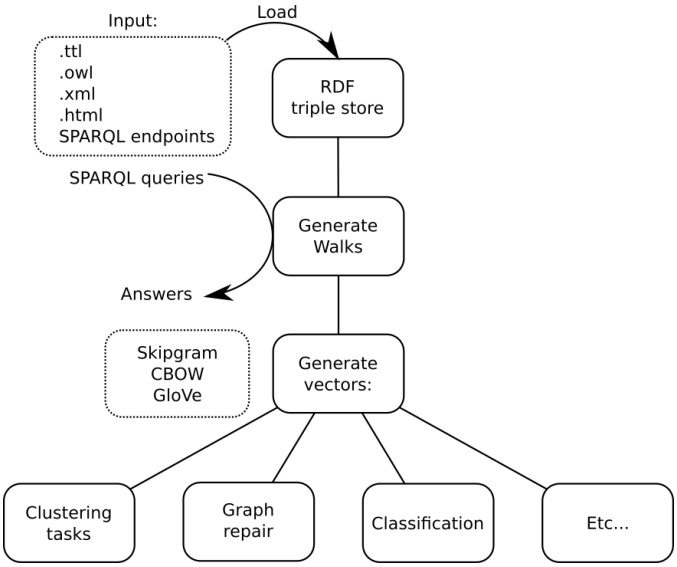

Figure 2.1: How can RDF data be embedded by systems designed for natural languages?

from all nodes, they create all possible walk patterns in the graph from that node until some threshold (i.e., max depth). This algorithm is a kind of a breadth-first search algorithm. This approach builds on the DeepWalk algorithm [72]. Ristoski and Paulheim also include, in the source code, a random walk approach [78]. The second approach proposed by the authors builds on Deep Graph Kernels [110]. This models graph subtrees instead of walks and measures the similarity of graphs by computing shared subtrees.

Another approach, proposed by Cochez *et al.* in [13], builds a co-occurrence matrix where, for each node, it is counted how often the other nodes appear close to this node (in the node's context). In this approach, it is not necessary to generate the actual walks, only the counts. This matrix can then be used to train the global vector for word representation presented in [71].

StarSpace provides yet another approach using supervised learning. In [108] it is proposed to learn a graph G consisting of (h, r, t) triples as selecting uniformly at random either a consists of a bag of features h and r while b consists of only t or that a consists of h only and b consists of r and t.

Agibetov and Samwald use StarSpace [3]. Here they are not embedding the relation r. They represent the edges by concatenating the embeddings of the nodes.

2.6.4 Embedding an ontology

We focus here on OWL 2 ontologies. OWL 2 ontologies can be considered a set of axioms. A document that the Skip-gram algorithm can use can be created directly by listing all the axioms. Smaili, Gao, and Hoehndorf apply this strategy for the Onto2Vec [90] and the OPA2Vec-system [91]. Since the "walks" will be very shallow, with only two nodes and one edge,

this approach is expected to conserve closely connected entities, and not expected to consider structural similarity.

We assume that some axioms, like for example disjointness will add noise to the embeddings. The axiom `ClassA disjointWith ClassB` would, for instance, make the embeddings of `ClassA` and `ClassB` more similar when one, in most cases, would like them to be more different.

Another approach is to consider the OWL-ontology a graph, and use it as input it to a system like RDF2Vec [80]. Still, it is preferable to emphasize concepts such as classes and hierarchical relationships between classes over other relationships. In the case of classes, it is possible to use not only the class description context and relationships but also include entities, instances of a class. As we will see, regarding the OWL-ontology as a graph, poses a few challenges.

2.6.5 Ontology projection and navigation graphs

As observed in several studies, including [94], [90], and [81], many OWL axioms are not directly representable as graphs. Consequently, the OWL ontology is not possible to navigate like a typical RDF-graph. OWL 2 allows many complex structures. Take for example axioms where the union or the intersection of two classes is expressed as the range of another union or intersection. The navigation from one such class to another and the visualizing of this as a graph is non-trivial. Two somewhat similar approaches have been proposed to solve this challenge. The first approach, developed by Soylu *et al.* in [94]. The other by Rodríguez-García and Hoehndorf in [81]. Both systems use OWL-reasoning and infer edges to project the OWL-ontology as a navigable graph. The systems that we will use in this thesis is the one by Soylu *et al.*

A navigational graph has predicates, constants and data types as nodes and the edges are relations between the nodes. Following the description in the paper [94], the edges between the nodes are constructed from the following object properties: *(i)* Domain and range axioms, *(ii)* Object property restrictions, *(iii)* Inverse properties, *(iv)* Role chains, *(v)* Top-down propagation, and *(vi)* Bottom-up propagation. And these datatype properties: *(i)* Domain and range axioms, *(ii)* Datatype restrictions, *(iii)* Top-down propagation, and *(iv)* Bottom-up propagation.

Chapter 3

Frameworks

This chapter introduces the approaches, systems and datasets that are relevant to this thesis. An exhaustive and complete description of the state of the art is outside the scope of this thesis. We start with the ontology alignment system LogMap in Section 3.1 before we move on to embedding tools for natural language in Section 3.2 on page 29. In Section 3.3 on page 36, we look at embedding systems for graphs and RDF-graphs. Section 3.4 on page 38 introduces a model for using pretrained word embeddings for ontology alignment. In Section 3.5 on page 40 we look at libraries and APIs for working with semantic technologies and machine learning. Then, in Section 3.6 on page 44 there is a description of the ontology projection tool used in the thesis before in the last section (Section 3.7 on page 46) there is an overview of the ontologies used in the experiments.

3.1 The ontology alignment system LogMap

3.1.1 How does it work?

LogMap 2 [36, 82] (LogMap) is primarily a tool for ontology alignment. It can, however, also be used for mapping debugging. The system can be used as a standalone command line tool or as a Java library. It accepts OWL 2 ontologies and outputs files containing the mappings between the input ontologies in different formats. LogMap is designed to be very scalable and handle large ontologies. It uses inversed indexes and partitions the ontologies into manageable parts to reduce complexity.

LogMap is capable of dealing with a wide variety of ontologies and the different variations of the LogMap-system show excellent performance on the different test tracks of the OAEI campagin [1, 2, 4].

The basic function of LogMap is illustrated by algorithm 1 on the following page (adapted from [36]).

The workflow of LogMap can be divided into major two parts. The first part is the creation of candidate mappings to maximize recall. The second part is the discarding of mapping to maximize precision.

Algorithm 1 LogMap

Input: $\mathcal{O}_1$, $\mathcal{O}_2$: input ontologies; **Interact:** Boolean value
Output: $\mathcal{M}$: mappings.

1: $\langle LI_1, LI_2 \rangle \leftarrow$ LexicalIndexes($\mathcal{O}_1, \mathcal{O}_2$)
2: $\mathcal{M}_? \leftarrow$ CandidateMappings(LI_1, LI_2)
3: $\langle O'_1, O'_2 \rangle \leftarrow$ Module($\mathcal{O}_1, \mathcal{O}_\in, \mathcal{M}$)
4: $\mathcal{M} \leftarrow$ ReliableMappings($\mathcal{M}_?$)
5: $\mathcal{M}_? \leftarrow \mathcal{M}_? \setminus \mathcal{M}$
6: $\langle \mathcal{P}'_1, \mathcal{P}'_2 \rangle \leftarrow$ PropEncoding($\mathcal{O}'_1, \mathcal{O}'_2$)
7: $\mathcal{M} \leftarrow$ Diagnosis($\mathcal{P}'_1, \mathcal{P}'_2, \mathcal{M}, \emptyset$)
8: $SI \leftarrow$ SemanticIndex($\mathcal{P}'_1, \mathcal{P}'_2, \mathcal{M}$)
9: $\mathcal{M}_? \leftarrow \mathcal{M}_? \setminus$ Discarded($LI_1, LI_2, SI, M_?$)
10: **if** $Interact = true$ **then**
11: $\quad \mathcal{M}_{user} \leftarrow InteractiveProcess(SI, M_?)$
12: $\quad \mathcal{M} \leftarrow$ Diagnosis($\mathcal{P}'_1, \mathcal{P}'_2, \mathcal{M} \cup \mathcal{M}_{user}, \mathcal{M}_{user}$)
13: **else**
14: $\quad \mathcal{M} \leftarrow$ Diagnosis($\mathcal{P}'_1, \mathcal{P}'_2, \mathcal{M} \cup \mathcal{M}_?, \mathcal{M}$)
15: return $\mathcal{M}$

Maximize recall

Ontology concept labels and URIs are stored using inverted indexes. Each URI is given a unique identifier which maintains the correspondence between the labels and the URIs. It uses techniques such as stemming to reduce the impact of variations of words. The indexes can be enhanced with similar concepts and synonyms by external resources like BioPortal [83], the SPECIALIST Lexicon [103] or WordNet [107].

LogMap then computes a large number of candidates mapping. It creates mappings for each concept with the same index labels using all combinations of all URIs reachable from these indexes. These mappings is the upper bound of the mappings created by the system. After the creation of these mappings, it will not add new mappings.

The third step of the LogMap algorithm is called module extraction. It divides the ontologies into smaller parts (modules) which group together concepts that naturally belong together. It is done to be able to perform reasoning tasks involving ontology parts and candidate mappings more efficiently. Because the computational complexity of such reasoning tasks depends heavily on the size of the ontologies at hand.

Maximize precision

The system will then identify mappings that are very likely to be valid based on string and structural similarity. A mapping is (typically) valid if the lexical information of the involved entities is very similar and the neighbours of the entities in one ontology also have mappings to neighbours of the entity in the other ontology. This set of mappings, which is considered very likely mappings is then diagnosed and "repaired," usually by discarding some

very few of these very likely mappings, using logical reasoning creating a set of "reliable" mappings. It then creates a "semantic index" to answer complex semantic queries faster. Using the inverted index and the semantic index, it identifies most of the remaining mapping-candidates as not mappings by checking that, if adding this concept, it makes some concept unsatisfiable. It also uses concepts such as word co-occurrence to revise similarity.

LogMap also allows human interaction to help decide difficult or ambiguous cases. It will then ask questions where the user can accept or reject mappings.

3.1.2 Using the ontology matching system LogMap

The LogMap framework can be used as a stand-alone matcher on command line. The program takes a number of arguments. An example of a run is:

```
$ java -jar logmap2_standalone.jar MATCHER file:/home/ontofile1.owl
  file:/home/ontofile2.owl /home/output true
```

where `MATCHER` means that it is to work as a matcher. `/home/ontofile1.owl` is the path to the first of the ontologies to be matched, `/home/ontofile2.owl` is the path of the second ontology, and `/home/output` a folder to put the resulting mapping. The true-argument says that HermiT [24] is used to check if the output produced by LogMap leads to unsatisfiable classes.

We compiled LogMap from the source code by cloning its GitHub (maven) project and using the command `mvn clean install`. The generated jar-file was included as a library in the maven project created for the experiments of this thesis. LogMap can then easily be used from a Java application as follows:

```
LogMap2_Matcher logMapMatcher = new LogMap2_Matcher(
    "file:" + firstOntologyFile,
    "file:" + secondOntologyFile,
    "/home/logmap_out/", true);
```

It will then perform the alignment between the first and second ontology files and output it to the path provided as the third argument. It is also possible to access the anchors and other mappings directly.

3.2 Neural language models

Neural language models are systems that use neural networks and embeddings in vector space to model natural language. It is central to many natural language processing tasks and it has a rather long history. Recently it has been possible to train quite complex models on a large corpus of data [51].

3.2.1 Training complexity

The training complexity of a model [51] is given by the equation

$$O = E \times T \times Q \tag{3.1}$$

Where E is the number of times the algorithm passes through the entire training set, T is the number of words in the training set and Q is the complexity of each training example, which changes from model to model.

3.2.2 Feedforward Neural Net Language Model (NNLM)

The Feedforward Neural Net Language Model [7] proposed by Bengio, Ducharme, and Vincent, is a simple neural network model for creating word embeddings that was quite popular. It introduces the concept of learning a word vectors in a distributed way. That way it was able to account for semantic similarities of the words. The model is similar to the CBOW-model (see Section 3.2.4). It has an input layer, projection layer with a shared mapping function, hidden layers, and output layers. The model uses a hyperbolic tangent (tanH) activation function on the hidden layer and the softmax function (see Section 3.2.3) on the output layer to produce positive probabilities from the input vectors.

The model is computationally complex between the projection and hidden layers and the softmax-function is relatively costly.

The complexity of each training examples for the NNLM-model is:

$$Q = N \times D + N \times D \times H + H \times V \tag{3.2}$$

Where N is the number of previous words encoded using 1-of-V encoding at the input layer, V is the size of the vocabulary, the dimensionality of the projection layer is $N \times D$ and H is the hidden layer size.

3.2.3 The softmax function

The purpose of the softmax activation function [51] is to normalize an input vector $\vec{v}$ with K dimensions in such a way that all dimensions will be in the interval $(0, 1)$ and the sum of all the dimensions is 1. This is done by, for all K dimensions of a vector $\vec{v} = [v_1 \dots v_k]$, calculating the exponential of the dimension divided by the sum of the exponential of all dimensions. More formally the softmax function for v_j is given by:

$$\sigma(v_j) = \frac{e^{v_j}}{\sum_{k=1}^{K} e^{v_k}} \tag{3.3}$$

3.2.4 The Continuous Bag-of-Words Model (CBOW)

The Continous Bag-of-Words model, described in [51], is inspired by the NNLM model. It got its name because the order of the word does not influence the projection. The input layer consists of the surrounding context, a window of words, C. Thus some words are before the target word and

some are from after the target word. In contrast to the NNLM-model there is no hidden layer.

The CBOW algorithm uses the words surrounding the target word (w_t) where c is the maximum distance from w_t to predict w_t. More formally, the objective of the model is to maximize the average log probability:

$$\frac{1}{T}\sum_{t=1}^{T}\log p(w_t \mid w_{t-c}\dots w_{t+c}) \tag{3.4}$$

To calculate the probability, $p(w_t \mid w_{t-c}\dots w_{t+c})$, one could do this by using a function f that maps pairs of context words and target words to real numbers. $f(context\_words, target\_word) \rightarrow \mathcal{R}^K$. This could for example be softmax (see Equation 3.3 on the preceding page) using the scalar product between the vector representing the context words ($\vec{v}_{w_c}$) and the vector representing the target word ($\vec{v}_{w_t}$) as input, as described in [50] and [9].

As the CBOW model does not have a hidden layer, the softmax function, with its linear complexity, becomes a major computational bottleneck of the model.

Two alternatives are used to approximate the operation of the softmax function. The Hierarchical Softmax and Negative Sampling.

The training complexity of each example using the CBOW model is given by:

$$Q = N \times D + D \times \log(V) \tag{3.5}$$

Where $N \times D$ is the size of the projection layer and V is the size of the vocabulary.

3.2.5 The continuous Skip-gram Model

The Continuous Skip-gram model (Skip-gram) [50, 51], is very similar to the CBOW-model, but unlike the CBOW-model, Skip-gram tries to predict the context words based on the current word instead of predicting the current word based on the context. Consider a series of words $w_0, w_1 \dots w_n$ and a window of words C, the Skip-gram model will try to maximize the average log probability of the following equation:

$$\frac{1}{T}\sum_{t=1}^{T}\sum_{-c\leq j\leq c, j\neq 0}\log p(w_{t+j} \mid w_t) \tag{3.6}$$

The complexity of each training example with Skip-gram is given by:

$$Q = C \times (D + D \times \log_2(V)) \tag{3.7}$$

Where C is the maximum distance of the words (the windows), D is the vector dimensionality, and V is the vocabulary size.

Skip-gram gives more weight to words close to the target word than more distant words because close words tend to be more related. As for the CBOW model, the softmax function becomes a major bottleneck and negative sampling or hierarchical softmax are used with this algorithm as well as more efficient alternatives.

3.2.6 Word2vec

Word2vec is the name of the system proposed by Mikolov *et al.* in [51]. The original implementation is written in the programming language *C* and allows the user to decide to use either CBOW or Skip-gram together with either negative sampling or hierarchical softmax to produce the word embeddings.

Using word2vec

There are several possible ways to use word2vec in applications. We have tested 3 of them and both the gensim library and deeplearning4j library are used extensively in the project.

The C-implementation. The original implementation of word2vec is a multithreaded C-implementation published by Google. It comes with some demo script and programs to explore the nearest neighbour and word analogies. The embeddings can be trained with 200 dimensions, using the Skip-gram model, window size 10 and output a binary model like this:

```
$ ./word2vec -train input_file.txt -output model.bin -binary 1
  -cbow 0 -size 200 -window 10
```

One can play with and assess the quality of the embeddings using the nearest neighbor using the distance and word analogy programs.

The gensim implementation. To test the gensim (python) implementation of word2vec, we followed a guide on the home page of the gensim library [23]. The following example program reads a document from disk, trains the word2vec model and writes the model.

```
import gensim

input_file = '/home/input.txt'
output_file = '/home/model.bin'

def read_input(input_file):
    """This method reads the input file format"""
    with open(input_file, 'r') as f:
        for i, line in enumerate(f):
            yield gensim.utils.simple_preprocess(line, deacc=False,
                min_len=2, max_len=15)


documents = list(read_input(input_file))

model = gensim.models.Word2Vec(documents, size=50, window=10,
    min_count=1, workers=12, iter=1, sg=1, hs=0, negative=25)
model.save(output_file)
```

We trained the model on a small fragment of Wikipedia (the first billion bytes, downloaded and preprocessed as explained in the `fastText` tutorial [106]). Asking the model what is closest to *Norway* it returns

(Sweden, 0.875...). Asking for the word closest to *thesis* it returns (dissertation, 0.795...). We tried to ask the model different questions and the answers it gave were very good in most of the cases. It is also possible to combine word vectors to give positive and negative concepts and it will return the closest word to the combined vector.

The deeplearning4j Java library. The following code trains a word2vec model using deeplearning4j. It looks up the nearest words to a token and find the similarity between two tokens before it writes the trained model to disk.

```
SentenceIterator iter = new BasicLineIterator(inputFilePath);
TokenizerFactory t = new DefaultTokenizerFactory();

Word2vec model = new Word2Vec.Builder()
              .minWordFrequency(5)
              .iterations(1)
              .layerSize(100)
              .seed(42).windowSize(5)
              .iterate(iter).tokenizerFactory(t).build();
model.fit();
Collection<String> closestToConference =
    model.wordsNearest("http://cmt#Conference", 10);
double sim = model.similarity("http://cmt#Conference",
    "http://ekaw#Conference");
WordVectorSerializer.writeWord2VecModel(model, outputFilePath);
```

3.2.7 The `fastText` model

In the word2vec system, the smallest unit is a word and one word is represented by one vector. Each word's vector is trained independently of any other word's vector, and there is no sharing of information. Mikolov *et al.* demonstrate that it is possible to do vector addition of two word vectors to combine the meaning of the words [50]. `FastText` [9] elaborates on this concept to include information contained in the structure of words. This is done by training the Skip-gram model (Subsection 3.2.5 on page 31) or the CBOW model (Section 3.2.4 on page 30) on character *n*-grams instead of words only. It represents a word as the sum of the vectors of the word's character *n*-grams. That way, the word vectors already take into account common sub-structures of words. Bojanowski *et al.* remark that while the word2vec model is unable to represent unseen words and needs much training data to get good word vectors, the `fastText` model generates good word vectors even on small training datasets and work well even on unseen words.

`FastText` does not only compute word vectors, but it can also create embeddings for classification tasks using a supervised model [38].

Using `fastText`

`FastText` can also be included in application in several ways. There is a C++ -implementation and an implementation in the gensim python library.

Deeplearning4J does, however, not include an implementation of `fastText`.

The C++ implementation. We first cloned the repository [45] and built the source using `make`. We trained the model with the same Wikipedia fragment as above. This method is described in the `fastText` documentation found on [45]. For example to train a `fastText` model using Skip-gram, character n-grams with sizes between 3 and 6, window size 10, 5 epochs, negative sampling with sampling size 5 and 200-dimensional embeddings and 8 threads. It can be done using the command:

```
$ ./fasttext skipgram -input input_file.txt -output model.bin
  -minn 3 -maxn 6 -ws 10 -epoch 5 -neg 5 -loss ns -thread 8 -dim 200
```

Instead of coming with additional programs to play with the embeddings and asses the quality it is bundled in the same program and can be used by adding parameters. It is possible to query the model for the nearest neighbours:

```
$ ./fasttext nn model.bin
```

This command starts a prompt where one can query for nearest neighbors. It is also possible to print the word vectors of words or work with analogies such as if we have the relation Oslo - Norway, what is then related to Sweden? `FastText` can also do text classification, for which it is a highly competitive system[108]. This can be done using supervised training where the training data is labeled.

The gensim implementation. Using the gensim implementation to work with `fastText` is analogous to using the word2vec model. The following snippet trains and saves a model using Skip-gram, 200 dimensional vectors, 8 threads, character *n*-grams between 3 and 6, a window size of 10 doing 5 epochs and with 25 negative samples.

```
import gensim
import logging


input_file = '/home/input.txt'
output_file = '/home/output.bin'

def read_input(input_file):
    """This method reads the input file format"""
    with open(input_file, 'r') as f:
        for i, line in enumerate(f):
            yield gensim.utils.simple_preprocess(line)

documents = list(read_input(input_file))
model = gensim.models.FastText(sg=1, hs=0, size=200, workers=8,
    word_ngrams=1, min_n=3, max_n=6, sentences=documents,
    window=10, min_count=1, iter=5, negative=25)
model.save(output_file)
```

The resulting vectors are similar to the vectors trained with word2vec.

3.2.8 The StarSpace model

Another recent example of a neural embedding model is StarSpace, which, in contrast to other models, aims to be a general-purpose neural embedding model. From the paper:

> *[StarSpace] can solve a wide variety of problems: labeling tasks such as text classification, ranking tasks such as information retrieval/web search, collaborative filtering-based or content-based recommendation, embedding of multi-relational graphs and learning word, sentence or document level embeddings. [108]*

StarSpace considers each entity as a collection of features (bag-of-features), and learns an embedding for each of the features. Depending on the task, the embedding of an entity can be defined as the sum of the embeddings of its features or an optimization. It also allows the user to choose the loss function and the similarity measure. Where L is the loss function, (a,b) is a pair of entities in the training data, (a,b^-) is a negative example (b is not associated with a in the training data), and the $sim()$ function is the similarity function, the objective of StarSpace is to optimize this function (using stochastic gradient descent):

$$\sum_{\substack{(a,b)\in E^+ \\ b^-\in E^-}} L^{batch}(sim(a,b), sim(a,b_1^-), \ldots, sim(a,b_k^-)) \tag{3.8}$$

StarSpace can in one aspect be considered to take the concepts explored in `fastText` one step further. Where `fastText` considers a word as a set of n-grams. StarSpace considers an entity e as a bag of features. This bag-of-features could be, for a word, n-grams. For a document it could be sentences or words. StarSpace is very flexible and contains many different algorithms for embeddings, and rather than an extension of `fastText` it is more a collection of tools and algorithms.

Using StarSpace as a neural language model

StarSpace was tested, along with the other models for creating word embeddings, but our experience is that the embeddings produced are of inferior quality and the time to train is longer. It could be that one can get better embeddings and faster training by experimenting more with the parameters. We did not find any official python or Java library, which makes it more difficult to include as the part of a workflow.

To use StarSpace we cloned the StarSpace GitHub-repository[95]. Following the instructions, we installed the boost-library using apt-get (`sudo apt-get install libboost-all-dev`). Then we compiled the source successfully using `make`. According to the documentation we can now train the model using a text-file. Unsupervised word embeddings are trained using trainMode 5. StarSpace allows for much flexibility using the parameters. Training a model can be done with for example this command:

```
$ ./starspace train -trainFile out.txt -model starspace.model
  -dim 50 -loss softmax -thread 10 -minCount 1 -ngrams 2
  -trainMode 5 -epoch 1 -maxNegSamples 25 -lr 0.01 -ws 5
```

StarSpace comes with a tool to inspect the nearest neighbours. This is built separately with `make query_nn`. It is possible to use the model created with trainMode=5 to find the nearest neighbours the same way as with `fastText` and word2vec.

3.3 Graph embedding and RDF-embedding

3.3.1 DeepWalk

The first approach to employ the Skip-gram model to graph embedding was the DeepWalk approach [72]. Where G is the graph to be embedded, $|E|$ is the number of edges, $|N|$ the number of nodes, w the number of walks per node and $|W|$ the total number of walks. The DeepWalk algorithm is described in pseudocode in Algorithm 2.

Algorithm 2 DeepWalk

Input: w - the number of walks per node, F input_file
Output: D - a document of walks

$G \leftarrow readGraph(F)$
$|E| \leftarrow numberOfEdges$
$|N| \leftarrow numberOfNodes$
$|W| \leftarrow w \times walkLength$
$walks = []$
$nodes = listAllNodes$
for $i \rightarrow w$ **do**
    $shuffle(nodes)$
    for all $node \in nodes$ **do**
        $walks.append(randomWalks(node))$

Algorithm 3 randomWalks

Input: w the number of walks per node, α restart probability
Output: z

$paths = []$
while $len(path) < w$ **do**
    $currentNode = path[-1]$
    if $len(currenthNode.adjacentNodes) > 0$ **then**
        if $randomNumber > \alpha$ **then**
            restartWalk
        else
            $paths.append(random(currentNode.adjacentNodes))$
    else
        $break$

3.3.2 Node2vec

Node2vec [26] extends the DeepWalk by redefining the notion of a node's neighborhood. The neighborhood of a node could be viewed as following a breadth-first search or a depth-first search or something in between. If the next node in the walk is following a breadth-first sampling, it tends to create embeddings that makes structurally similarily nodes similar. If it is following a depth-first strategy, it would create embeddings where interconnected nodes are more similar to each other. Node2vec has two new parameters (p and q) that let the user bias the walks to be more similar to a depth-first search or breadth-first search. It can focus on the fine details of the close neighborhood which will make the embeddings of strongly interconnected nodes be more similar. Alternatively, it can favor deep walks which will tend to make structurally similar nodes more similar. Node2vec also uses the Skip-gram model to create embeddings from the walks.

3.3.3 StarSpace as a graph embedding model

StarSpace can embed multiple types of data. Including multi-relational knowledge graphs [108]. It can be used to predict labels for text and to embed knowledge graphs. This can be done by considering a graph a collection of classifications. Using *subject* to *property*, *object* and from *subject*, *property* to *object* statements. Graph embeddings without explicitly embedding the edge could be created using (*subject*, *object*)-pairs [3].

StarSpace has 6 trainModes (0-5) with different functionalities. It will associate left and right-hand side. So it could for example associate the nodes without the relation or it could be a concatenation of the subject and property with the object. We trained with some 5,000 triples taken from the UMLS Metathesaurus [102] where each triple was duplicated and alternating if the predicate or the object was to be the "label" [95] with trainMode=0. It was possible to answer queries on the form *virus affects ...* or *... produces amino_acid_peptide_or_protein* and the answers made some sense. If we were to ask for some subject it did not have in its vocabulary (for example *truck affects*) it would answer the same as for the predicate only.

3.3.4 RDF2Vec

RDF2Vec [80] was introduced in Subsection 2.6.3 on page 24. The work on RDF2Vec demonstrates that neural language models can be used to embed RDF-graphs by considering entities and relations between entities instead of word sequences. It was shown that such entity representations outperform existing techniques for representing RDF graphs.

We downloaded and built the code for RDF2Vec. It comes without any documentation and it was quite challenging to make it work. It is implemented in Java and has some python to glue the system together. RDF2Vec comes with several important independent files.

The first file is `WalkGenerator.java`, which contain the code to generate the actual walks. It takes as a input the path to a Jena TDB triple store

(introduced in Section 3.5.6 on page 43), the path to the output file and a number of integer arguments deciding how many threads are to be used, number of paths to be created and the depth of the paths.

The WalkGenerator creates a file where all the paths generated are on the format $v_r \rightarrow e_{1i} \rightarrow v_{1i}$. This file has to be processed to fit the input of the neural language system that is to be used. To be able to use word2vec, all the $->$ are substituted to spaces, only the last one in all path is substituted with a newline character. This is the purpose of the `PathCleaner.java`-file. That way each element of the path will be considered by word2vec a word and each path a sentence.

We were not able to run it and test it as a standalone system. Export the RDF2Vec as a jar-library and importing it using the path to a maven project in Eclipse [31], however worked without complications. Depending on which of the implementations that are used, it can output a gzip-file or a plain text file. We then tried to feed this into word2vec (using the gensim library) and it created the embeddings as expected.

The RDF2Vec can be used as a library in Java like this.

```
WalkGeneratorRand walkGenerator = new WalkGeneratorRand();
walkGenerator.generateWalks(REPO_LOCATION, TEMP_FILE,
    numberOfWalks, walkDepth,
   numberOfThreads, offset, limit);
PathCleaner.cleanPaths(TEMP_FILE, CLEAN);
gunzipFile(CLEAN);
```

3.4 Pretrained models

This thesis investigates the usefulness of applying vector space embeddings to the task of discovering similarities and help finding correspondences among ontologies. One obvious way to include the use of word embeddings in ontology alignment systems is to use pretrained vector models to match on a word to word basis. There are already several articles discussing the value of using such models [58, 96]. The two articles have rather different approaches to solving the ontology alignment problem, but both articles show that word embeddings is a valuable extension to the "ontology alignment toolbox". The implementation created in this project aims at reproducing the results reported in the papers and to serve as a baseline to compare with other ontology embedding strategies.

This pretrained model approach is useful for comparing lexical information associated with the entities. The actual URIs can also be compared if it contains lexical information. To be able to compare the URIs, it must first be parsed into one or multiple words. This is achieved using regular expressions. Comments and labels attached to the URIs are also used as a way of finding correspondences.

The system uses OWL API to read the two ontologies and extract the information needed from them. We focus on word embeddings at the terminological level of the ontology. This implementation goes through

all the classes from the first ontology O_1 and checks for similarity for all classes in the second ontology O_2 first for URI (URI), then labels (L), then the comments (C).

All the lexical information (URI, L, C) must be preprocessed to be able to find word-embeddings. URIs are often multiple words written using the camelCase, with an underscore (_) or hyphen (-) to separate words. Regex is used to split these up to and do case normalization to form separate lowercase words. Any punctuations numbers and multiple consecutive spaces are also removed. For the labels and comments, camelCase is not normal so it just converts all to lowercase. Hyphens and underscores are removed and all blanks are converted to a single space. Numbers are left untouched.

It finds the similarity measure that is highest between URI, L and C of the candidate entities $e1_{candidate}$ and $e2_{candidate}$ from O_1 and O_2, respectively. When it has compared all the entities in O_2 to the one in O_1 the similarity score is compared against an equality threshold (e.g., 0.8). If the similarity is above this threshold, the mapping ($\langle e1_{candidate}, e2_{candidate}, confidence \rangle$) is added to the alignment. It is assumed that any entity has at most one corresponding entity in the other ontology. Therefore, only the best match for the O_2-class is considered.

As a measure for equality, the algorithm uses cosine similarity to determine the similarity between two word vectors. If any word is not found in the pretrained model, it will be unable to find any correspondence. Consequently, in a real-world system, it is necessary to include other ways to determine string similarity in addition to the word embeddings. When dealing with technical vocabulary, it can be particularly challenging to find a text corpus that is big enough to train the model.

The algorithm for element-level matching is an adaption from [96] and it can be represented using the pseudocode in Algorithm 4.

Algorithm 4 Element level matching using pretrained model

$Entities_1 = C1 \cup OP_1 \cup DP_1$
$Entities_2 = C2 \cup OP_2 \cup DP_2$
for all $e_1 \in Entities_1$ **do**
    $maxSim = 0$
    $candidate = null$
    for all $e_2 \in Entities_2$ **do**
        $sim = max\{Sim(name_1, name_2),$
        $Sim(label_1, label_2),$
        $Sim(comment_1, comment_2)\}$
        if $sim > maxSim$ **then**
            $maxSim = sim$
            $candidate = e_2$
    $alignment \leftarrow alignment + (e_1, candidate, maxSim)$
return $alignment$

3.5 Triplestores and APIs

3.5.1 Jena

Jena [5] is a Java-based framework for building semantic web applications. It comes with an API for handling RDF, RDFS and OWL 1.1. Jena supports different RDF-store models, such as in memory and persistent on disk storage, and it has support for SPARQL-queries. It can read and write a variety of different formats such as turtle, RDF-XML and N-triples. Jena allows the user to use both built-in and custom reasoners.

Jena has one important drawback for our purposes. It still does not support OWL 2. It does, however, support reasoning and SPARQL-queries over RDF-graphs. It also comes with services for data storage such as the TDB-triple store (Subsection 3.5.6 on page 43) and the Jena Fuseki web-server.

The following code example creates an in-memory ontology model on which it performs reasoning using a built-in reasoner. It then performs a SPARQL-query to find all instances of type `owl:Class` in the dataset. The URI is printed to the console.

```
OntModel ontModel =
    ModelFactory.createOntologyModel(OntModelSpec.OWL_DL_MEM);
ontModel.read(inputFile, "TTL");
reasoner = ReasonerRegistry.getOWLReasoner();
Model infModel = ModelFactory.createInfModel(reasoner, ontModel);

String queryString = "SELECT ?s WHERE { ?s "
    + " <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> "
    + " <http://www.w3.org/2002/07/owl#Class> }"
Query query = QueryFactory.create(queryString);
QueryExecution qe = QueryExecutionFactory.create(query, infModel);
ResultSet results = qe.execSelect();
while (results.hasNext()) {
    QuerySolution result = results.next();
    System.out.println(result.get("s").toString());
}
qe.close();
```

3.5.2 Eclipse Deeplearning4j

Eclipse Deeplearning4j (DL4J) [98] is a deep learning library, and an open source project, written for use on the Java Virtual Machine (JVM) to work with Java and Scala and fronted by Skymind [89]. It is designed to work with Hadoop and able to work with both CPUs and GPUs. It requires `Apache Zeppelin` to be installed on the system.

DL4J's first element is the DataVec library which is used to prepare and load data. This is done by using the `RecordReader`-interface and the `RecordReaderDataSetIterator`. The `DataIterator` can then be passed as an argument to the fit()-method on the model.

The second element in ND4J is a tensor library to work with linear algebra, tensors, and vectors. A tensor is a multidimensional array. It includes a number (rank) where a tensor with rank 0 is a scalar, a tensor with rank 1 is a vector, a tensor with rank 2 is a matrix, and tensors with rank 3 and above are referred to as tensors [99].

A tensor in DL4J is represented as the NDArray object. It differs from a common multidimensional Java-array in some aspects. It stores numbers as a flat array contiguously in memory and it is stored off-heap, outside the JVM which gives improved performance. It is possible to change the order of how the array is stored in memory (row wise or column wise) which can give improved performance in some applications.

Basic operations of ND4J are creation and algebraic operations such as addition and multiplication on matrices. This code for example:

```
INDArray nd = Nd4j.create(new float[]{1,2,3,4},new int[]{2,2});
nd.add(4);
nd.mul(2);
```

will create the 2- matrix:

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$$

and perform matrix addition and multiplications which will yield the following 2 matrices in succession:

$$\begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix} \begin{bmatrix} 10 & 12 \\ 14 & 16 \end{bmatrix}$$

Several matrix operations are available such adding, multiplying and subtracting matrices and transpose, adding and deleting rows and dimensions. An example of cosine similarity is given below:

```
import static org.nd4j.linalg.ops.transforms.Transforms.*;

public double calculateCosine(INDArray d1, INDArray d2) {
    return cosineSim(d1, d2);
}
```

For the subject of this thesis DL4J it is interesting primarily because it includes support for the word2vec model. In DL4J the only implementation of the word2vec model is the Skip-gram model (Subsection 3.2.5 on page 31). It is not possible to use the CBOW model. A code snippet showing how one could train a word2vec-model using DL4J is given in the word2vec subsection (3.2.6 on page 32).

3.5.3 Gensim

Gensim [79] is an open source library for creating unsupervised word embeddings from text and working with these word embeddings in python. It is implemented in Python and Cython. The library is implemented to be

very robust and scalable. It comes with support for many different models such as word2vec, `fastText`, Doc2Vec, and Latent Semantic Indexing. In addition it provides many tools for working with the models.

Gensim is compatible with, and typically used in combination with NumPy [104] which is a python library for mathematical computations that aims to provide a framework for efficient mathematical operations on vectors and matrices. This combination lets the user easily perform custom calculations on the vectors.

A code example using gensim to train a word2vec model is given in Subsection 3.2.6 on page 32 and example code using gensim to train a `fastText` model is given in Subsection 3.2.7 on page 33.

3.5.4 OWL API

The OWL API [30] is a Java API for working with OWL ontologies. It is built considering the specification of OWL 2. The OWL API allows for manipulating OWL 2 structures and OWL 2 reasoning. It also validates OWL 2 profiles such as QL, EL, and RL.

In the OWL API, the ontology is viewed as a set of axioms and annotations. Unlike Jena, the axioms and classes are not at the level of RDF triples. The OWL API classifies the entities into classes (C), object properties (OP), data properties (DP) and instances. These sets are assumed to be disjoint.

In Jena one would define OWL classes as follows (assuming that 'model' is a OntModel object):

```
OntClass a = model.createClass(URI + "a");
OntClass b = model.createClass(URI + "b");
a.disjointWith(b);
```

In OWL API, one would work on the level of axioms and would achieve the same by doing something like this, assuming that 'df' is a DataFactory object and 'o' is an OWLOntology object:

```
OWLClass a = df.getOWLClass(URI + "a");
OWLClass b = df.getOWLClass(URI + "b");
OWLDisjointClassesAxiom disj = df.getOWLDisjointClassesAxiom(a,
    b);
o.add(disj);
```

A short tutorial and basic instructions on how to work with the OWL API are found in [46].

3.5.5 RDFox

RDFox [54] is a RDF triplestore reasoner developed by the University of Oxford which is highly scalable and efficient. It is written in C++ but comes with APIs for both Java and Python. The model is in-memory for performance, so the scalability is bound by RAM. Still, it is quite economical

when it comes to memory usage per triple. Both reasoning and SPARQL-querying are parallelized. RDFox is used for the creation of the ontology projection.

The RDFox store can be used in Java to store a model, read from a file, and to query the model using a SPARQL-query as follows:

```
DataStore store = null;
try {
   store = new DataStore(DataStore.StoreType.ParallelSimpleNN);
   store.setNumberOfThreads(numberOfThreads);
   store.importFiles(new File[] { new File(inputFile) });

   TupleIterator tupleIterator = null;
   String queryString = "SELECT ?x WHERE { ?x ?y ?z }";
   try {
      tupleIterator = store.compileQuery(queryString);
      if (tupIt.getArity() == 0) {
         return null;
      }
      for (long multiplicity = tupleIterator.open(); multiplicity >
          0;
      multiplicity = tupleIterator.advance()) {
         Resource resource = tupleIterator.getResource(0);
         String resourceString = resource.toString();
         System.out.println(resourceString);
      }
      finally {
         if (tupleIterator != null) {
            tupleIterator.dispose();
         }
      }
   }
} finally {
   if (store != null) {
      store.dispose()
}
```

3.5.6 Jena TDB

Jena TDB is Jena's RDF store. There exists a newer generation TDB2, but the original TDB is the one that is used in RDF2Vec. It is described as having high performance and supports RDF storage and query. It uses a folder on the hard drive to store the RDFs and allows normal Jena transactions to work with the store. Jena further comes with some scripts to load triples into a TDB triple store from a file without using Jena transactions `tdbloader` and `tdbloader2`. RDF2Vec uses this library to hold the data for the generation of the walks, which are generated using SPARQL-queries.

The store can be populated from command line using the scripts like this:

```
$ tdbloader --loc /home/test/repo /home/input.ttl
```

Next there is an example of how the model could be created and populated with data from a turtle file in Java using the Jena API. It is possible to use the model without using the transaction constructs, begin, commit, and end, but it is recommended.

```
Dataset dataset = TDBFactory.createDataset(REPO_LOCATION);
dataset.begin(ReadWrite.WRITE);
model = dataset.getDefaultModel();
model.read(inputFile, "TURTLE");
dataset.commit();
dataset.end();
dataset.close();
```

3.6 Ontology projection tools

The *Optique* team developed an ontology projection tool [94]. The implementation is available at [19]. Some of the most important classes, with methods and variables are illustrated in Figure 3.1 on the next page.

This ontology projection tool is implemented in Java and uses OWL API to work with the ontology and to create a navigable graph of an ontology. The graph is stored using RDF4J (Sesame). The ontology projection tool uses HermiT [24] or ELK [40] together with RDFox [54] as reasoning engines. Projecting the ontology to a graph is important for applications that need to navigate an ontology like a graph because of the challenges discussed in Subsection 2.6.5 on page 26. With the projection, the ontology can be navigated like a normal RDF-graph.

Let O be the ontology that we are to project, C_1, C_2, ... C_n are classes in O, R_o is an object property in O, and R_o^- is the inverse property of R_o. Let R_d be a datatype property in O, dt a datatype. x and y are numerical values, *restriction* is one of the restrictions used in OWL-ontologies like for example `some`, `only`, `min`, `max`, `exactly` restrictions. l is a literal and $l_1, l_2 \ldots l_n$ are enumerations.

To create the navigation graph G of the ontology O, all the classes (C) and datatypes (dt) in O are projected to nodes in G. The edges are added to the graph according to the axioms of the ontology. The rules are summarized in Table 3.1 on the next page. Most of the conditions are axioms in the OWL 2-ontology, but some also include checking if G contains an edge.

Domain and range expression are translated to edges. If it contains a class expression using union or intersection an edge is added between all named, pertaining classes. Object property restrictions are added to edges as well as role chains. Inverse property edges are added and edges are added through top-down and bottom-up propagation through the `subClassOf` and `subPropertyOf` axioms. The same is done for the data properties including property restrictions, domain and range expressions and top-down and bottom-up propagation.

Using eclipse, we exported a jar-file from the implementation available at [19]. The file was added to the maven project, and using this library, it is possible to create, access, and save the navigational graph.

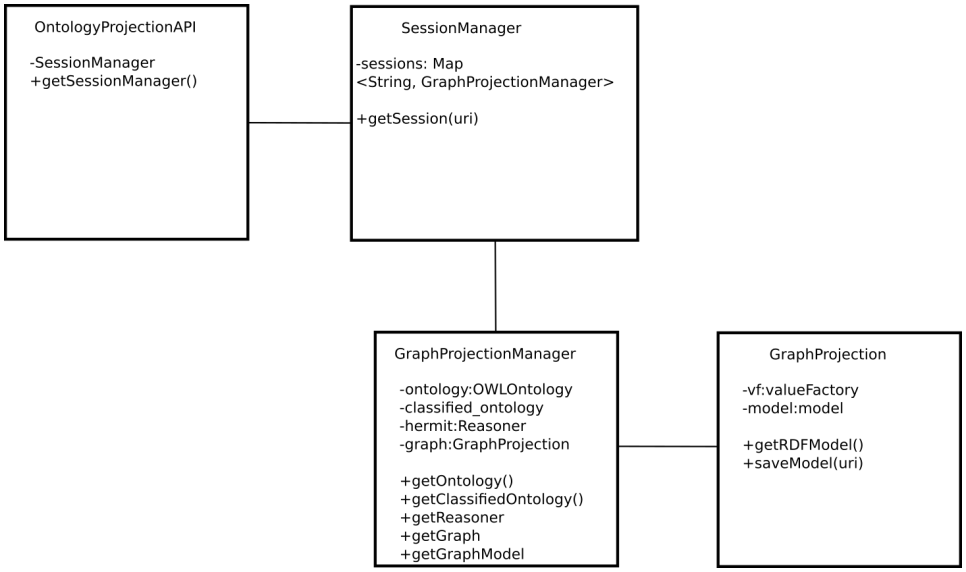

Figure 3.1: Core components of the ontology projection from Optique

Condition_1	Condition_2	Edge in G
$A\ SubClassOf:\ R_o\ restriction\ B$		$A \xrightarrow{R_o} B$
$R_o\ restriction\ B\ SubClassOf:\ A$		$A \xrightarrow{R_o} B$
$R_o\ Domain:\ A$	$R_o\ Range:\ B$	$A \xrightarrow{R_o} B$
$A\ SubClassOf:\ R_o\ value\ b$	$b\ type\ B$	$A \xrightarrow{R_o} B$
$R_o\ InverseOf:\ R_o^-$	G includes $B \xrightarrow{R_o^-} A$	$A \xrightarrow{R_o} B$
$A\ SubClassOf:\ A_{sup}$	G includes $A_{sup} \xrightarrow{R_o} B$	$A \xrightarrow{R_o} B$
$A_{sub}\ SubClassOf:\ A$	G includes $A_{sub} \xrightarrow{R_o} B$	$A \xrightarrow{R_o} B$
$A\ SubClassOf:\ R_d\ restriction\ dt$		$A \xrightarrow{R_d} dt$
$R_d\ Domain:\ A$	$R_d\ Range:\ dt$	$A \xrightarrow{R_d} dt$
$A\ SubClassOf:\ R_d\ value\ l$	$l\ type\ dt$	$A \xrightarrow{R_d} dt$
$A\ SubClassOf:\ A_{sup}$	G includes $A_{sup} \xrightarrow{R_d} dt$	$A \xrightarrow{R_d} dt$
$A_{sub}\ SubClassOf:\ A$	G includes $A_{sub} \xrightarrow{R_d} B$	$A\ R_d\ dt$
$A\ SubClassOf:\ R_d\ restriction\{l_1 \cdots l_n\}$		$A \xrightarrow{R_d} l_i$
$R_d\ Domain:\ A$	$R_d\ Range:\ \{l_1 \cdots l_n\}$	$A \xrightarrow{R_d} l_i$
$A\ SubClassOf:\ R_d\ value\ l_i$		$A \xrightarrow{R_d} l_i$
$B\ SubClassOf:\ A$		$A \xrightarrow{broader} B$

Table 3.1: The rules used to construct the navigational graph from the OWL 2 ontology

3.7 Datasets

Here, we describe some of the datasets that was used to test the systems. All the ontologies are from the OAEI tracks, and the OWLViz view tool in the *protegé ontology editor* was used to create the class hierarchies [52, 73].

3.7.1 Ekaw

The ekaw ontology [61] is one of several ontologies describing conferences. This ontology has 74 classes, 33 object properties and no datatype properties. It uses the SHIN DL expressiveness. The class hierarchy of the ekaw ontology is shown in figure A.1 on page 112.

3.7.2 Cmt

The cmt is another ontology in the OAEI's conference track. It has 36 classes, 10 datatype properties and 49 object properties. Its DL expressiveness is ALCIN. The class hierarchy of cmt is shown in A.2 on page 114.

3.7.3 The anatomy track

The OAEI anatomy track [61] consist of two ontologies; mouse and human. It is the task of finding the alignment between the mouse anatomy ontology with 2744 classes and 3 object properties and the human anatomy ontology with 3304 classes and 2 object properties. The human anatomy ontology is a fraction of the NCI Thesaurus [53] and the mouse ontology is the Adult Mouse Anatomy [29]. In these ontologies, the URIs contain no lexical information. The lexical information is found in the `rdfs:label` field, and can be quite technical.

3.7.4 The largebio track

The OAEI largebio track [61] consist of three real-world ontologies from the biomedical domain. The Nathional Cancer Institute Thesaurus (NCI) ontology [53], the SNOMED CT ontology [92] and the Foundational Model of Anatomy (FMA) ontology [21]. Several matching tasks can be performed on this track:

- FMA - NCI small fragments (3,696 and 6,488 classes, respectively)
- FMA - NCI whole ontologies (78,989 and 66,724 classes, respectively)
- FMA - SNOMED small fragments (10,157 and 13,412 classes, respectively)
- FMA - SNOMED large fragments (78,989 and 122,464 classes, respectively)
- SNOMED - NCI small fragments (51,128 and 23,958 classes, respectively)

- SNOMED - NCI large fragments (55,724 and 122,464 classes, respectively)

The reference alignments on the largebio track are created by using the UMLS [37] and in many instances contain multiple possible mappings that could lead to unsatisfiable classes. These are, however, marked with a ?-mark in the reference [59].

Part II

The project

Chapter 4

Embedding OWL 2 Ontologies with OWL2Vec

There are two fundamentally different approaches for the use of vector space embeddings for ontology alignment, ontology analysis, and ontology similarity. The first approach is to use pretrained word embeddings as described in Section 3.4 on page 38.

These embeddings are usually trained on a large general corpus of text such as Wikipedia, but can also be trained on a corpus from a specific domain. Another approach is to embed the semantic structure of the ontology itself. In this approach, the embeddings are not trained beforehand but instead created as part of a workflow on the ontology or ontologies at hand.

We have explored several strategies to create semantic embeddings from OWL 2 ontologies. First, we tested RDF2Vec on the projection of an ontology. Secondly, we implemented a system that simulates RDF2Vec, but with weighted edges (**OWL2Vec**). We experimented with several strategies to improve this system, and as a result, we implemented several separate, specific strategy systems. Finally, we implemented a walks generation strategy that works similar to node2vec (**OWL2Vec+**). This implementation also incorporated the strategies learned from many of the former systems and can "simulate" some of their behaviors. Although not entirely, as the walks generation strategy is different.

Besides, we have explored different systems for the generation of the embeddings from the walks. Likewise, for the task of ontology alignment, we have examined some strategies to improve recall. The components we have explored for the generation of the embeddings are illustrated in Figure 4.1 on the next page. The ontology alignment strategies can be seen in Figure 4.2 on the following page. To help the user identify the names of the walk generation systems among strategies and descriptions of systems, the names of all walk generation systems are printed in **boldface**.

This chapter begins by discussing the limitations of using RDF2Vec to embed ontologies. Section 4.2 on page 53 introduces **OWL2Vec**, a system implementing different strategies to improve RDF2Vec for ontology embedding. In Section 4.3 on page 54, we describe **OWL2Vec+**, a different

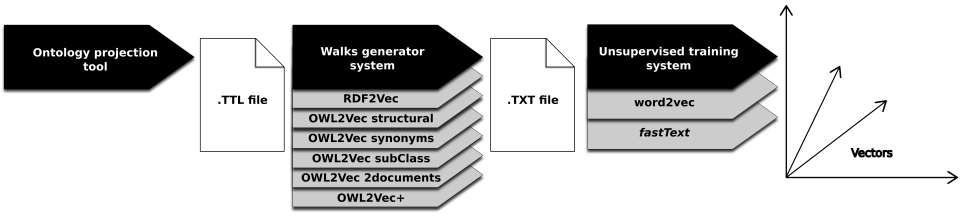

Figure 4.1: The components of the creation of the embeddings

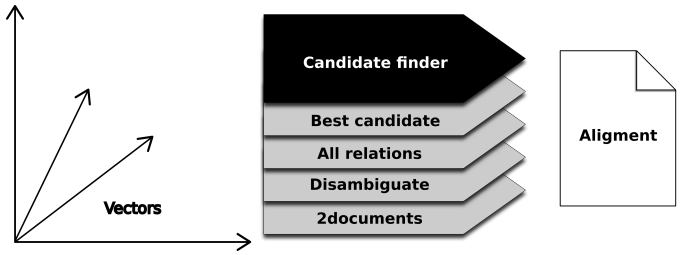

Figure 4.2: The alignment systems

system for embedding OWL 2 ontologies based on node2vec. Section 4.4 on page 61 describes approaches and strategies we employ to use semantic embeddings for ontology alignment. The next section (Section 4.5 on page 63) discusses the role of properties in embeddings, while Section 4.6 on page 64 evaluates how labels and synonyms can be exploited to improve the embeddings. Section 4.7 on page 65 is dedicated to the evaluation of the many parameters that can be adjusted to tweak the system.

4.1 Limitations of RDF2Vec

RDF2Vec is a system designed to create embeddings from RDF graphs [80], and it shows excellent performance for a variety of tasks such as data mining. It works by first selecting all entities in the graph, then creating walks starting from each of the entities. For our purposes, however, it is necessary to change this. By using the following SPARQL-query we select all classes instead of all entities.

```
SELECT DISTINCT ?s WHERE { ?s
    <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
<http://www.w3.org/2002/07/owl#Class> . }
```

The RDF2Vec is not able to walk an OWL 2 ontology as is because of the challenges discussed in Subsection 2.6.5 on page 26. The ontology must first be projected to an RDF-graph. Only then, and by using the result of the SPARQL-query above, is it possible to walk the OWL ontology. This strategy is, however, insufficient for our purposes because it does not take into account the importance of common ontology concepts such as subclasses and disjointness. For RDF2Vec a datatype property is as important as a `subClassOf`-relation. Using these walks and training embeddings with a system such as word2vec could return subclasses as very similar to the

superclass, and two distinct classes appear close to each other because both have a subClassOf-relationship to some class. Even the `disjointClasses`-axiom could make classes appear as candidates for equal classes because any axiom makes them appear close in the document. We observed similar behavior in Onto2Vec [90] and the OPA2Vec-system [91], as discussed in Section 2.6.4 on page 25.

4.2 A family of RDF2Vec inspired systems for OWL 2

To overcome some of the limitations of using RDF2Vec, we implemented several strategies to generate random walks in ways that mirror the RDF2Vec [80] random walk strategy. We call these strategies **OWL2Vec**. These strategies rely on a completely new implementation tailored to the specific use of walking navigational graphs projected from OWL 2-ontologies. Since datatype properties are not yet considered in OWL2Vec (see Section 4.5 on page 63), we adapted the ontology projection tool described in Section 3.6 accordingly. The implementation is multithreaded to improve runtime on large ontologies.

We use SPARQL-queries to find classes and properties. For each of the classes in the ontology, we find the adjacent edges and nodes with a given depth and create a graph of all potential walks where all edges have weights. This graph is then used to generate all the walks for the given class. The system adds weights (see Table 4.1) to the edges according to the axiom created by the graph projection tool. It generates walks by taking one step at a time. The direction of the next step is determined by randomly picking an edge from the set of weighted edges, but the probability of picking an edge is relative to its weight. It is, however, worth mentioning that all the edge weights are easy to change, and could be considered parameters that the user can tweak to fit a specific application.

Labels and comments are left out of the walks as they do not contribute to the ontology structure. We also introduce a possible random jump as in page rank algorithms. The purpose of the random jump is to avoid dead ends and loops, and it can only choose nodes in the graph created for the class at hand.

The top-concept, `owl:Thing`, is omitted from the walks as it does not add any useful information. Furthermore, `owl:Thing` increases the complexity

Type of edge	Weight
subClassOf	1
type	0
object property	0.2
random jump	0

Table 4.1: Suggested weights used for the probability of the next step in random walks

of the graphs, and it adds noise by putting unrelated classes together. The `inverseOf` axiom is also omitted as this would make inverse properties too similar.

Let O_p be the projected ontology to be walked (O_p could be the union of two or more ontologies, potentially with anchor mappings), Q the list of all classes and properties and q is either a property or a class. Further, let G be the walk graph containing the edges and nodes reachable by k steps in the graph, let e be an edge and n a node and D the walk document to be created. Also, let m be the number of walks to create for each n.

Algorithm 5 OWL2Vec: a general RDF2Vec inspired system

```
Q ← All classes and properties in O_p
for all q ∈ Q do
    G ← get edges and nodes within k steps to q
    for all e ∈ E do
        e ← add weight based on the edge type
    W ← generate m random walks using the edges and weights
    D ← write the walks to the document
```

We have created several variants of this system to test different strategies: *(i)* **RDF2Vec** is using the RDF2Vec-system with the new SPARQL-query on the projection graph *(ii)* **OWL2Vec structural**, the system described above. *(iii)* **OWL2Vec subClass**, a system that creates all combinations of `subClassOf`-relations until walk depth and ignores all other relations and the number of walks per node. *(iv)* **OWL2Vec synonyms**, a system that outputs either the URI or randomly substitutes the URI with one synonym in the walks[1]. *(v)* **OWL2Vec 2doc**, a system that outputs two documents. One document containing the URIs only, and another document that prints a random synonym as a representation for the URI, but excluding the URI.

4.3 OWL2Vec+

There are several challenges with the systems above. First of all, each system is locked to a single strategy and to change walks strategy it is necessary to change the entire system. Second, they have many similarities, therefore maintaining them all would be tiresome. Third, they are too slow to scale effectively on large ontologies and at the same time being able to do long walks. The above systems are only scalable using very short walks. Fourth, it is desirable to be able to bias the walks in order to get better embeddings for specific applications.

We, therefore, implemented a more flexible OWL-ontology embedding system, **OWL2Vec+**. In addition to using ideas from the systems in 4.2 we including ideas from the node2vec-system 3.3.2, add more parameters for the user to influence how the walks should be created, and implement a new, more efficient algorithm for walks creation.

[1]The role of the synonyms is discussed in section 4.6 on page 64

We store all nodes in an internal structure along with each node's immediate neighborhood. That way, there is no need to create a neighborhood graph for each node. The entire graph is in memory and can be reused. This graph-in-memory-approach could potentially create scalability issues. Nonetheless, we have tested large ontologies on this implementation with no memory problems.

OWL2Vec+ is also implemented to use multithreading. When needed, it will add synonyms, and a `superClass`-relation (to do the reverse of the `subClass`-relation). Besides, it is possible to add entities to the classes.

The underlying implementation of the **OWL2Vec+** algorithm is described in pseudocode in Algorithm 6 on the following page. For brevity, this code is very simplified.

4.3.1 Biased walks

To control the direction of the walks, we no longer look at only the last node in the current walk and choose a random next edge from that node. Instead, we let the last two nodes influence the choice. Let the last node in the walk be *dst*, and the one before that be *src*. We still consider only the edges from *dst* as potential next edges, but *src* can influence the weight of the edges.

Two variables, p and q, are used together with the *src* and *dst* nodes to bias the walks. A high p will penalize edges returning from *dst* back to *src*, thus reduces the likelihood that a node is repeated. A low p increases the likelihood of repeating an edge. It means that having a high p will make the walks maintain the direction of a path while a walk with low p will keep going back and forward between two edges if that is possible (depending on the available edges and the direction) and covering in more detail the close proximity of the source node. A low p-value tends to bring siblings and parent nodes closer together, thus giving more weight to the immediate neighborhood and less to the larger graph structures. A high q value, on the other hand, will penalize edges pointing to nodes without edges returning to the src-node. A low q-value will have the opposite effect of favoring nodes that cannot return to the src. Thus low q will have the effect of going faster away from its source node. Using these parameters, it is possible to bias the embeddings to focus primarily on structural similarity or primarily on closely connected classes.

4.3.2 Complexity

Choosing a random edge when using weights involves iterating over a possibly large amount of edges. Node2vec uses alias sampling to choose the next edge. The alias set is calculated beforehand. Node2vec can do this as all edges have equal weight before using the p and q variables. In **OWL2Vec+**, the sampling has to be done after the creation of the node graph because we deal with weighted edges. Doing alias sampling on each node would give us much overhead. We have to iterate once over all the node's edges to update the weights according to the p and q parameters, but we can avoid iterating again for the lookup by using a treemap structure to

Algorithm 6 OWL2Vec+

Input: $p, q, number\_of\_walks, depth, number\_of\_threads,$
Output: D

$N \leftarrow all\_classes$
for all $n \in N$ **do**
    $n \leftarrow add\_all\_adjacent\_edges$
    $n \leftarrow add\_all\_synonyms$ (if requested)
for $i = 0; i < number\_of\_walks; i++$ **do**
    for all $node \in N$ **do**
        GENERATERANDOMWALK(node)

procedure GENERATERANDOMWALK(node)
    $walk = [node]$
    while $l < depth$ **do**
        $current = walk[l-1]$
        if $l = 1$ **then**
            $e \leftarrow$ CHOOSERANDOMEDGE($current$)
            if no next edge **then**
                break
            $l.append(e.outNode)$
        else
            $previous = walk[l-2]$
            $e \leftarrow$ CHOOSERANDOMEDGE($previous$, $current$)
            if no next edge **then**
                break
    return $walk$

procedure CHOOSERANDOMEDGE($srcNode$, $dstNode$)
    if $dstNode$ has outEdges **then**
        $E \leftarrow outEdges$
        $H \leftarrow newTreeMap\ for\ weighted\ edges$
        for all $e \in E$ **do**
            if $e.outNode = srcNode$ **then**
                $e.weight /= p$
            if $e.outNode$ has **no** edge with outnode = $srcNode$ **then**
                $e.weight /= q$
            add e to H according to weights
        $nextEdge \leftarrow H.next()$ using randomNumber
        return $nextEdge$
    else
        return $null$

store the edges with the obtained weight. The complexity of creating the treemaps would be O(|E|log|E|). Where |E| is the number of edges in the graph. The put operation runs in logN time, and each edge must be inserted into a collection of edges. In practice each treemap will be small, not at all near |E|. Considering thus the size of each map to be a relatively small constant (the average number of adjacent edges of a node). We expect the runtime of the creation of the treemaps to be close to O(|E|). Once created, this structure can do the lookup in O(log |E|) time. As a result, the complexity of the creation of the walks will be O(|N| + |E|). It will iterate over all nodes N, then insert each edge into a treemap to calculate the weights. Looking up the weights will be a comparably cheap operation after the treemaps have been created.

4.3.3 Walking up and down

We add a returning edge from parent to child for each `subClassOf`-edge. The addition of this edge is useful for the **OWL2Vec+** because then, instead of terminating the walk when it reaches a dead end, it can change direction and try another path.

The system does not implement shuffling of the nodes between each walk as node2vec because it makes the implementation slower. It would imply more thread synchronization. Now each thread can start on the next iteration after writing and do not have to wait for the other threads to finish writing. As all the other operations are done in memory, the file writing operation with thread synchronization can become a limiting factor.

The most important relationships in the ontology are the hierarchical relationships. The hierarchy is the backbone of the ontology and is represented in OWL by a set of `subClassOf`-relationships. If the walks are created only by following these edges, it will generate a series of short, similar walks as illustrated in Figure 4.3 on page 59. The walks will be short because the hierarchy is usually not high enough to generate, for instance 50 steps. The walks will be similar because when starting from the bottom going up it will, except for complex `subClass`-structures, have only one way to go towards the top.

On the other hand, if we were to reverse the `subClassOf`-relation, starting on the top, we can go along many possible paths towards the bottom as shown in Figure 4.4 on page 59. However, still, when we reach the bottom node, we would be at a dead end. Some walk systems (e.g., page rank) solve the dead end problem by adding the possibility of a random jump. We find, however, that this situation can be solved effectively and adds less noise by adding both `subClassOf` and the reverse `superClassOf`-relation. By adding these edges, we are virtually creating a non-directional OWL-hierarchy and can walk in both directions as shown in Figure 4.5. If we combine this strategy with the *p*-parameter, which adjust the likelihood of returning to the last node, the walks can go in one direction only (at any given time). Either up or down until it gets stuck. Then, it can reverse the direction and go the other way until it gets stuck again. If it can walk both directions, the system can generate walks from one sibling to another via

the parent node. Walking between siblings is, in contrast, not possible in a system that walks the hierarchy using `subClass`-relations only.

We can give different weights to the `subClassOf`-edge and the reverse edge. It could be argued that the `superClassOf`-edge should have a slight preference since there are more nodes at the bottom of the hierarchy than at the top. Thus we should have the walks spend more time in the lower parts of the hierarchy. But not so much that it could not go up a few steps before it goes back down. To favor the `superClassOf`-edges would also increase the likelihood of getting siblings in the same window.

Nonetheless, favoring `superClassOf`-edges has a major drawback. It does not conserve the subsumption of the embeddings. The concepts at the top of the hierarchy are more general and more important. Having the `subClassOf` probability higher than that of the `superClassOf` tends to keep more of the hierarchy's subsumption in the embeddings. In other words, it makes a `subClass`'s vector more similar to that of the parent than to that of the siblings. Consequently, we decide to set the `superClassOf`-edge is to 0.7 and leave the `subClassOf`-edge at 1.0.

To understand why we should be able to walk both ways, recall that the purpose of the walks is to create embeddings using, for instance, the Skip-gram or the CBOW algorithm. As we have seen in Section 3.2 on page 29, Skip-gram and CBOW include words both from before and after the target word, so the final order of the words is irrelevant. Only the distance to the target word is important. Words outside the window are not considered, and words that are closer to the target word are given higher weight in the Skip-gram model than more distant words. Consequently, the strategy of walking both ways should not be changing the actual embeddings, just enabling the walk-algorithm to do more and varied walks in a node's neighborhood.

The object properties are also important. Walking on object properties is the only way we can capture the relationship between separate hierarchies. Consequently, these edges can be seen as bridges between otherwise separate hierarchies of unrelated items as depicted in Figure 4.6 on page 60. The weight for these edges should be low to avoid noise. Because they usually link different, related concepts.

4.3.4 Optimizations

The ontology projection stores the triples in a turtle-file, and the graph is read from file and stored in a triplestore. The node lists are created by passing SPARQL-queries to the triplestore. To optimize the speed of accessing the triples, we have used the RDFox data store. That way all triple access is in memory and SPARQL-queries are parallelized. In order to apply weights to the edges as described, a fast way is to store all the nodes in memory with their edges. The nodes are stored by first reading in all the nodes using a simple SPARQL-query. At the same time, the system creates a HashMap<String, Node> to be able to look up the nodes when it has the URI. Then, for each node, it finds all (relevant) adjacent edges and adds them to the node's edge list. Then, the out-node is added to each of the edges' node-list. To add the out-edges, the system needs to look up the

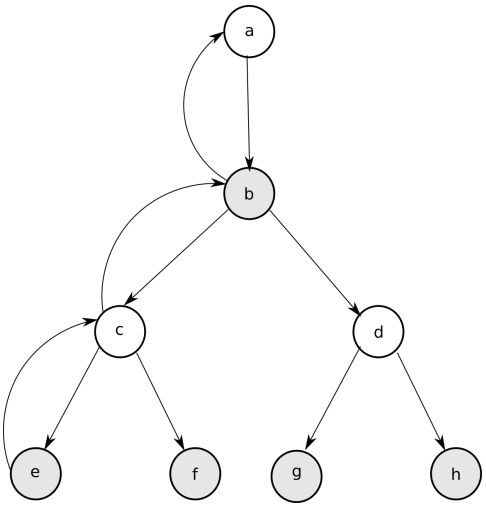

Figure 4.3: There is only one possible walk from e when following the `subClassOf`-edges. Black, straight lines represent edges. Curved lines represent the walk

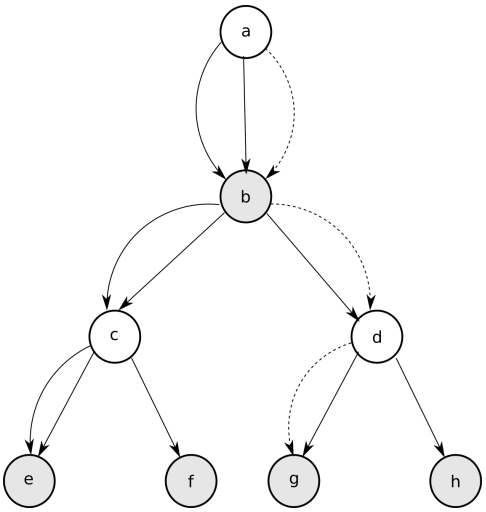

Figure 4.4: There are several possible walks from a when following the `superClassOf`-edges. Black, straight lines represent edges. The other lines represent two of the possible walks

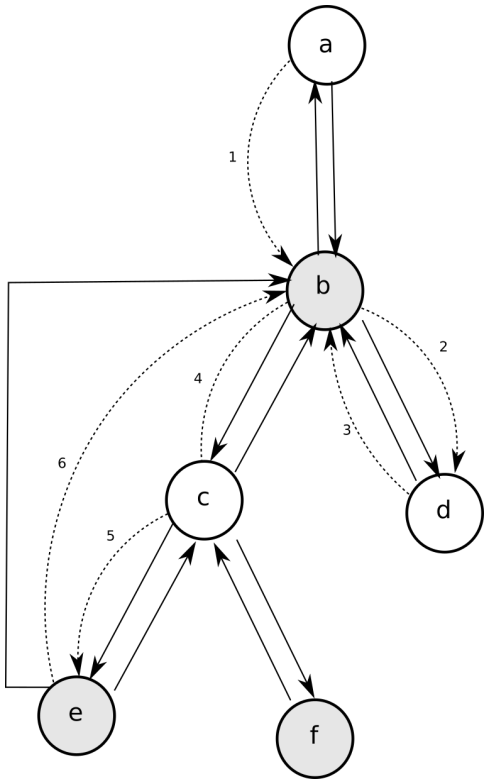

Figure 4.5: How the walks can look like if it is possible to go in both directions. The black lines are possible edges, and the dashed lines represent a possible walk along the edges

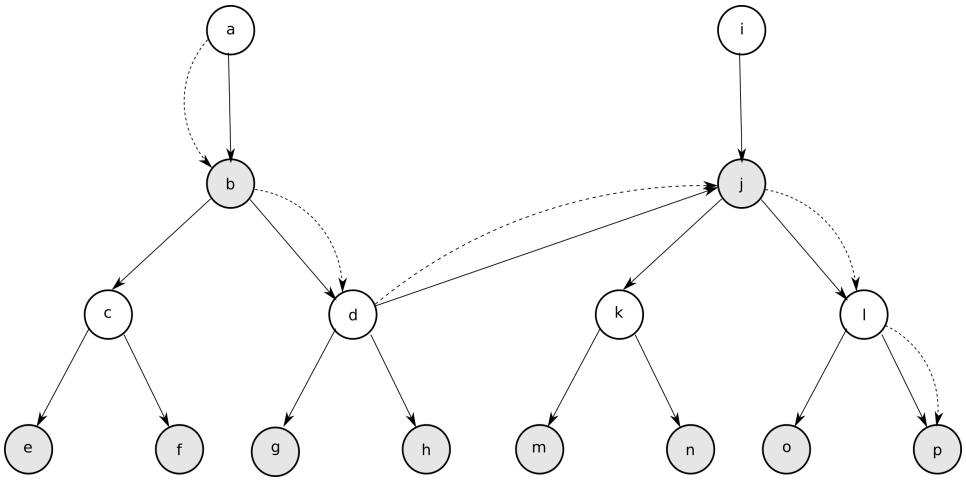

Figure 4.6: How an object property can provide a link between two hierarchies of unrelated items. The black straight lines represent edges, and the dashed lines represent a possible walk along the edges.

correct node-object using the name found in the SPARQL-query. How this is done, has a considerable impact on performance. It can find it by searching the node list in O(N), or by using the HashMap in almost O(1).

The random selection of the weighted edges using a treemap structure still takes time. We improve this first by returning whatever edge it has if the node has only one edge. Second, we have added the ability to cache the edge collections (the treemaps) in a 2-dimensional ConcurrentHashMap, and the experiments in Section 5.5 on page 89 suggest substantial efficiency gains. The caching of the edge collections can be turned on using a parameter.

The rendering of the nodes in the document also has a significant impact on the time to generate the document. It is because it uses different string operations such as regex to parse and normalize the strings. By caching these strings in the Element object instead of calculating them each time a node or edge is referenced, gives significant speedup.

4.4 Alignment strategies

In order to use the ontology embeddings for ontology alignment, we embed the different ontologies in the same vector space. We need, however, a way to "align" the embeddings. Either we can assume that there is a functional relationship between the embeddings of the different ontologies, and use the anchors to find that relationship. Alternatively, we can insert the anchors into the ontology before the projection, and as a consequence generate walks between the ontologies.

Aligning the ontologies using a transformation matrix

We begin by taking the union of the two ontologies that are to be aligned and create a projection of the combined ontology. The OWL2Vec-tool uses this graph to generate embeddings.

Having this embedding model, we apply the idea from [49] to use a transformation matrix. To find the transformation matrix, we need anchors. The anchors could come from any source as long as they are of high confidence. Using the correspondences, we can, by stochastic gradient descent, estimate a linear relationship (i.e., the transformation matrix) between the two ontologies. Then, we can use this relationship to estimate where, given a vector from the first ontology, the position of the corresponding vector from the other ontology should be in the vector space. As for the pretrained word embeddings approach, we use cosine similarity to determine the similarity of two vectors.

The transformation matrix approach is expressed in pseudocode in Algorithm 7 on the following page.

Aligning the ontologies by including the anchors in the projection

The embeddings of the elements of two distinct ontologies can be aligned in the same vector space using anchors. If no anchors are provided, the

Algorithm 7 Structure alignment using a transformation matrix

$O_U = O_1 \cup O_2$
$G \leftarrow O_U$ projecting to graph
$D \leftarrow G$ generate random walks
$M \leftarrow D$ using word2vec
input anchors A
$Dict \leftarrow [A]$
$T \leftarrow Dict$ infer transformation matrix
for all $e \in O_1$ **do**
    $maxSim = 0$
    $candidate = null$
    for all $e_2 \in Entity_2$ **do**
        $sim = max\{Sim(name_1, name_2),$
        $Sim(label_1, label_2),$
        $Sim(comment_1, comment_2)\}$
        if $sim > maxSim$ **then**
            $maxSim = sim$
            $candidate = e_2$
    $alignment \leftarrow alignment + (e_1, candidate, maxSim)$
return $alignment$

concepts of the different ontologies would be embedded separately in different parts of the vector space. We will need at least one anchor in each cluster to get the ontologies to align on all topics. Nevertheless, the more anchors we provide, the better the ontologies will align.

We start by reading both ontologies (O_1 and O_2) from files using OWL API, Then, we create a third ontology (O_U), which starts as the union of O_1 and O_2, and then we add the anchors as `equivalentClass`, `equivalentObjectProperty` and `equivalentDataProperty`-axioms. These anchors will create bridges between the two ontologies for the walks. The projection tool will later perform reasoning (using the Pellet or Elk reasoner) on the new ontology.

This reasoning will, in case of statements such as `C R D` and `E` $\equiv$ `C`, add the corresponding statement, `E R D`. The OWL2Vec tool uses the projected graph to generate embeddings. If it has enough anchors, the anchors should help to move the embedding of the two graphs so that similar concepts come close together. One could then compare the entities as in the other approaches by checking if the cosine similarity is above or below some threshold.

Alignment strategy variations

We are using several variants based on the two general strategies described above: *(i) transformation matrix* is the system using a transformation matrix described in Algorithm 7. *(ii) best candidate* is the system using anchors in projection described in Algorithm 8 on the facing page. *(iii) all relations*

Algorithm 8 Structure alignment by directly including the anchors in O_U

```
anchors ← LogMap (or any other methods for finding high confidence mappings)
O_U ← O_1 ∪ O_2
for all (a_1, a_2) ∈ anchors do
    O_U ← O_U + (a_1 ≡ a_2)
G ← O_U by graph projection
D ← G by random walks
M ← D by word2vec
for all c_1 ∈ C_1 do
    maxSim = 0
    candidate = null
    for all c_2 ∈ C_2 do
        sim = max{Sim(name_1, name_2),
        Sim(label_1, label_2),
        Sim(comment_1, comment_2)}
        if sim > maxSim then
            maxSim = sim
            candidate = c_2
    alignment ← alignment + (c_1, candidate, maxSim)
return alignment
```

does not find the best, but includes all the mappings that are above the equality threshold. Finding all the possible correspondences is useful to observe how much recall it is possible to obtain with a certain equality threshold or system. *(iv) disambiguate* uses the structural embeddings with a lower equality threshold to find a set of possible matches. Then it uses edit distance to choose one of them. *(v) two documents* also uses the structural embeddings to find a set of possible matches, but it uses embeddings of the lexical information in the ontology, created from the second document, to disambiguate.

4.5 Properties in the embeddings

4.5.1 Object properties

The object properties are useful when creating the embeddings because they help to add additional characteristics to the different classes and by providing a path to another part of the ontology hierarchy. That way it makes us able to capture the relationship between for example `author` and `paper`. It is, however, not easy to create useful structural embeddings for the actual properties because there is often less information about the properties in an ontology than the classes. Besides, they tend to connect very different concepts. In contrast, `subClass`-relations relate only similar concepts. Consequently, we narrow the focus in this thesis to the creation of class-embeddings only. Nevertheless, the system can output embeddings

for object properties, but they must be considered less reliable.

4.5.2 Datatype properties

Another, even more significant, challenge with the random walks is for the data properties. These properties will always be at the end of a walk as they always relate some class to a string (a datatype). It makes all datatype properties appear as the structurally equal. The embeddings system will mostly ignore string because it by default ignores low-frequency words. The only context they have is a single class. It is, as far as we understand, impossible to distinguish them using walks only.

In order to distinguish datatype properties, it would be necessary to use other methods. First one should introduce a way to distinguish data types, cardinality restrictions and so on. Even then methods such as string comparisons, pretrained word embeddings are likely to be needed to distinguish datatype properties. Because of these challenges we choose to leave this for future work and ignore datatype properties for now.

4.6 Labels and synonyms in the embeddings

A challenge for the embeddings model described above is that, for some classes, it is difficult to disambiguate `subClassOf`-relations. An example from the ekaw-ontology is:

- `Late_Registered_Participant` $\sqsubseteq$ `Conference_Participant`
- `Early_Registered_Participant` $\sqsubseteq$ `Conference_Participant`

As there is no more information about these classes in the ontology they are structurally equal and not possible to distinguish using purely structural methods.

As described in subsection 4.2, labels and comments are left out of the walks. They do, however, contain useful information. In addition to the lexical information in the labels and comments, one should also use potential information in the URIs by parsing and extracting the words.

Instead of including labels and comments as structures in the structural walks these descriptions could be used to substitute the URIs, so they are included in the walks at the correct place. There are several ways this can be done. All synonyms could be joined into one string separated by whitespace. This string, containing all related lexical information to a concept could randomly substitute the URI. Alternatively, one synonym chosen at random could substitute the URI, or we could choose to exclude the URIs entirely. While this would not be able to replace the addition of anchors in the ontology, it will make concepts that contain the same words more similar. It would also influence the vectors of its surrounding words (within the window).

Another strategy is to create an additional document so that we have two documents for the generation of embeddings. One containing the

structural relationships using the URIs and one document containing the lexical information, this would of course also have structural information, but all equal words across the ontologies would be represented with one vector. These vectors would again influence the vectors of its context words. One will then have to combine the two similarity scores obtained by the two documents.

4.6.1 Sentence vectors

Comments and labels and even URIs often consist of more than one word. Having a sentence makes it impossible to calculate the cosine similarity directly. There are several proposed ways to cope with this situation, and a simple method is to use the average vector of all the words and take the cosine similarity with the average vector of the sentence of the other ontology. This approach has, according to [41], shown to be a good way to represent a sentence, but not without limitations. One will, for example, lose information about word order. Additionally, it could be a good idea to remove stop words that do not add meaning to the sentence. For example *the, and*. These words would make very different sentences more similar (e.g., *The dog* and *the emotion* would be 50 % equal). Using the average vector, the sentences will be more similar when there are more similar words regardless of the importance of the words. Finding the average is done by adding up dimension for dimension and dividing each dimension on the number of words. Another approaches commonly used is adding or subtracting word vectors. The resulting vector can be normalized if necessary. The cosine distance does not depend on the length of the vectors, so adding and averaging should give similar results.

The focus of the thesis is not about combining word vectors into sentence vectors, so we keep it simple and use averaging. However, we would like to highlight this because it is an important issue that we encounter often, and it is likely to influence the results.

4.7 Hyperparameters

Any machine learning system has parameters that have to be tuned before training. This system is no exception.

4.7.1 Dimensions

The dimension of the embedding determines the precision of the model. If the dimension is high, more properties could be preserved. Preserving more properties is useful if the aim is to create an as good as possible model of the graph. On the other hand, having a too precise model could lead to overfitting. For example, if the goal is to predict missing information in the graph, it is necessary to allow for some generalization. A high dimensional model takes more time to train than a low dimensional model. A low dimensional model will mean more generalization and a high dimensional model will be more detailed.

Ristoski and Paulheim report that the vectors with a dimension of 500 generally performed better than the vectors of 200 [80]. Grover and Leskovec, on the other hand, report that performance tends to increase up to dimensions of about 100 [80]. After that, there is not much improvement.

4.7.2 Skip-gram or CBOW?

Both word2vec and `fastText` allows the user to choose between Skip-gram and CBOW. Ristoski and Paulheim observe in [80] that the Skip-gram-model performs better than the CBOW-model on RDF-graphs.

Mikolov says in a google group [48] that Skip-gram works better than CBOW on a small amount of training data while CBOW is much faster to train and gives better performance on frequent words.

4.7.3 Hierarchical softmax or negative sampling?

Ristoski and Paulheim comment that empirical studies show that negative sampling tends to give better performance than hierarchical softmax [80].

4.7.4 Character n-grams

For the `fastText` algorithm, one has to decide the max and min length of the n-grams. Default values are 3-6 characters, and according to [9] this is an arbitrary value that seems to work well.

4.7.5 Window size

Another parameter that affects the quality of the embeddings is the window size. A typical value is 5, but increasing the window tends to increase the quality of the embeddings.

4.7.6 Epochs

The Epochs count says how many times the training will be performed on the dataset. If we have little data it can be necessary to train many times. `FastText` claims to train on less data than word2vec do. If we do too many epochs, we risk overfitting.

4.7.7 Node2vec parameters

The parameters p and q are described in the section on node2vec (Subsection 3.3.2 on page 37).

4.7.8 Depth of walk

Longer walks make it possible to include relationships that are far away in the graph, thus include more structural information. They also result in a larger dataset for training. Shorter walks give information on close relationships only.

4.7.9 Number of walks

The number of walks tells the system how many random walks to generate for each node. A larger value gives a larger dataset. A small value will give few different examples of the context of the node, but to many walks will return many equal walks. It is related to to the number of epochs needed to train. With many walks, the number of epochs could be lower.

4.7.10 Equality threshold

This parameter is for ontology matching only. The equality threshold must be set for each matching task because it is influenced by the amount of training. A high value will increase precision but limit recall. If the embeddings are very dense, the value must be set high to avoid false positives, but if the embeddings are sparse, the value must be lower to include correct matches.

Chapter 5

Experiments and results

Many experiments have been conducted in order to demonstrate the usefulness of the proposed ontology embeddings framework (OWL2Vec) on both inter- and intra-ontology similarity tasks. The different OWL2Vec systems and strategies have been presented in Sections 4.2 and 4.3.

First, Section 5.1 presents an experiment on clustering and labeling. Section 5.2 on page 73 shows the experiments using the systems for ontology matching. Section 5.3 on page 84 is dedicated to using the LogMap's anchors and try to improve LogMap's recall. Then, an experiment using the embeddings to find a measure of semantic similarity between ontologies is presented in section 5.4 on page 89. Last, section 5.5 on page 89 provides the experiments on scalability.

5.1 Intra-ontology clustering

This experiment aims to demonstrate the usefulness of the OWL2Vec-systems on intra-ontology similarity tasks. First, we did a visual assessment of the embeddings using different strategies and systems. These results are available in Appendix B.1 on page 119. We also did a short similarity test for the **RDF2Vec**, the Onto2Vec (introduced in Section 2.6.4 on page 25) and **OWL2Vec+**. The results of these experiments are in Appendix B.2 on page 122. Finally, we did an experiment on node clustering on the ekaw-ontology (introduced in Section 3.7 on page 46).

For the clustering task, we used the Agglomerative Clustering algorithm in the sklearn Python library [70]. After doing the clusterings, we reduced the vector dimensions to 2 with the PCA library in order to plot them on a page. The names of all the points are left out for readability. By adding all the vectors in each cluster, we get a vector representing the cluster. By finding the nearest neighbor to the cluster vector, we expect to find the most central concept of the cluster. This concept is used as a label for the cluster.

These evaluations focus only on the ontology's structure, so no lexical information is included in the walks. The settings for word2vec are mostly equal for all the tests, with Skip-gram and 200-dimensional embeddings. The number of epochs was adjusted, however, to fit the different sizes of the documents generated by the walks systems. **OWL2Vec+** uses walks

with depth 40. **RDF2Vec** uses walks with depth 5 because it takes a long time to generate longer walks. We increase the epoch for the training to compensate for the smaller documents. The Onto2Vec-system create very small documents, so we increase the epoch count even more for this system. The Onto2Vec does not use a projection graph. It works directly on the OWL 2 ontology.

The **RDF2Vec** and the **OWL2Vec+** system were trained directly on URIs, but only the URI-part is presented. In all three experiments, the ekaw ontology is partitioned into 7 clusters.

The tables of the cluster contents are found in Appendix B on page 119.

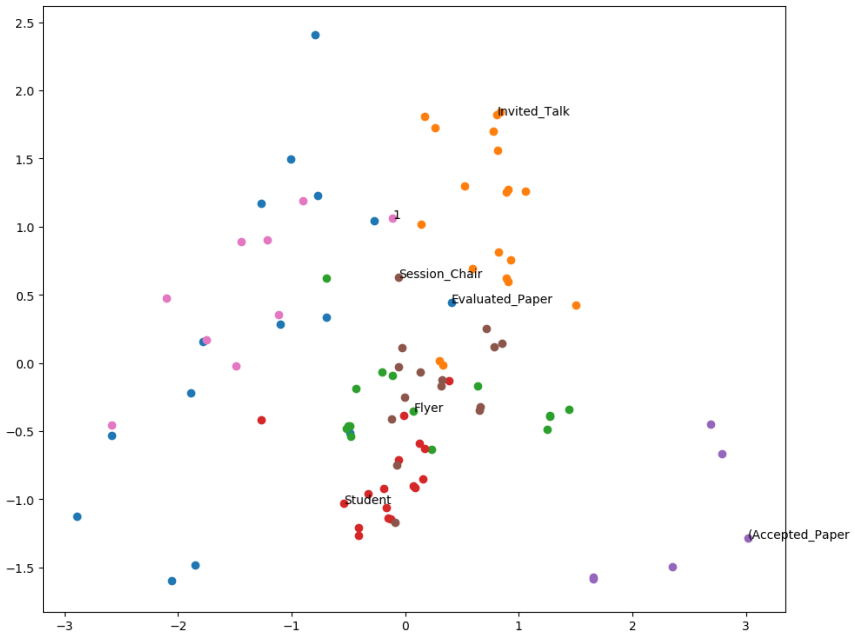

Figure 5.1: Dividing the embedding of ekaw into 7 clusters using the Onto2Vec framework for the walks and agglomerative clustering for the clustering task

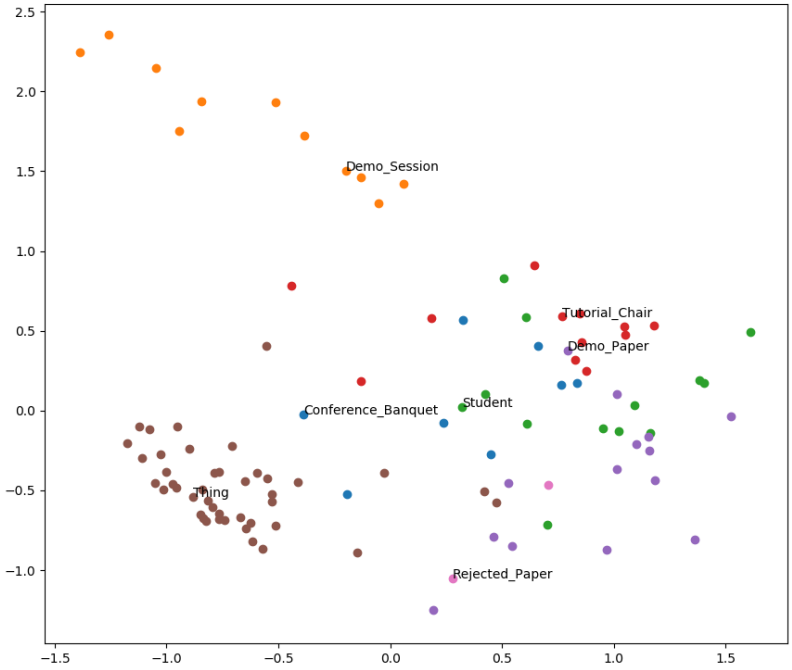

Figure 5.2: Dividing the embedding of ekaw into 7 clusters using the **RDF2Vec** framework for the walks and agglomerative clustering for the clustering task

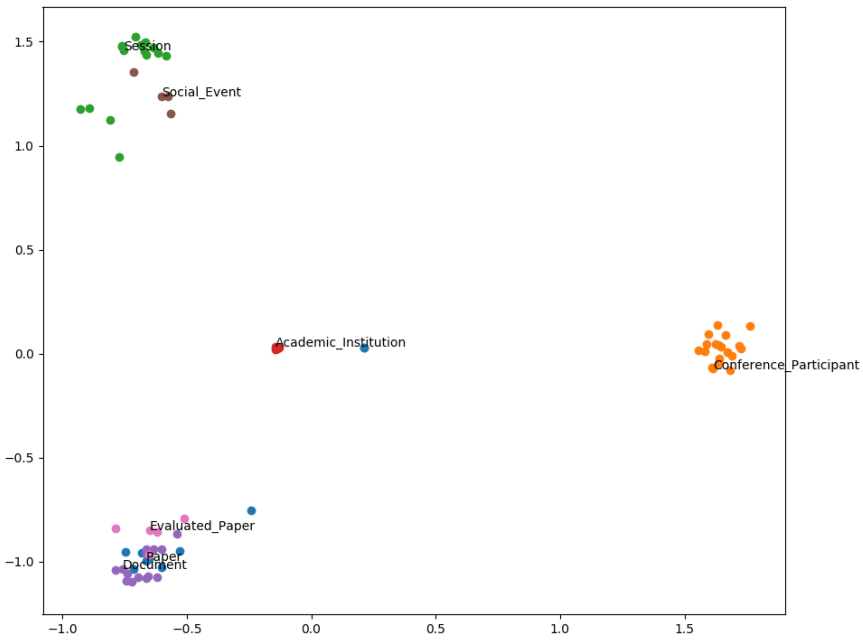

Figure 5.3: Dividing the embedding of ekaw into 7 clusters using the **OWL2Vec+** framework for the embeddings and agglomerative clustering for the clustering task

5.2 Ontology matching tasks

5.2.1 Pretrained model

We used a model with 100-dimensional vectors trained on a small fragment of Wikipedia [1]. In these experiments, we use an equality cutoff for the cosine similarity of 0.8 for the conference tracks and 0.99 for the anatomy track. The actual mappings found for ekaw-cmt is in Appendix D.1 on page 135.

Matching task	Precision	Recall	F-measure
ekaw-ekaw	0.97	1	0.99
ekaw-cmt	0.67	0.75	0.7
Anatomy	0.55	0.25	0.34

Table 5.1: Performance of pretrained embeddings for ontology alignment

5.2.2 Joint embeddings

It is essential to have some initial anchors in order to use structural embeddings for ontology matching. Consequently, the experiments here do not intend to discover all mappings, but rather to discover to which extent the models depend on the anchors and to see if it is possible to get a higher return of mappings from the system than the input of anchors.

The use of random walks and machine learning allows for some randomness, but this should be averaged out by performing multiple runs. It is also important to note that the choice of anchors is not arbitrary. Some anchors or combination of anchors will give better performance than others. For this reason, we shuffle the set of anchors before each test run. The tests are performed at least five times (unless other is stated) and the average scores are reported.

The ontologies used in the standalone ontology matching experiments are two ontologies from the OAEI conference track: ekaw and cmt. Also, we use a copy of the ekaw ontology to test how the systems can perform on two structurally (and lexically) equal ontologies. The gold standard alignment from OAEI is used both for anchors and for the calculation of recall, precision and $F_{0.5}$-measure (F-measure). For some of the systems, we also include a test with the OAEI anatomy track.

The correlation coefficient between the fraction of all known anchors and the recall of the system was calculated using the Pearson correlation coefficient (PCC) [69]. PCC is defined as the covariance divided by the product of the standard variations. If the coefficient is ρ, $cov(x, y)$ is the covariance between the variables x and y, σ_X is the standard deviation of X and σ_Y is the standard deviation of Y, PCC is given by:

$$\rho = \frac{cov(X, Y)}{\sigma_X \sigma_Y} \tag{5.1}$$

[1] This is the same Wikipedia fragment as in Subsection 3.2.6 on page 32

Where $cov(X, Y)$ is given by:

$$\frac{\sum(X_i - X_{avg})(Y_i - Y_{avg})}{n-1} \tag{5.2}$$

Some alignment strategies were tested in combination with the different walks systems and embeddings strategies to see the effect of for example using all matches, disambiguating using edit distance and using both structural and lexical embeddings. The different ontology alignment strategies are introduced in Section 4.4 on page 61.

Running and comparing the experiments is challenging because of the many parameters involved. The parameters have to be changed between runs because some systems work poorly or not at all using the parameters that work well for other systems. An overview of the parameters used for the different test runs is found in Table 5.2 on the next page.

System	Parameter	Value
All embedding systems		
all alignment strategies	window size	10
	dimensions	100
	number of walks	100
	negative samples	25
	include edges	false
	embedding system	gensm w2v
	embeddings representation	full uri
	workers	12
	Skip-gram	true
OWL2Vec structural		
all alignment strategies	epochs	50
best candidate	walk depth	5
	equality threshold	0.7
disambiguate	walk depth	5
	equality threshold	0.5
	lexical eq. thres	0.7
RDF2Vec		
best candidate	walk depth	4
	epochs	15
	equality threshold	0.7
OWL2Vec synonyms		
all alignment strategies	epochs	50
	walk depth	5
	number of walks	400
best candidate	equality threshold	0.6
disambiguate	equality threshold	0.5
	lexical eq. thresh.	0.7

transformation m	equality threshold	0.5
	lexical eq. thresh.	0.7
OWL2Vec 2doc		
two documents	structure epochs	15
	lexical epochs	50
	structural eq thresh	0.5
	lexical eq thresh	0.7
	walk depth	5
OWL2Vec+		
best candidate	epochs	15
	equality threshold	0.85
	number of walks	50
	walk depth	40
	p	1.2
	q	0.5
OWL2Vec+–synonyms		
best candidate	epochs	5
	equality threshold	0.85
	number of walks	50
	walk depth	40
	p	1.2
	q	0.5
OWL2Vec+–two documents		
two documents	label doc epochs	50
	uri epochs	5
	equality threshold	0.85
	number of walks	50
	walk depth	40
	p	1.2
	q	0.5

Table 5.2: The parameter settings for each OWL2Vec system in combination with each alignment strategy

For all ekaw-ekaw and ekaw-cmt matching experiments, we include the results for 0, 40, 60 and 100 % of the anchors. The details for every single test run on the ekaw-ekaw matching are left to Appendix C on page 129, and the details for the cmt-ekaw experiments to Appendix D on page 135. We have also included the Pearson's Correlation between the fraction of anchors and recall. The precision tends to be independent of the number of anchors, so no such calculation is included for the precision.

We also include experiments using **OWL2Vec+** with synonyms and `fastText` for ekaw-ekaw and cmt-ekaw matching to demonstrate the potential for the matching similar ontologies using `fastText`.

The included figures (Figure 5.4 on page 78 and Figure 5.5 on page 82)

plot the F-measure against the number of anchors included. We use the F-measure because the precision varies a lot between the different tests and to compare the quality of each run, it must include both the notion of recall and precision.

Ekaw-ekaw matching task

walks	model	precision	recall	fmeasure
OWL2Vec structural	*best candidate*	0.87	0.75	0.81
OWL2Vec structural	*disambiguate*	0.97	1.00	0.99
OWL2Vec synonyms	*best candidate*	0.85	0.81	0.83
OWL2Vec synonyms	*disambiguate*	0.96	0.97	0.97
OWL2Vec synonyms	*all relations*	0.44	0.92	0.59
OWL2Vec synonyms	*transformation m*	1.00	0.90	0.95
RDF2Vec	*best candidate*	0.83	0.63	0.72
OWL2Vec subClass	*best candidate*	0.78	0.60	0.68
OWL2Vec 2doc	*two documents*	0.57	0.48	0.52
OWL2Vec+	*best candidate*	0.93	0.87	0.89
OWL2Vec+ [1]	*best candidate*	0.97	0.97	0.97
OWL2Vec+ [2]	*best candidate*	0.97	1.00	0.99

[1] Including synonyms and URIs in the same document

[2] Including synonyms and URIs in the same document, trained with `fastText`

Table 5.3: A comparison of the systems performance on 100 % anchors. on the ekaw-ekaw matching.

walks	model	precision	recall	fmeasure
OWL2Vec structural	*best candidate*	0.85	0.48	0.61
OWL2Vec structural	*disambiguate*	0.85	0.84	0.84
OWL2Vec synonyms	*best candidate*	0.83	0.61	0.70
OWL2Vec synonyms	*disambiguate*	0.93	0.85	0.89
OWL2Vec synonyms	*all relations*	0.51	0.72	0.59
OWL2Vec synonyms	*transformation m*	1.00	0.77	0.87
RDF2Vec	*best candidate*	0.60	0.44	0.50
OWL2Vec subClass	*best candidate*	0.73	0.68	0.71
OWL2Vec 2doc	*two documents*	0.63	0.52	0.57
OWL2Vec+	*best candidate*	0.83	0.66	0.74
OWL2Vec+ [1]	*best candidate*	0.94	0.79	0.86
OWL2Vec+ [2]	*best candidate*	0.97	0.99	0.98

[1] Including synonyms and URIs in the same document
[2] Including synonyms and URIs in the same document, trained with `fastText`

Table 5.4: A comparison of the systems performance on 60 % anchors. on the ekaw-ekaw matching.

walks	model	precision	recall	fmeasure
OWL2Vec structural	*best candidate*	0.86	0.31	0.46
OWL2Vec structural	*disambiguate*	0.72	0.62	0.67
OWL2Vec synonyms	*best candidate*	0.74	0.42	0.54
OWL2Vec synonyms	*disambiguate*	0.89	0.68	0.76
OWL2Vec synonyms	*all relations*	0.50	0.52	0.50
OWL2Vec synonyms	*transformation m*	1.00	0.49	0.66
RDF2Vec	*best candidate*	0.73	0.27	0.39
OWL2Vec subClass	*best candidate*	0.67	0.65	0.66
OWL2Vec 2doc	*two documents*	0.71	0.56	0.64
OWL2Vec+	*best candidate*	0.71	0.44	0.52
OWL2Vec+ [1]	*best candidate*	0.95	0.57	0.70
OWL2Vec+ [2]	*best candidate*	0.95	0.90	0.92

[1] Including synonyms and URIs in the same document
[2] Including synonyms and URIs in the same document, trained with `fastText`

Table 5.5: A comparison of the systems performance on 40 % anchors. on the ekaw-ekaw matching.

walks	model	precision	recall	fmeasure
OWL2Vec structural	*best candidate*	0.40	0.01	0.01
OWL2Vec structural	*disambiguate*	0.71	0.03	0.06
OWL2Vec synonyms	*best candidate*	0.86	0.13	0.23
OWL2Vec synonyms	*disambiguate*	0.84	0.34	0.48
OWL2Vec synonyms	*all relations*	0.69	0.15	0.24
OWL2Vec synonyms	*transformation m*	0.04	0.01	0.02
RDF2Vec	*best candidate*	0.00	0.00	0.00
OWL2Vec subClass	*best candidate*	0.04	0.04	0.04
OWL2Vec 2doc	*two documents*	0.76	0.29	0.41
OWL2Vec+	*best candidate*	0.70	0.01	0.03
OWL2Vec+ [1]	*best candidate*	0.40	0.02	0.03
OWL2Vec+ [2]	*best candidate*	0.75	0.03	0.05

[1] Including synonyms and URIs in the same document
[2] Including synonyms and URIs in the same document, trained with `fastText`

Table 5.6: A comparison of the systems performance on 0 % anchors. on the ekaw-ekaw matching.

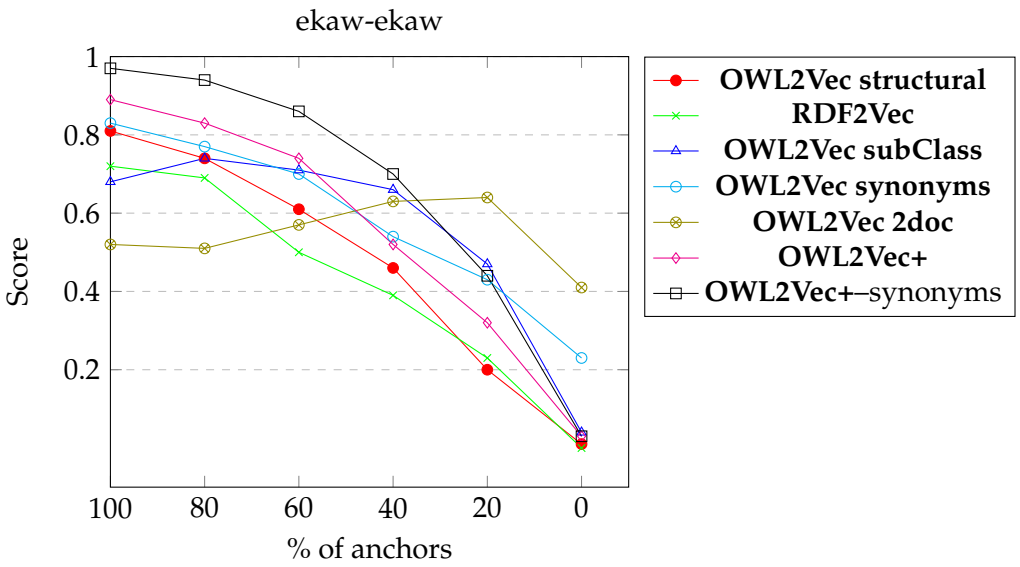

Figure 5.4: F-measure for different systems on ekaw-ekaw matching

walks	model	correlation
OWL2Vec structural	*best candidate*	0.996
OWL2Vec structural	*disambiguate*	0.967
OWL2Vec synonyms	*best candidate*	0.992
OWL2Vec synonyms	*disambiguate*	0.961
OWL2Vec synonyms	*all relations*	0.988
OWL2Vec synonyms	*transformation m*	0.969
RDF2Vec	*best candidate*	0.990
OWL2Vec 2doc	*two documents*	0.340
OWL2Vec+	*best candidate*	0.988
OWL2Vec subClass	*best candidate*	0.743

Table 5.7: Correlation between F-measure and anchors on ekaw-ekaw matching

Ekaw-cmt matching task

The results of the tests with **RDF2Vec** are weak when it comes to recall and precision. Looking at the actual results, however, they are not entirely off. The system usually suggests similar concepts, but the system seems unable to disambiguate structurally similar concepts. The results of an example run with **RDF2Vec** is found in Table D.2 on page 137.

walks	model	precision	recall	fmeasure
OWL2Vec structural	*best candidate*	0.70	0.60	0.64
OWL2Vec structural	*disambiguate*	0.26	0.85	0.39
OWL2Vec synonyms	*best candidate*	0.74	0.45	0.56
OWL2Vec synonyms	*disambiguate*	0.38	0.68	0.48
OWL2Vec synonyms	*all relations*	0.63	0.40	0.48
OWL2Vec synonyms	*transformation m*	0.53	1.00	0.69
RDF2Vec	*best candidate*	0.17	0.13	0.14
OWL2Vec subClass	*best candidate*	0.60	0.95	0.74
OWL2Vec 2doc	*two documents*	0.61	0.75	0.67
OWL2Vec+	*best candidate*	0.69	0.98	0.81
OWL2Vec+ [1]	*best candidate*	0.97	0.78	0.86
OWL2Vec+ [2]	*best candidate*	0.60	0.95	0.73

[1] Including synonyms and URIs in the same document
[2] Including synonyms and URIs in the same document, trained with `fastText`

Table 5.8: A comparison of the systems performance on 100 % anchors. on the cmt-ekaw matching.

walks	model	precision	recall	fmeasure
OWL2Vec structural	*best candidate*	0.64	0.30	0.53
OWL2Vec structural	*disambiguate*	0.16	0.53	0.24
OWL2Vec synonyms	*best candidate*	0.73	0.20	0.31
OWL2Vec synonyms	*disambiguate*	0.36	0.45	0.40
OWL2Vec synonyms	*all relations*	0.77	0.33	0.45
OWL2Vec synonyms	*transformation m*	0.30	0.63	0.41
RDF2Vec	*best candidate*	0.00	0.00	0.00
OWL2Vec subClass	*best candidate*	0.33	0.75	0.45
OWL2Vec 2doc	*two documents*	0.60	0.70	0.64
OWL2Vec+	*best candidate*	0.66	0.55	0.59
OWL2Vec+ [1]	*best candidate*	0.93	0.50	0.63
OWL2Vec+ [2]	*best candidate*	0.61	0.63	0.62

[1] Including synonyms and URIs in the same document
[2] Including synonyms and URIs in the same document, trained with `fastText`

Table 5.9: A comparison of the systems performance on 60 % anchors. on the cmt-ekaw matching.

walks	model	precision	recall	fmeasure
OWL2Vec structural	*best candidate*	0.35	0.23	0.27
OWL2Vec structural	*disambiguate*	0.14	0.43	0.21
OWL2Vec synonyms	*best candidate*	0.83	0.23	0.34
OWL2Vec synonyms	*disambiguate*	0.32	0.23	0.26
OWL2Vec synonyms	*all relations*	0.49	0.18	0.23
OWL2Vec synonyms	*transformation m*	0.15	0.38	0.21
RDF2Vec	*best candidate*	0.00	0.00	0.00
OWL2Vec subClass	*best candidate*	0.19	0.40	0.28
OWL2Vec 2doc	*two documents*	0.62	0.70	0.66
OWL2Vec+	*best candidate*	0.55	0.50	0.52
OWL2Vec+ [1]	*best candidate*	1.00	0.43	0.59
OWL2Vec+ [2]	*best candidate*	0.65	0.40	0.49

[1] Including synonyms and URIs in the same document
[2] Including synonyms and URIs in the same document, trained with `fastText`

Table 5.10: A comparison of the systems performance on 40 % anchors. on the cmt-ekaw matching.

walks	model	precision	recall	fmeasure
OWL2Vec structural	*best candidate*	0.00	0.00	0.00
OWL2Vec structural	*disambiguate*	0.11	0.08	0.09
OWL2Vec synonyms	*best candidate*	0.30	0.05	0.08
OWL2Vec synonyms	*disambiguate*	0.30	0.05	0.08
OWL2Vec synonyms	*all relations*	0.20	0.03	0.04
OWL2Vec synonyms	*transformation m*	0.08	0.13	0.10
RDF2Vec	*best candidate*	0.00	0.00	0.00
OWL2Vec subClass	*best candidate*	0.07	0.23	0.10
OWL2Vec 2doc	*two documents*	0.37	0.20	0.26
OWL2Vec+	*best candidate*	0.20	0.03	0.04
OWL2Vec+ [1]	*best candidate*	0.60	0.08	0.39
OWL2Vec+ [2]	*best candidate*	0.47	0.15	0.23

[1] Including synonyms and URIs in the same document
[2] Including synonyms and URIs in the same document, trained with `fastText`

Table 5.11: A comparison of the systems performance on 0 % anchors. on the cmt-ekaw matching.

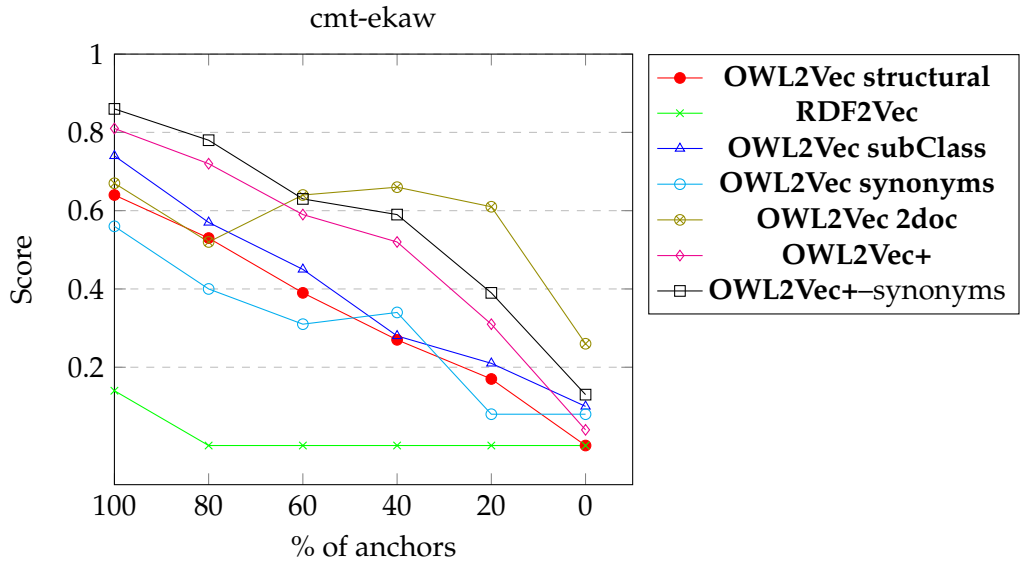

Figure 5.5: F-measure for different systems on cmt-ekaw matching

walks	model	correlation
OWL2Vec structural	*best candidate*	0.994
OWL2Vec structural	*disambiguate*	0.990
OWL2Vec synonyms	*best candidate*	0.956
OWL2Vec synonyms	*disambiguate*	0.982
OWL2Vec synonyms	*all relations*	0.948
OWL2Vec synonyms	*transformation m*	0.982
RDF2Vec	*best candidate*	0.654
OWL2Vec 2doc	*two documents*	0.688
OWL2Vec+	*best candidate*	0.991
OWL2Vec subClass	*best candidate*	0.991

Table 5.12: Correlation between recall and anchors on cmt-ekaw matching

5.2.3 Anatomy track

We tested the **OWL2Vec+**-system with *best candidate*-strategy on the anatomy track. On this test, we did only one run for each of the anchor steps. The PCC between anchors and recall was found to be 0.997.

Anchors	Precision	Recall	F-measure
100	0.58	0.96	0.72
80	0.49	0.78	0.60
60	0.40	0.59	0.47
40	0.30	0.40	0.34
20	0.21	0.20	0.21
0	0.04	0.00	0.00

Table 5.13: The **OWL2Vec+** and *best candidate*'s performance on the anatomy track using decreasing % anchors

5.2.4 More than two ontologies

We also conducted an experiment where we created one large document containing all the information in the ekaw, cmt, iasted, and the edas conference ontologies. Then, we added the reference anchors for edas-iasted, cmt-iasted, cmt-edas, ekaw-iasted, and ekaw-edas to the ontology. To create the embeddings we used the **OWL2Vec+**-system. We did 5 epochs, and only one test run for each anchor step. The equality threshold was set to 0.9. Object properties were not included in this test (i.e., the weight of object property edges was set to 0).

Anchors	Precision	Recall	F-measure
100	0.40	0.75	0.52
80	0.46	0.75	0.57
60	0.37	0.75	0.50
40	0.50	0.75	0.60
20	0.38	0.75	0.50
0	0.45	0.62	0.52

Table 5.14: The **OWL2Vec+** with *best candidate*'s performance on matching **ekaw-cmt** conference track including the ontologies iasted, edas and gold standard anchors for the other ontologies using decreasing % anchors.

5.3 Improving LogMap

To determine if it is possible to improve precision or recall on state-of-the-art systems, we use LogMap [82] as an example of a state-of-the-art system. This system is introduced in Section 3.1 on page 27. We add LogMap as a dependency in the embeddings system and by doing the matching, we can access the anchors from the matching system. These anchors should be sane and free of any logical inconsistencies. In this experiment, the anchors are used directly as anchors, and **OWL2Vec+** is used to generate embeddings. LogMap also provides access to other categories of mappings. There are "hard discarded mappings," "conflictive mappings," "discarded mappings" in addition to the anchors.

The anchors are added to the mappings as a means to create the aligned embeddings and are expected to have very high cosine similarity. We investigate if any of these anchors get a lower similarity than expected. These anchors will be candidates for further examinations.

Likewise, we check if there are any mappings with high cosine similarity among the discarded mappings. Before flagging them as possible true mappings, we must check that none of the elements in the possible mapping do appear as elements in the anchors set. If they do, we assume that this has higher confidence and not include them in the set.

We evaluated a few methods for detecting outliers among the anchors and the discarded mappings. One way was using the mean and standard deviation. The other method we tested was the Inter Quartile Range (IQR). Another possibility is to take the n largest or smallest cosine values and check them for possible errors. We will then, however, have to estimate how many errors the alignment contains.

Parameter	Value
dimensions	50
window size	10
number of walks	50
walk depth	10
`subClassOf`-weight	1
`superClass`-weight	1
object property-weight	0.4
Skip-gram	true
negative samples	25
include edges	false
embeddings system	gensim word2vec

Table 5.15: Parameters for the experiment improving-LogMap experiments

The parameters used in the test runs are shown in Table 5.15. The NCI-FMA task (full ontologies) is done with 10 walks and depth 3. To evaluate

the results, we use the gold standard alignments. We can then see how many mappings the embeddings mark as possible bad anchors or good discarded mappings and find how many of these are following the gold standard alignment.

The results of the experiments with the full FMA-NCI ontologies are found in Table 5.16 while the experiments with the small fragments of FMA-NCI are found in Table 5.17 on the next page. In Table 5.18 on page 87 the results from the matching between NCI and SNOWMED is presented, and finally, the results from the anatomy track are found in Table 5.19 on page 88. In all the tables, we present several strategies for finding potential correct mappings marked as false, and false anchors. *Mapping* is the type of mappings set (this is found in LogMap), *found* is how many mappings the filtering strategy returned, and *correct* is how many of the mappings found are actually correct. For anchors, *correct* is how many of the low confidence anchors are not found in the gold standard. For discarded mappings, *correct* is how many of the high confidence discarded mappings are found in the gold standard.

mapping	found	correct	fraction
Top-20			
anchors	20	2	0.1
hard	20	0	0.0
conflict	20	0	0.00
discarded	20	5	0.25
Mean +- stdDev			
anchors	69	6	0.09
hard	0	0	0.00
conflict	0	0	0.00
discarded	0	0	0.00

Table 5.16: Detecting false mappings and map discardings on largebios FMA-NCI full ontologies

We also did test runs of this on the cmt-ekaw. With 50 walks, 40 depth, trained with word2vec on 50 dimensions, window 20, 5 iterations, 25 negative sampling. Using the mean stdDev it correctly identified one of the anchors that LogMap added as a false anchor. That is the `cmt:Reviewer - ekaw:PossibleReviewer`-mapping. Doing multiple runs it would also occasionally suggest that the `cmt:Paper - ekaw:Paper` and the `cmt:Conference - ekaw:Conference` could be false anchors. Alternatively, that `cmt:PaperAbstract - ekaw:Abstract` could be a false discarded mappings. However, this still has a low similarity.

mapping	found	correct	fraction
Top-1000			
anchors	1000	36	0.035
hard	1000	15	0.015
conflict	734	90	0.12
discarded	270	35	0.13
Top-20			
anchors	20	1	0.05
hard	20	0	0.0
conflict	20	4	0.2
discarded	20	7	0.35
Top-5			
anchors	5	0	0.0
hard	5	0	0.0
conflict	5	1	0.4
discarded	5	2	0.2
Mean +/- stdDev			
anchors	281	17	0.06
hard	0	0	0.00
conflict	102	14	0.14
discarded	0	0	0.0
IQR			
anchors	130	7	0.05
hard	0	0	0.00
conflict	41	7	0.17
discarded	0	0	0.0

Table 5.17: Detecting false mappings and map discardings on largebios FMA-NCI small fragments

mapping	found	correct	fraction
Top-20			
anchors	20	1	0.05
hard	20	1	0.05
conflict	20	2	0.1
discarded	20	12	0.6
Mean +- stdDev			
anchors	268	18	0.067
hard	3316	214	0.065
conflict	251	71	0.28
discarded	317	118	0.37
IQR			
anchors	58	7	0.12
hard	0	0	0.00
conflict	88	18	0.20
discarded	13	8	0.62

Table 5.18: Detecting false mapping and map discardings on large bios NCI-SNOMED small fragments

mapping	found	correct	fraction
Top-1000			
anchors	1000	21	0.021
hard	1000	58	0.058
conflict	64	1	0.015
discarded	10	1	0.1
Top-5			
anchors	5	0	0.0
hard	5	1	0.2
conflict	5	0	0.0
discarded	5	1	0.2
Top-20			
anchors	20	1	0.05
hard	20	1	0.05
conflict	20	0	0.05
discarded	10	1	0.1
Mean +/- stdDev			
anchors	223	3	0.02
hard	490	26	0.05
conflict	12	0	0.00
discarded	4	1	0.25
IQR			
anchors	63	2	0.03
hard	0	0	0.0
conflict	0	0	0.00
discarded	0	0	0.0

Table 5.19: Detecting false mappings and map discardings on anatomy

5.4 Semantic similarity of ontologies

We define a representative vector for an ontology as the sum of the vectors of all the nodes of the ontology. Only class embeddings are used [2] . As an estimation of the semantic similarity of two vectors, we jointly embed the two ontologies in the same vector space together with all gold standard mappings. We include the URIs only in the document and train with word2vec, Skip-gram. Add the vectors for each ontology and use the cosine similarity on the representative vectors. Using embeddings of size 100, we get the following similarities:

First onto	Second onto	similarity
ekaw	ekaw2	0.998
cmt	ekaw	0.907
cmt	edas	0.850
cmt	iasted	0.754
cmt	sigkdd	0.951
ekaw	sigkdd	0.812
ekaw	iasted	0.873
edas	ekaw	0.912
edas	iasted	0.841
edas	sigkdd	0.777
iasted	sigkdd	0.820
mouse	human	0.986

Table 5.20: Semantic similarity between the ontologies in the conference and anatomy track

5.5 Scalability

All test were performed on a personal laptop running an Ubuntu Linux system with 4-4-0 kernel, 12GB of RAM and a 2,3 GHz dual core (4 virtual cores with hyperthreading) Intel I5 CPU.

For the EKAW ontology, the process of creating the walk document takes about 2-3 seconds with 100 walks per node and a depth of 40.

5.5.1 Anatomy

The human ontology from the OAEI track has 3,304 classes. Creating the walks with depth 40, and 50 walks per class, takes about 25 seconds on a personal computer. By caching the edge collections in a map, we reduce the time to 15 seconds, suggesting a speedup of 40 %. While running with this

[2] See Section 4.5 on page 63 for a discussion of the role of OWL 2 properties in OWL2Vec.

ontology the memory consumption of Java (using top) is at about 12 %, this is about 1.4GB.

The projection takes only about 4 seconds, and the creation of the node graph takes about one second.

5.5.2 Largebio

The NCI-ontology that is part of the NCI-SNOMED small fragments track has 23,958 classes. The creation of the node graph took 5 seconds, but the adding of the edges took some time as it contains 29,974 edges. It uses about 17 % of the memory while running (with the caching of edge weights), and the creation of the walks with a depth of 40 and 100 walks per node in total took 104 seconds.

The largest ontology in the largebio track is the SNOMED large fragment. It contains 122,464 classes. The node graph was initialized in 19 seconds, and it added 211,527 edges to the graph. Creating the same walks, depth 40 and 100 walks it used 27 % of memory. It took about 22 minutes to create the walks, and the document that was created was 26 GB.

On the full OAEI largebio track with NCI - FMA matching. The total number of classes in the combined ontology is 145,712. For the first trial, we limited the number of classes for walks to 100,000. We created the walks document with depth 40, and 50 random walks for each class. The load was at about 379 % of CPU usage (4 virtual cores) and 27.9 % RAM usage (of 12 GB) without the caching of the node weights. The resulting walks document was at about 12 GB. Training the embeddings with gensim word2vec using Skip-gram, 50 dimensions, 25 negative samplings on this document was a lot harder on the system than generating the walks. It used about 90 % of RAM and 90 % of swap-memory. In the end, the time to create the walk was 30 minutes and the time to train the model was about 10 minutes. The reason it began to use swap space was that the reading of the 12 GB document that can not fit into memory.

We then generated walks with a depth of 8, and 10 random walks for each class using the same system, but with no class limit. The walks generation took 11 minutes, and the document size was 1.1GB. The training was done with a dimensionality of 20, CBOW, window 5, and 5 negative samples took only 33 seconds.

5.5.3 Gene ontology

The Gene Ontology (GO) has about 50 000 classes. It runs well and creates the document with a walk depth of 40 steps and 50 walks per class, including reasoning with ELK and graph projection in about 6 minutes on a personal computer creating a document of about 3.7GB. With the caching of the edge collections, the time reduces to about 4.1 minutes. The graph projection with reasoning finished in about 30 seconds.

Chapter 6

Discussion

6.1 Pretrained embeddings

The main focus in the thesis has not been the use of pretrained word embeddings as the usefulness of such embeddings already has been studied extensively in [58, 97]. It is, however, worth mentioning that using pretrained word embeddings have several advantages. First, it does not require any anchors, and as many ontologies have a similar vocabulary, it can do a good job. It seems to be an improvement over simple lexical similarity measures because it inherently includes semantically similar words. It is, however, still necessary to decide a cutoff value, which, if set too low, will include too many matches and set too high will exclude good matches. Our pretrained vectors were simple, and one can expect better result using high quality word vectors and vectors trained for a specific domain.

The anatomy track had a low score compared to the others. The low precision is due to the averaging of the word vectors as described below. The low recall could be attributed to that the training corpus did not have the technical vocabulary.

Another thing that could improve this strategy is to use a better way to represent multiple words as a vector. In this study, we use averaging. A problem with this strategy is that it gives equal weight to each word. Therefore two sets of words that include several common words would give a too high similarity score. However, better and more sophisticated strategies are available. We leave this as a challenge.

Limitations for the pretrained embeddings include that, having one vector for each word, does not account for the case where a word can have different meanings in different contexts, for example, homonyms. The word embeddings should be considered one of many tools and part of a larger workflow, as they ignore, for example, the ontology's satisfiability and structural information.

6.2 Walk strategies

The "vanilla" RDF2Vec is not possible to use to embed an OWL-ontology. First of all, it is supposed to n walks for each entity e such that e is of type

owl:Thing in the dataset. For ontologies, it makes more sense to speak of n walks for each c of type owl:Class. RDF2Vec can be used by first projecting the ontology to a navigable RDF-graph. Even then it has overall poor performance. It is slow and the embeddings, although they contain useful information, are not very good. The way it generates the walks makes it challenging to create long walks, and the walks contain much noise.

One thing that creates noise is that RDF2Vec performs reasoning on the projected graph. To perform reasoning on the projected graph makes no sense. It is built to be navigable, and it already conserves all the important relationships. The graph ends up with having many entries stating that a class is owl:subclass of owl:Thing. Consequently, the walks often do not do anything more than going from some class directly to owl:Thing. Then it can go past owl:Thing via some object property and back to owl:Thing again. These short walks through owl:Thing draw many unrelated elements into proximity, thus creates noise.

Still, **RDF2Vec** is not without value. It gives many reasonable suggestions, but it confuses similar structures. If A and B are both subclasses of C, it will provide answers such that $\texttt{A} \equiv \texttt{B}$ or $\texttt{A} \equiv \texttt{C}$. An example result is provided in Table D.2 on page 137. It is not difficult to imagine why this can happen. Word2vec will embed words that have a similar meaning very close to each other. Since C could replace A in any setting, and it can replace B the same way, they are likely to be embedded very close together.

Additionally, **RDF2Vec** tends to confuse terms that, in the ontology, are declared as disjoint with the same class. For example, if A and B both are disjoint with C, it tends to produce similar vectors for A and B. The amount of information about each class in the ontology is limited, so all axioms will significantly influence the embeddings. Any axiom on the form D R C where there is another axiom E R C will tend to move the embeddings for D and E closer because they appear in the same context.

Although **OWL2Vec structural** still suffers from the confusion of structurally similar elements, it shows better performance than **RDF2Vec** in all the experiments.

OWL2Vec synonyms have better performance than the two others. It can leverage textual information to distinguish siblings and parent classes.

The **OWL2Vec+**-walks generally shows better performance than all the other walk types except when the candidate finder use some way to disambiguate using lexical embeddings as well as the structural ones. It is the only system that can achieve a recall that is higher than the fraction of anchors it receives as input on the cmt-ekaw matching. The reason for the improved performance is that it combines walks going from the top towards the bottom with walks that come from the bottom and goes toward the top, and also that it can keep walking up and down the hierarchy and by object property-edges without stopping at the edge of the graph. That way one can get much longer walks with related classes and more diverse walks including not only parent-child relationships in the window but also sibling relationships. It does not confuse similar entities as much as with the **RDF2Vec** and **OWL2Vec structural** systems. The structurally similar classes are still very similar in terms of cosine similarity, but the best match

is usually correct. This improvement is because it can do more training on the local neighborhood and capture more of the small differences.

On the matching tests, there is a tendency to create walks that go back and forward between the anchors. This tendency adds value in emphasizing the relationship between these, but it could potentially decrease the value of the embeddings by making the anchors more important than the rest of the graph. The **OWL2Vec subClass** system falls into this trap. When there is no other way to keep walking it keeps repeating the anchors. The **OWL2Vec+** system can also do this if the p-value is low. With a high p-value, it does not return while there exists another possible path. Thus it can avoid the loop entirely.

The inclusion of the synonyms close to the URIs in the same document tends to increase performance. In particular when we have few anchors. It is because the unequal URIs appear in the context of the same lexical words. This creates more similar vectors for the URIs.

6.3 Matching strategies

While the focus of this thesis is not how to do ontology matching, but instead on ontology embeddings. We have created a few different strategies to get results. All of which are very simple. They are too simple to be scalable and do matching on the large ontologies. Even though we create the walks and train the embeddings, matching the ontologies using the cartesian product is not feasible for large ontologies.

The best match strategy is useful in many cases. The effort to disambiguate the best matches by using textual information has little effect on the **OWL2Vec synonyms**. It does, however, improve the performance of **OWL2Vec structural**. If we reduce the structural equality threshold, it tends to improve precision. Lowering this threshold means, however, that we rely more on the lexical equality and less on the embeddings.

The *all relations*-alignment strategy can improve recall. It is not useful as a stand alone strategy, because the effect on precision is detrimental. It is useful to see if the mappings could be found.

The *transformation matrix* shows high precision, and the recall is more or less equal to the anchors provided. We get this result because the model is trained directly using the anchors. It will most likely find the anchors and only the anchors with a high equality threshold. The *transformation matrix* is capable of finding structurally similar classes as well, but then the cutoff must be reduced, this will also reduce the precision.

6.4 Embeddings systems and OWL 2 embeddings

We have discussed several systems capable of creating embedding from text in the Neural language models section (Section 3.2 on page 29). The tools have some very different characteristics. Most notably, the word2vec-model [51] considers each word a separate, atomic unit and creates the embedding for the word while `fastText` includes not only the context

information but is capable of creating embeddings for character n-grams. Thus it takes into account lexical information of what is embedded. If we are embedding a URI, this means that it could be grouping the entities with common namespaces closer together. If we are using synonyms or uri_parts to represent the entities, this could turn out to be a great asset as it would take into account both lexical similarity and structural similarity at the same time. On ontologies that are similar both structurally and lexically, these embeddings are capable of finding many correspondences.

It does also mean that our embeddings would not be purely structural as it could be using word2vec and would make no sense in some situations (for example by embedding URIs that do not contain information).

While StarSpace, is shown to be useful for embedding knowledge graphs [3], it was not found to add anything useful to ontology embeddings using random walks. It can create word vectors, but the results we got was not on par with the embeddings from word2vec and `fastText`. How to use StarSpace effectively for OWL 2 embeddings is, however, an interesting topic.

Using the `fastText` system when including synonyms did show an improvement on the ontology matching tasks. It is particularly useful when including the synonyms in the same document as the URIs. This result was expected as it emphasizes lexical similarity. It can, however, also introduce false positives.

We tried to use a pretrained word vector model to initialize the vectors of OWL2Vec, but this was not successful. First of all, it is difficult to determine how much of the vectors are actually from the pretrained vectors and how much is from the additional training on the ontology.

Another drawback of using pretrained vectors to initialize the vectors is that we have to use the words of each URI and the words of any synonym in the ontology to get an overlapping vocabulary between the two instead of being able to embed the URIs or URI-parts as they appear in the ontology. Using the words and synonyms of an ontology, return the matchings of the words that are lexically equal. This can not be attributed to the vectors, just that the same word has one vector. We are therefore not able to say anything about the benefit of using pretrained with extra training on the ontology.

6.4.1 Ekaw-ekaw vs. ekaw-cmt

It is easier to find correspondences using structural embeddings between two ontologies when they are structurally similar. We observe this effect in Table 5.4 on page 77, where several systems achieve both high precision and recall using 60 % of the possible anchors. Another thing to take into consideration is that between ekaw and ekaw there are many more anchors to add because all classes have a match. Between cmt and ekaw, however, there are only eight matches in the gold standard.

We observe that the correlation between matching recall and the number of anchors is strong. The correlation is higher when matching ekaw-cmt because there are fewer structural and string similarities to work with, and fewer anchors. Consequently, each anchor is essential. All have a

Pearson correlation of above 0.9 between anchors and recall except for **RDF2Vec** and **OWL2Vec 2doc**. For the **RDF2Vec** it is because of poor performance. The **OWL2Vec 2doc** system seems to reach top performance quickly. The limiting factor is the performance of the lexical embeddings on disambiguating the set of potential mappings. By changing how the alignment system works, this could probably be improved.

6.4.2 Several ontologies

When the system provides anchors from both ontologies to a third ontology, the system can use these as anchors for matching. These anchors are extended to the ontologies we want to match by reasoning when creating the projection graph. The performance of the multi ontology matching system is then virtually unaffected by the addition of ekaw-cmt anchors. This is an example of using mediation ontologies for matching.

6.5 Improving LogMap

It is challenging to find additional mappings to the ones provided by LogMap. First, LogMap already has high recall and precision on the Anatomy and the small largeBio track.

We show that it is possible to improve the chance of finding some false negatives and false positives using ontology embeddings together with statistical methods or by taking the ones with the highest overall cosine similarity among the discarded mappings and lowest overall cosine among the anchors. More work is, however, needed to understand how this strategy can be used to improve state-of-the-art ontology matching systems.

6.6 Scalability

The **OWL2Vec+** system is designed to handle large ontologies. By holding the entire graph in memory, using parallelization, and caching names, maps, and structures, it can efficiently embed large ontologies.

OWL2Vec+ create the walks for the gene ontology in little more than 4 minutes using long and many walks, and the merge between the walks for NCI and FMA with anchors finished in 30 minutes. While it is possible to create walks and embed very large ontologies, it is, however, preferable to limit the size and number of walks on these ontologies because the target document blows up to several GB. These documents are larger than the ram size on a regular computer. It is, however, possible to train these documents as gensim is designed with the idea of large documents and does not require the entire document to be in memory [79].

The system has been tested on small and large ontologies from the OAEI and can create embeddings from even the union of the largest largebio ontologies in a relatively short time. One has to give attention to the parameter settings, however, in order to not create gigantic walk documents or run out of heap space. It is even possible to create massive documents

even from the small ontologies by generating large numbers of very long walks. One would, however, not be doing more than repeating the same statements over and over.

6.7 Semantic similarity

Representing an ontology by taking the sum of all the vectors of all the classes gives a vector that should represent the general idea of the ontology classes. It represents the purpose or the focus area of the ontology. It does not take into account differences in size, properties, or lexical differences. There are many things one could include here to get a better estimate of the similarity between two ontologies. For example the size, the hierarchy, and the lexical similarity. Still, the cosine of two ontologies gives an exciting similarity score.

Semantic similarity of fragments of ontologies could be defined and compared the same way.

6.8 Object properties and datatype properties

The systems for embedding ontologies presented in this thesis focus exclusively on creating embeddings of classes and are not in any way optimized for the embeddings of properties. It can include property names in the walks, but this is because the inclusion of edge names can potentially improve the performance of the class embeddings system. It is not able to include datatype properties. Embedding properties, is, however, an exciting future research topic.

6.9 Analytical tasks

The embeddings show exciting results on the clustering task. The clusters found using **OWL2Vec+** are excellent and, in most cases, the clusters represent what one naturally would consider related concepts. The PCA images show how related concepts tend to cluster together.

6.10 Hyperparameters

The amount of training and increased dimensionality generally improves the embeddings. For the clustering tasks, the results are miserable for low dimensional embeddings or little training. The way to increase training is either to output more and longer walks or to increase epoch count.

For prediction tasks and ontology alignment, it can be the other way around. Having too much training and too high dimensionality makes it challenging to detect possible candidates. Seemingly, for tasks that need a high degree of generalization, the dimensions should not be too high. For a precision task, however, the embeddings must be more detailed.

Setting the p very low together with high `subClassOf`-probability on alignment tasks can give outstanding results when using many anchors. These results are, however, not due to the systems the ability to embed ontology structure, but rather its ability to find the anchors that were inserted into the ontology. The anchors will have `subClassOf`-relationships both ways, and when encouraging to go to repeat the last node, it tends to produce long walks consisting of the two classes of the anchors only. These walks make the vectors of the anchors very similar, and they are easily found. It is therefore essential to use the parameters in such a way that it, not only finds the anchors that were provided but also finds other structurally and lexically related classes.

The edge weight settings greatly influence the quality of the embeddings. We find that favoring the `subClassOf-` over the `superClassOf`-edge and give object properties a low weight (or even ignore the object properties entirely) gives the best results. A more comprehensive discussion of the edge weights is found in Subsection 4.3.3 on page 57.

Part III

Conclusion and future work

Chapter 7

Conclusion

In this thesis, we have conducted an exploratory study. We have demonstrated that OWL 2 embeddings are useful, and hope, with this work, to inspire further interests and research in this direction.

The ontology embeddings created by OWL2Vec are very promising and show an improvement with respect to the state of the art solutions.

Jointly embedding two (or more) ontologies in the same vector space and align them with anchor mappings is shown to be useful for finding new mappings. The strategy is also shown to provide relevant, and potentially useful, information on the structural similarity of OWL 2-ontologies that can be exploited by ontology matching systems as an additional similarity measure.

The downside of using such embeddings for ontology alignment is the need for anchors, but the anchors can, as shown, easily be found using existing ontology alignment tools such as LogMap 2.

7.1 Future work

We have identified several areas that need further investigation, and there are probably many more. The reader is encouraged to implement the embeddings into existing ontology matching tools, and to design strategies to leverage the new structural (and potentially lexical) similarity measures.

To improve state-of-the-art ontology matching systems using ontology embeddings, one could use the cosine score as an additional measure of the similarity between two concepts. The similarity score could then be used together with the system's other similarity scores to increase or decrease the likelihood of keeping or discarding a potential mapping.

OWL2Vec can be used as a novel strategy to cluster one ontology, but ontology clustering is also useful as part of an ontology alignment workflow (as in [35]).

Improve the embeddings for the object properties could be useful. One would have to take advantage of the property hierarchy, restrictions, and possibly lexical information. The inclusion of data types and data properties could potentially improve the embeddings of classes. Embeddings of two classes would be closer if they were to have the same data type properties

with the same data types. One would have to figure out a way to achieve this.

The ontology can be seen, at least in part, as a hierarchy of more and more specific concepts. It should be possible to exploit this concept more than it is done by using the walk. The embeddings should be such that the sum of all the siblings in a branch should be close to the embedding of the parent concept while maintaining the property that the sibling's vectors also should be quite similar. However, if the vector of one concept should be closest to its sibling or its parent, that would depend on the application.

Another exciting concept to include in the walks is the possibility of walking directly from one sibling to another. Even though the siblings are not directly connected by any edge, they are very related.

Expanding on the idea that two ontologies could be compared by adding all the vectors, it could be useful to implement a way to measure the similarity between two ontology clusters. This can be relevant for the blocking of ontologies to reduce the search space. The system could be able to give two different scores, one for structural equality and one for lexical equality. The ontology similarity measure could be expanded to include other information such as lexical information, size, and hierarchical complexity to give a complete picture.

Investigating and implementing better strategies for combining word vectors when comparing concepts with several words will improve systems using lexical embeddings.

The ontology embeddings could be useful also for entity matching. This was shown to be the case in [90].

It could also be interesting to see how the results would be if we used a global approach as proposed in [13]. Novel strategies using StarSpace to create OWL 2 embeddings without doing walks could also be an exciting future research topic.

Bibliography

[1] Manel Achichi *et al.* "Results of the Ontology Alignment Evaluation Initiative 2016". In: *CEUR Workshop Proceedings*. Vol. 1766. RWTH, 2016, pp. 73–129.

[2] Manel Achichi *et al.* "Results of the Ontology Alignment Evaluation Initiative 2017". In: *CEUR Workshop Proceedings*. Vol. 2032. RWTH, 2017, pp. 61–113.

[3] Asan Agibetov and Matthias Samwald. "Fast and Scalable Learning of Neuro-Symbolic Representations of Biomedical Knowledge". en. In: *arXiv:1804.11105 [cs]* (Apr. 2018). arXiv: 1804.11105 [cs].

[4] Alsayed Algergawy *et al.* "Results of the Ontology Alignment Evaluation Initiative 2018". In: *13th International Workshop on Ontology Matching*. 2018, pp. 76–116.

[5] *Apache Jena* -. URL: https://jena.apache.org/.

[6] C. Batini, M. Lenzerini, and S. B. Navathe. "A Comparative Analysis of Methodologies for Database Schema Integration". en. In: *ACM Computing Surveys* 18.4 (Dec. 1986), pp. 323–364. ISSN: 03600300. DOI: 10.1145/27633.27634.

[7] Yoshua Bengio, Réjean Ducharme, and Pascal Vincent. "A Neural Probabilistic Language Model". In: *Neural Information Processing Systems (NIPS)*. 2000, pp. 932–938.

[8] Tim Berners-Lee, James Hendler, and Ora Lassila. "The Semantic Web". In: *Scientific american* 284.5 (2001), pp. 34–43.

[9] Piotr Bojanowski *et al.* "Enriching Word Vectors with Subword Information". en. In: *arXiv:1607.04606 [cs]* (July 2016). arXiv: 1607.04606 [cs].

[10] Antoine Bordes *et al.* "Translating Embeddings for Modeling Multi-relational Data". In: *27th Annual Conference on Neural Information Processing Systems (NIPS)*. 2013, pp. 2787–2795.

[11] Andrew ButterfieldAndrew Butterfield and Gerard Ekembe Ngondi-Gerard Ekembe Ngondi. "Multivariate Analysis". en. In: *A Dictionary of Computer Science*. Ed. by Andrew Butterfield and Gerard Ekembe Ngondi. Oxford University Press, Jan. 2016. ISBN: 978-0-19-968897-5.

[12] Hongyun Cai, Vincent W. Zheng, and Kevin Chen-Chuan Chang. "A Comprehensive Survey of Graph Embedding: Problems, Techniques and Applications". en. In: *arXiv:1709.07604 [cs]* (Sept. 2017). arXiv: 1709.07604 [cs].

[13] Michael Cochez *et al.* "Global Rdf Vector Space Embeddings". In: *International Semantic Web Conference*. Springer, 2017, pp. 190–207.

[14] "Cosine Similarity". en. In: *Wikipedia* (Dec. 2018). Page Version ID: 876126121.

[15] *DBpedia*. URL: http://wiki.dbpedia.org/about (visited on 02/06/2018).

[16] AnHai Doan *et al.* "Learning to Match Ontologies on the Semantic Web". In: *The VLDB Journal The International Journal on Very Large Data Bases* 12.4 (Nov. 2003), pp. 303–319. ISSN: 1066-8888, 0949-877X. DOI: 10.1007/s00778-003-0104-2.

[17] Bob DuCharme. *Semantic Web Semantics vs. Vector Embedding Machine Learning Semantics - Bobdc.Blog*. URL: http://www.snee.com/bobdc.blog/2016/09/semantic-web-semantics-vs-vect.html.

[18] *Embed - Oxford Reference*. en. URL: http://www.oxfordreference.com/view/10.1093/acref/9780199571123.001.0001/m_en_gb0262220.

[19] *Ernesto Jimenez-Ruiz / Ontology-Services-Toolkit*. en. URL: https://gitlab.com/ernesto.jimenez.ruiz/ontology-services-toolkit.

[20] Floriana Esposito, Nicola Fanizzi, and Claudia d'Amato. "Recovering Uncertain Mappings Through Structural Validation and Aggregation with the MoTo System". In: *Proceedings of the 2010 ACM Symposium on Applied Computing*. SAC '10. Sierre, Switzerland: ACM, 2010, pp. 1428–1432. ISBN: 978-1-60558-639-7. DOI: 10.1145/1774088.1774390. URL: http://doi.acm.org/10.1145/1774088.1774390.

[21] *Foundational Model of Anatomy | Structural Informatics Group*. URL: http://si.washington.edu/projects/fma (visited on 02/06/2018).

[22] Fabien Gandon, Marta Sabou, and Harald Sack. "Weaving a Web of Linked Resources". In: *Semantic Web* 8.6 (Aug. 7, 2017). Ed. by Fabien Gandon, Marta Sabou, and Harald Sack, pp. 767–772. ISSN: 22104968, 15700844. DOI: 10.3233/SW-170284. URL: http://www.medra.org/servlet/aliasResolver?alias=iospress&doi=10.3233/SW-170284 (visited on 02/07/2018).

[23] *Gensim: Topic Modelling for Humans*. en. URL: https://radimrehurek.com/gensim/.

[24] Birte Glimm *et al.* "HermiT: An OWL 2 Reasoner". In: *J. Autom. Reasoning* 53.3 (2014), pp. 245–269.

[25] Palash Goyal and Emilio Ferrara. "Graph Embedding Techniques, Applications, and Performance: A Survey". en. In: *Knowledge-Based Systems* 151 (July 2018), pp. 78–94. ISSN: 09507051. DOI: 10.1016/j.knosys.2018.03.022. arXiv: 1705.02801.

[26] Aditya Grover and Jure Leskovec. "Node2vec: Scalable Feature Learning for Networks". en. In: *arXiv:1607.00653 [cs, stat]* (July 2016). arXiv: 1607.00653 [cs, stat].

[27] Thomas R. Gruber. "Toward Principles for the Design of Ontologies Used for Knowledge Sharing". In: *International Journal of Human-Computer Studies* 43.5 (1995), pp. 907–928. ISSN: 1071-5819. DOI: https://doi.org/10.1006/ijhc.1995.1081.

[28] Tom Gruber. "Ontology". en. In: *Encyclopedia of Database Systems*. Ed. by LING LIU and M. TAMER ÖZSU. Boston, MA: Springer US, 2009, pp. 1963–1965. ISBN: 978-0-387-39940-9. DOI: 10.1007/978-0-387-39940-9_1318.

[29] Terry F Hayamizu *et al.* "The Adult Mouse Anatomical Dictionary: A Tool for Annotating and Integrating Data". en. In: *Genome Biology* (2005), p. 8.

[30] Matthew Horridge and Sean Bechhofer. "The OWL API: A Java API for OWL ontologies". In: *Semantic Web* 2.1 (2011), pp. 11–21.

[31] Eclipse Foundation Inc. *The Platform for Open Innovation and Collaboration | The Eclipse Foundation*. en. URL: https://www.eclipse.org/.

[32] *Inference - W3C*. URL: https://www.w3.org/standards/semanticweb/inference (visited on 01/31/2018).

[33] *Introducing the Knowledge Graph: Things, Not Strings*. en. URL: https://googleblog.blogspot.com/2012/05/introducing-knowledge-graph-things-not.html (visited on 04/23/2019).

[34] Jerome Euzenat and Pavel Shvaiko. *Ontology Matching*. en. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013. ISBN: 978-3-642-38720-3 978-3-642-38721-0. DOI: 10.1007/978-3-642-38721-0.

[35] Ernesto Jimenez-Ruiz *et al.* "Breaking-down the Ontology Alignment Task with a Lexical Index and Neural Embeddings". en. In: *arXiv:1805.12402 [cs]* (May 2018). arXiv: 1805.12402 [cs].

[36] Ernesto Jiménez-Ruiz *et al.* "Large-scale Interactive Ontology Matching: Algorithms and Implementation". In: *20th European Conference on Artificial Intelligence (ECAI)*. 2012, pp. 444–449.

[37] Ernesto Jiménez-Ruiz *et al.* "Logic-based assessment of the compatibility of UMLS ontology sources". In: *J. Biomedical Semantics* 2.S-1 (2011), S2.

[38] Armand Joulin *et al.* "Bag of Tricks for Efficient Text Classification". en. In: Association for Computational Linguistics, 2017, pp. 427–431. DOI: 10.18653/v1/E17-2068.

[39] *JWS Special Issue on Knowledge Graphs/En – International Center for Computational Logic*. URL: https://iccl.inf.tu-dresden.de/web/JWS_special_issue_on_Knowledge_Graphs.

[40] Yevgeny Kazakov, Markus Krötzsch, and Frantisek Simancik. "The Incredible ELK - From Polynomial Procedures to Efficient Reasoning with EL Ontologies". In: *J. Autom. Reasoning* 53.1 (2014), pp. 1–61.

[41] Tom Kenter, Alexey Borisov, and Maarten de Rijke. "Siamese CBOW: Optimizing Word Embeddings for Sentence Representations". en. In: Association for Computational Linguistics, 2016, pp. 941–951. DOI: 10.18653/v1/P16-1089.

[42] M. T. Khadir, A. Djeddai, and W. Djeddi. "XMap++: A Novel Semantic Approach for Alignment of OWL-Full Ontologies Based on Semantic Relationship Using WordNet". In: *Innovation in Information & Communication Technology (ISIICT), 2011 Fourth International Symposium On*. IEEE, 2011, pp. 13–18.

[43] Markus Krötzsch, Frantisek Simancik, and Ian Horrocks. "A Description Logic Primer". en. In: *arXiv:1201.4089 [cs]* (Jan. 2012). arXiv: 1201.4089 [cs].

[44] Wen-Syan Li and Chris Clifton. "Semantic Integration in Heterogeneous Databases Using Neural Networks". In: *20th International Conference on Very Large Data Bases (VLDB)*. 1994, pp. 1–12.

[45] *Library for Fast Text Representation and Classification.: Facebookresearch/-fastText*. Facebook Research, Aug. 2018.

[46] Nicolas Matentzoglu and Ignazio Palmisano. *An Introduction to the OWL API*. URL: http://syllabus.cs.manchester.ac.uk/pgt/2018/COMP62342/introduction-owl-api-msc.pdf.

[47] John Mccarthy. "Circumscription - A Form of Non-Monotonic Reasoning". eng. In: (1980).

[48] Thomas Mikolov. *De-Obfuscated Python + Question – Google Groups*. URL: https://groups.google.com/forum/#!searchin/word2vec-toolkit/c-bow/word2vec-toolkit/NLvYXU99cAM/E5ld8LcDxlAJ.

[49] Tomas Mikolov, Quoc V. Le, and Ilya Sutskever. "Exploiting Similarities among Languages for Machine Translation". en. In: *arXiv:1309.4168 [cs]* (Sept. 2013). arXiv: 1309.4168 [cs].

[50] Tomas Mikolov *et al.* "Distributed Representations of Words and Phrases and their Compositionality". In: *27th Annual Conference on Neural Information Processing Systems (NIPS)*. 2013, pp. 3111–3119.

[51] Tomas Mikolov *et al.* "Efficient Estimation of Word Representations in Vector Space". In: *arXiv:1301.3781 [cs]* (Jan. 2013). arXiv: 1301.3781 [cs].

[52] Mark A. Musen. "The Protégé Project: A Look Back and a Look Forward". en. In: *AI Matters* 1.4 (June 2015), pp. 4–12. ISSN: 23723483. DOI: 10.1145/2757001.2757003.

[53] *NCI Thesaurus*. URL: https://ncit.nci.nih.gov/ncitbrowser/ (visited on 02/06/2018).

[54] Yavor Nenov *et al.* "RDFox: A Highly-Scalable RDF Store". en. In: *The Semantic Web - ISWC 2015*. Ed. by Marcelo Arenas *et al.* Vol. 9367. Cham: Springer International Publishing, 2015, pp. 3–20. ISBN: 978-3-319-25009-0 978-3-319-25010-6. DOI: 10.1007/978-3-319-25010-6_1.

[55] Markus Nentwig *et al.* "A Survey of Current Link Discovery Frameworks". In: *Semantic Web* 8.3 (Jan. 1, 2017), pp. 419–436. ISSN: 1570-0844. DOI: 10.3233/SW-150210. URL: https://content.iospress.com/articles/semantic-web/sw210 (visited on 01/12/2018).

[56] DuyHoa Ngo and Zohra Bellahsene. "YAM++: A Multi-Strategy Based Approach for Ontology Matching Task". In: *International Conference on Knowledge Engineering and Knowledge Management*. Springer, 2012, pp. 421–425.

[57] Maximilian Nickel, Lorenzo Rosasco, and Tomaso Poggio. "Holographic Embeddings of Knowledge Graphs". en. In: *arXiv:1510.04935 [cs, stat]* (Oct. 2015). arXiv: 1510.04935 [cs, stat].

[58] Ikechukwu Nkisi-Orji *et al.* "Ontology Alignment Based on Word Embedding and Random Forest Classification". In: *European Conference on Machine Learning and Knowledge Discovery in Databases (ECML/PKDD)*. 2018, pp. 557–572.

[59] *OAEI Largebio track*. URL: http://www.cs.ox.ac.uk/isg/projects/SEALS/oaei/.

[60] *Ontologies - W3C*. URL: https://www.w3.org/standards/semanticweb/ontology (visited on 01/31/2018).

[61] *Ontology Alignment Evaluation Initiative*. URL: http://oaei.ontologymatching.org/2018.

[62] *Ontology | Origin and Meaning of Ontology by Online Etymology Dictionary*. URL: https://www.etymonline.com/word/ontology (visited on 01/30/2018).

[63] *OWL 2 Web Ontology Language Manchester Syntax (Second Edition)*. URL: https://www.w3.org/TR/owl2-manchester-syntax/.

[64] *OWL 2 Web Ontology Language XML Serialization (Second Edition)*. URL: https://www.w3.org/TR/owl2-xml-serialization/.

[65] *OWL - Semantic Web Standards*. URL: https://www.w3.org/OWL/ (visited on 02/07/2018).

[66] *OWL Web Ontology Language Guide*. URL: https://www.w3.org/TR/2004/REC-owl-guide-20040210/.

[67] Cheolsoo Park, Clive Cheong Took, and Joon-Kyung Seong. "Machine Learning in Biomedical Engineering". en. In: *Biomedical Engineering Letters* 8.1 (Feb. 2018), pp. 1–3. ISSN: 2093-9868, 2093-985X. DOI: 10.1007/s13534-018-0058-3.

[68] Sebastian Rudolph Pascal Hitzler Markus Krotzsch. *Foundations of Semantic Web Technologies*. 2010.

[69] "Pearson Correlation Coefficient". en. In: *Wikipedia* (Feb. 2019). Page Version ID: 883889256.

[70] F. Pedregosa *et al.* "Scikit-learn: Machine Learning in Python". In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.

[71] Jeffrey Pennington, Richard Socher, and Christopher Manning. "Glove: Global Vectors for Word Representation". en. In: Association for Computational Linguistics, 2014, pp. 1532–1543. DOI: 10.3115/v1/D14-1162.

[72] Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. "DeepWalk: Online Learning of Social Representations". en. In: *arXiv:1403.6652 [cs]* (2014), pp. 701–710. DOI: 10.1145/2623330.2623732. arXiv: 1403.6652 [cs].

[73] *Protégé*. URL: https://protege.stanford.edu/.

[74] *RDF 1.1 Turtle*. URL: https://www.w3.org/TR/turtle/.

[75] *RDF 1.1 XML Syntax*. URL: https://www.w3.org/TR/rdf-syntax-grammar/.

[76] *RDF Schema 1.1*. URL: https://www.w3.org/TR/rdf-schema/.

[77] *RDF - Semantic Web Standards*. URL: https://www.w3.org/RDF/ (visited on 02/07/2018).

[78] *RDF2Vec source code*. URL: http://data.dws.informatik.uni-mannheim.de/rdf2vec/code/.

[79] Radim Řehůřek and Petr Sojka. "Software Framework for Topic Modelling with Large Corpora". English. In: *Proceedings of the LREC 2010 Workshop on New Challenges for NLP Frameworks*. http://is.muni.cz/publication/884893/en. Valletta, Malta: ELRA, May 2010, pp. 45–50.

[80] Petar Ristoski and Heiko Paulheim. "RDF2Vec: RDF Graph Embeddings for Data Mining". In: *The Semantic Web – ISWC 2016*. Ed. by Paul Groth *et al.* Vol. 9981. Cham: Springer International Publishing, 2016, pp. 498–514. ISBN: 978-3-319-46522-7 978-3-319-46523-4. DOI: 10.1007/978-3-319-46523-4_30.

[81] Miguel Ángel Rodríguez-García and Robert Hoehndorf. "Inferring Ontology Graph Structures Using OWL Reasoning". en. In: *BMC Bioinformatics* 19.1 (Dec. 2018). ISSN: 1471-2105. DOI: 10.1186/s12859-017-1999-8.

[82] Ernesto Jimenez Ruiz. *Logmap-Matcher. LogMap: An Ontology Alignment and Alignment Repair System*. 2019. URL: https://github.com/ernestojimenezruiz/logmap-matcher.

[83] Manuel Salvadores *et al.* "BioPortal as a dataset of linked biomedical ontologies and terminologies in RDF". In: *Semantic Web* 4.3 (2013), pp. 277–284.

[84] A. L. Samuel. "Some Studies in Machine Learning Using the Game of Checkers". In: *IBM Journal of Research and Development* 3.3 (July 1959), pp. 210–229. ISSN: 0018-8646. DOI: 10.1147/rd.33.0210.

[85] Max Schmachtenberg, Christian Bizer, and Heiko Paulheim. "Adoption of the Linked Data Best Practices in Different Topical Domains". In: *The Semantic Web – ISWC 2014*. Ed. by Peter Mika *et al.* Vol. 8796. Cham: Springer International Publishing, 2014, pp. 245–260. ISBN: 978-3-319-11963-2 978-3-319-11964-9. DOI: 10.1007/978-3-319-11964-9_16.

[86] *Semantic | Origin and Meaning of Semantic by Online Etymology Dictionary*. URL: https://www.etymonline.com/word/semantic (visited on 01/31/2018).

[87] *Semantic Web - W3C*. URL: https://www.w3.org/standards/semanticweb/ (visited on 01/31/2018).

[88] P. Shvaiko and J. Euzenat. "Ontology Matching: State of the Art and Future Challenges". In: *IEEE Transactions on Knowledge and Data Engineering* 25.1 (Jan. 2013), pp. 158–176. ISSN: 1041-4347. DOI: 10.1109/TKDE.2011.253.

[89] *Skymind*. URL: http://skymind.ai/.

[90] Fatima Zohra Smaili, Xin Gao, and Robert Hoehndorf. "Onto2Vec: Joint Vector-Based Representation of Biological Entities and Their Ontology-Based Annotations". en. In: *Bioinformatics* 34.13 (July 2018), pp. i52–i60. ISSN: 1367-4803, 1460-2059. DOI: 10.1093/bioinformatics/bty259.

[91] Fatima Zohra Smaili, Xin Gao, and Robert Hoehndorf. "OPA2Vec: Combining Formal and Informal Content of Biomedical Ontologies to Improve Similarity-Based Prediction". en. In: *Bioinformatics* (Nov. 2018). Ed. by Jonathan Wren. ISSN: 1367-4803, 1460-2059. DOI: 10.1093/bioinformatics/bty933.

[92] *SNOMED International*. URL: https://www.snomed.org/ (visited on 02/06/2018).

[93] Richard Socher *et al.* "Reasoning With Neural Tensor Networks for Knowledge Base Completion". In: *27th Annual Conference on Neural Information Processing Systems (NIPS)*. 2013, pp. 926–934.

[94] Ahmet Soylu *et al.* "OptiqueVQS: A Visual Query System over Ontologies for Industry". en. In: *Semantic Web* 9.5 (Aug. 2018). Ed. by Freddy Lecue, pp. 627–660. ISSN: 22104968, 15700844. DOI: 10.3233/SW-180293.

[95] *StarSpace: Learning Embeddings for Classification, Retrieval and Ranking*. Aug. 2018.

[96] Maosong Sun, Yang Liu, and Jun Zhao, eds. *Chinese Computational Linguistics and Natural Language Processing Based on Naturally Annotated Big Data*. Vol. 8801. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2014. ISBN: 978-3-319-12276-2 978-3-319-12277-9. DOI: 10.1007/978-3-319-12277-9.

[97] "Ontology Matching With Word Embeddings". In: Lecture Notes in Computer Science 8801 (2014). Ed. by Maosong Sun, Yang Liu, and Jun Zhao. DOI: 10.1007/978-3-319-12277-9.

[98] Eclipse Deeplearning4j Development Team. *Eclipse Deeplearning4j: Open-source distributed deep learning for the JVM*. Apache Software Foundation, 2018. URL: http://deeplearning4j.org (visited on 02/10/2019).

[99] *Tensors in ND4J | Deeplearning4j*. URL: https://deeplearning4j.org/docs/latest/nd4j-tensor.

[100] *The Linking Open Data Cloud Diagram*. URL: http://lod-cloud.net/ (visited on 02/06/2018).

[101] Tim Berners-Lee. *WorldWideWeb: Summary*. 1991. URL: https://www.w3.org/People/Berners-Lee/1991/08/art-6484.txt.

[102] *UMLS Metathesaurus*. eng. List of Links. URL: https://www.nlm.nih.gov/research/umls/knowledge_sources/metathesaurus/.

[103] *UMLS SPECIALIST Lexicon*. URL: https://www.ncbi.nlm.nih.gov/books/NBK9680/.

[104] S. van der Walt, S. C. Colbert, and G. Varoquaux. "The NumPy Array: A Structure for Efficient Numerical Computation". In: *Computing in Science Engineering* 13.2 (Mar. 2011), pp. 22–30. ISSN: 1521-9615. DOI: 10.1109/MCSE.2011.37.

[105] *Wikidata*. URL: https://www.wikidata.org/wiki/Wikidata:Main_Page (visited on 04/23/2019).

[106] *Word Representations · fastText*. en. URL: https://fasttext.cc/index.html.

[107] *WordNet | A Lexical Database for English*. URL: https://wordnet.princeton.edu/.

[108] Ledell Wu *et al.* "StarSpace: Embed All The Things!" In: *arXiv:1709.03856 [cs]* (Sept. 2017). arXiv: 1709.03856 [cs].

[109] Neha Yadav, Anupam Yadav, and Manoj Kumar. *An Introduction to Neural Network Methods for Differential Equations*. en. SpringerBriefs in Applied Sciences and Technology. Dordrecht: Springer Netherlands, 2015. ISBN: 978-94-017-9815-0 978-94-017-9816-7. DOI: 10.1007/978-94-017-9816-7.

[110] Pinar Yanardag and S.V.N. Vishwanathan. "Deep Graph Kernels". en. In: ACM Press, 2015, pp. 1365–1374. ISBN: 978-1-4503-3664-2. DOI: 10.1145/2783258.2783417.

Appendix A

Ontology descriptions

A.1 Ekaw

A.1.1 Ekaw Object properties

Here are the object properties in the ekaw-ontology.

- authorOf
- coversTopic
- eventOnList
- hasPart
  - hasEvent
  - iverse_of_partOf_7
- hasReview
- hasReviewer
- hasUpdatedVersion
- heldIn
- listsEvent
- locationOf
- organisedBy
  - scientificallyOrganisedBy
  - technicallyOrganisedBy
- organises
  - scientificallyOrganises
  - technicallyOrganises
- paperInVolume
- paperPresentedAs
- partOf
  - partOfEvent
- presentationOfPaper
- publisherOf
- referencedIn
- reference
- reviewerOfPaper
- topicCoveredBy
- updateVersionOf
- volumeContainsPaper
- writtenBy
  - reviewWrittenBy

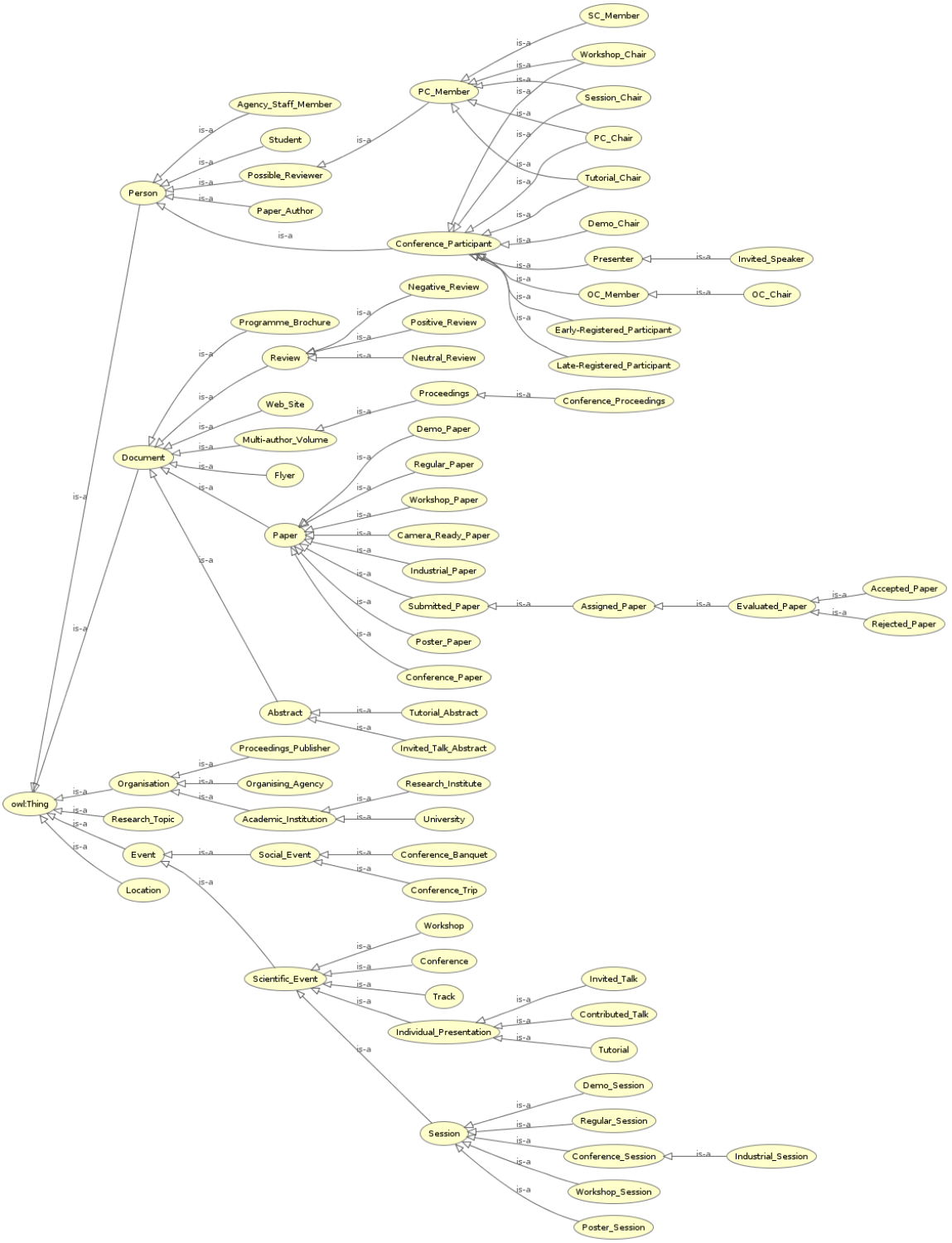

Figure A.1: The class hierarchy of the ekaw ontology using OWLViz in protegé

A.2 Cmt

A.2.1 Cmt Object Properties

- acceptedBy
- acceptPaper
- addedBy
- addProgramCommitteeMember
- adjustBid
- adjustedBy
- assignedByReviewer
- assignedTo
- assignExternalReviewer
- assignReviewer
- co-writePaper
- detailsEnteredBy
- enableVirtualMeeting
- endReview
- enterConferenceDetails
- finalizePaperAssignment
- hardcopyMailingManifestPrintedBy
- hasAuthor
- hasBeenAssigned
- hasBid
- hasCo-author
- hasConferenceMember
- hasConflictOfInterest
- hasDecision
- hasProgramCommitteeMember
- hasSubjectArea
- markConflictOfInterests
- memberOfConference
- memberOfProgramCommittee
- paperAssignmentFinalizedBy
- paperAssignmentToolsRunBy
- printHardcopyMailingManifest
- readByMeta-Reviewer
- readByReviewer
- readPaper
- rejectedBy
- rejectPaper
- reviewCriterialEnteredBy
- reviewerBiddingStartedBy
- runPaperAssignmentTools
- setMaxPapers
- startReviewerBidding
- submitPaper
- virtualMeetingEnabledBy
- writePaper
- writeReview
- writtenBy

A.2.2 Cmt Datatype properties

- acceptsHardcopySubmissions
- date
- email
- logoURL
- maxPapers
- name
- paperID
- reviewsPerPaper
- siteURL
- title

Figure A.2: The class hierarchy of the cmt ontology using OWLViz in protegé

A.3 Other class hierarchies

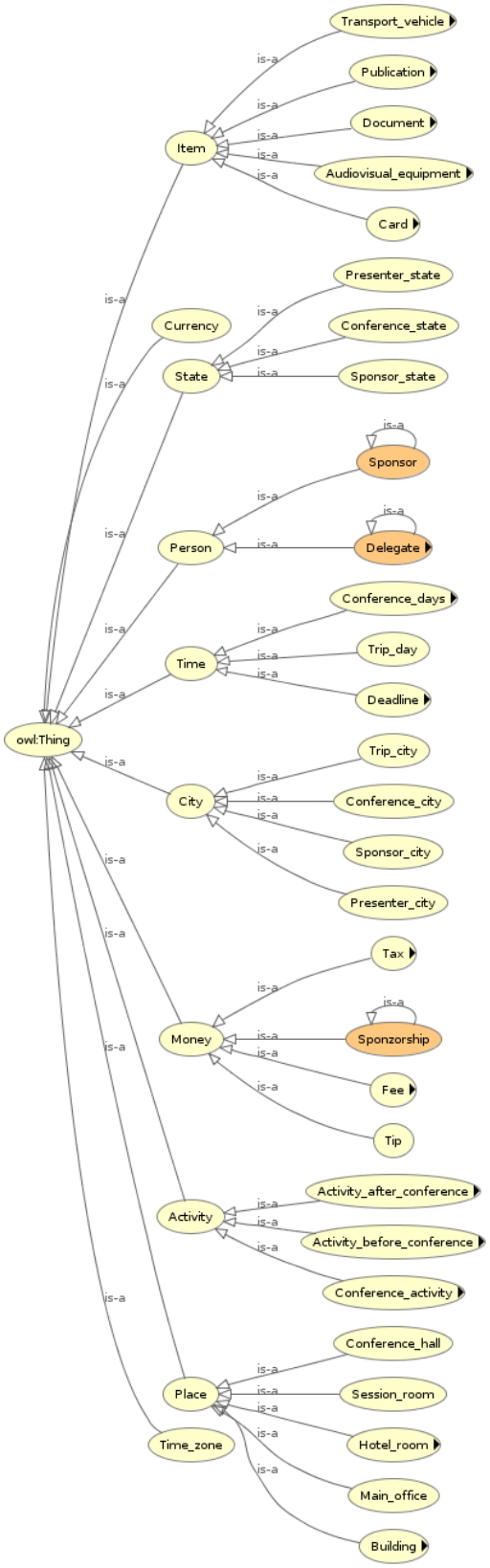

Figure A.3: The highest 3 levels of the class hierarchy of the iasted ontology using OWLViz in protegé

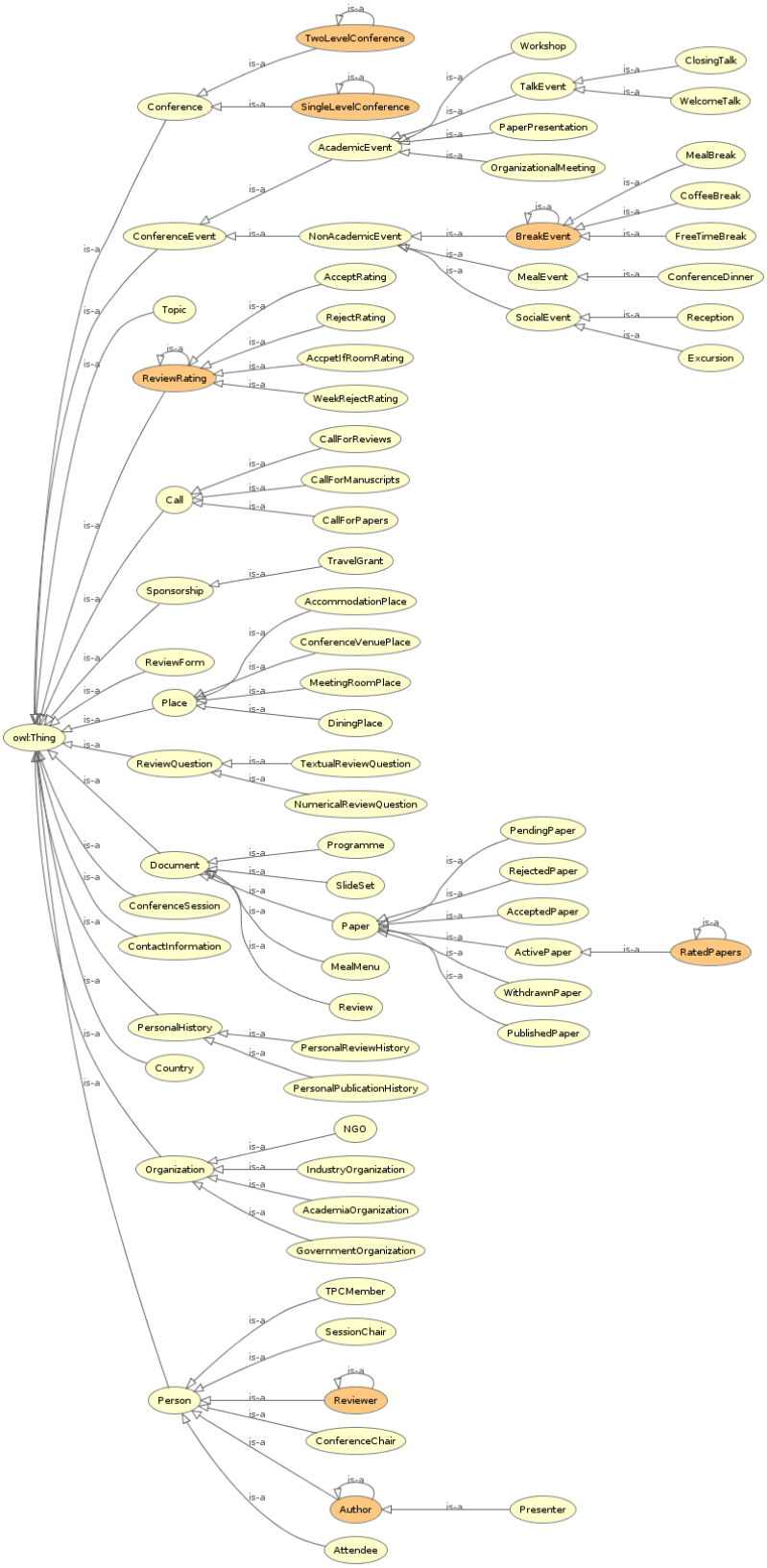

Figure A.4: The class hierarchy of the edas ontology, but without the many topics, using OWLViz in protegé

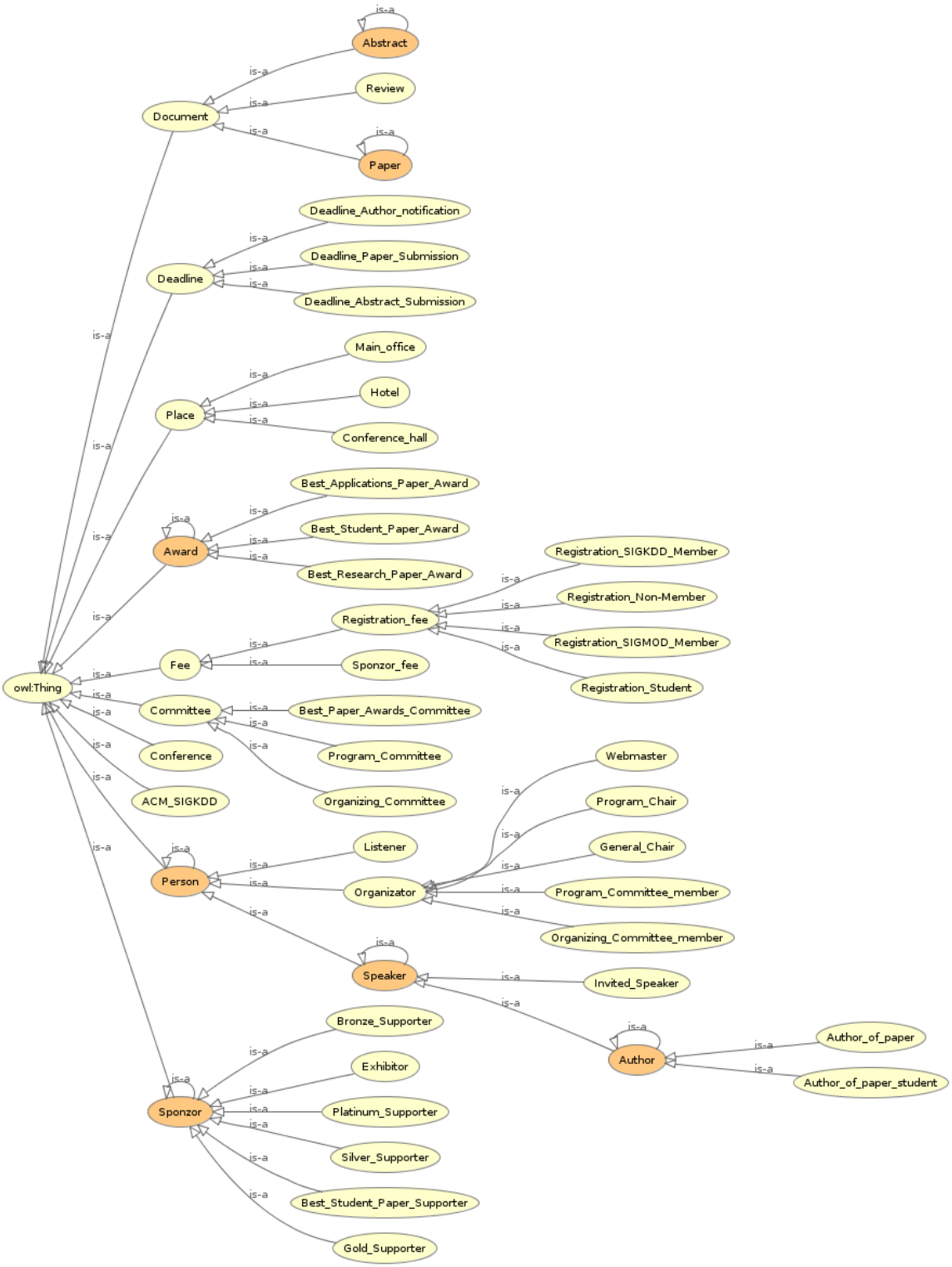

Figure A.5: The class hierarchy of the sigkdd ontology using OWLViz in protegé

Appendix B

Clustering experiments

B.1 Visual assesment

To visually asses the embeddings, we use the python sklearn librarys implementation of PCA. Using PCA we reduce dimensionality to 2 and can plot this in two dimensional space. Only a few plots within each group of related concepts have been included. This way it is easier to se how the vectors relate to each other. The settings for the word2vec trainer are equal for all the plots, with Skip-gram and 200-dimensional embeddings. However, for the walks that go in both direction, we need to reduce the number of epochs because these walks allow for more training data. All embeddings also include other types of edges, but with a lower weight. The **RDF2Vec** walks were created with a depth of 5. If we use deeper walks, it uses a lot of time to generate them. Creating walks with depth 5, however, goes really fast. We increase the epoch for the training to compensate for the small documents. The Onto2Vec-system also create very small documents so we increase the epoch count for this as well.

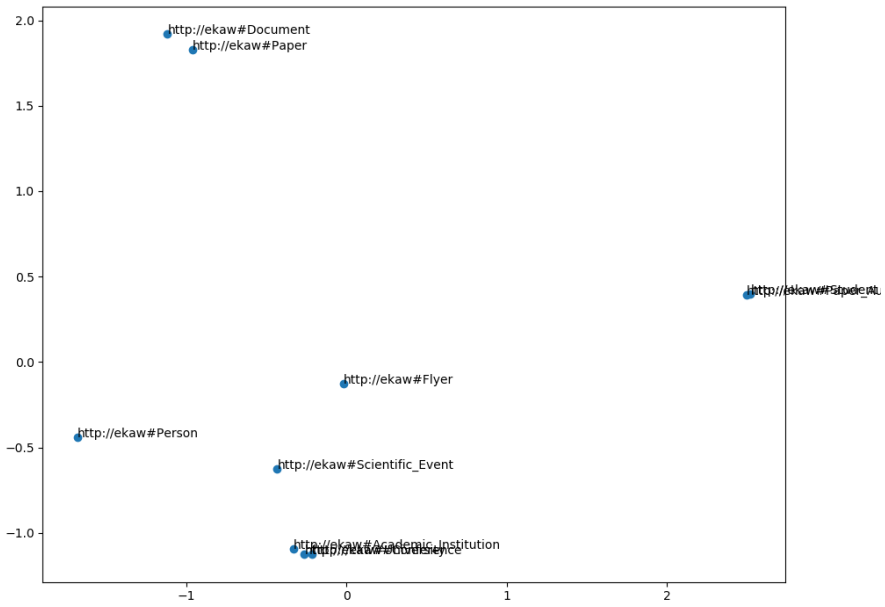

Figure B.1: Plotting some of the points in the ekaw using the **OWL2Vec sub-Class**-system

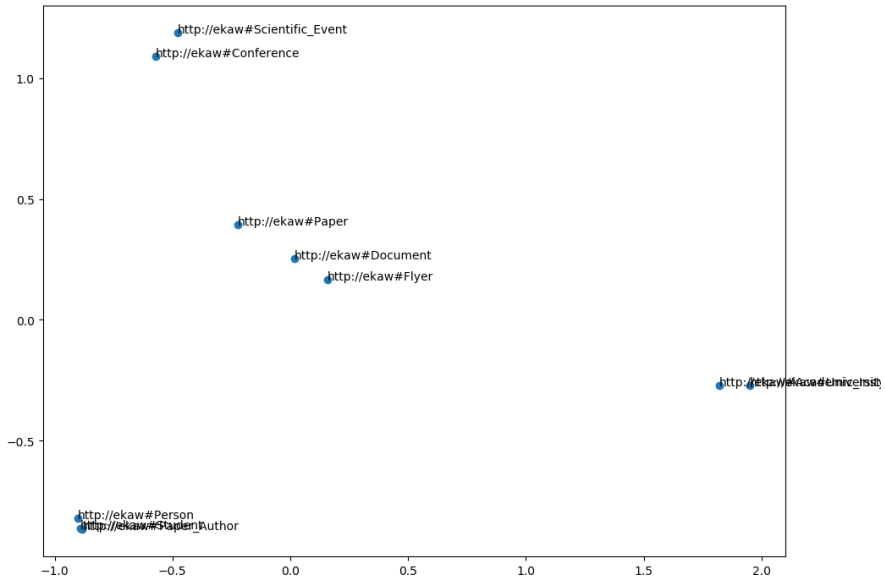

Figure B.2: Plotting some of the points in the ekaw, using the **OWL2Vec+-** system

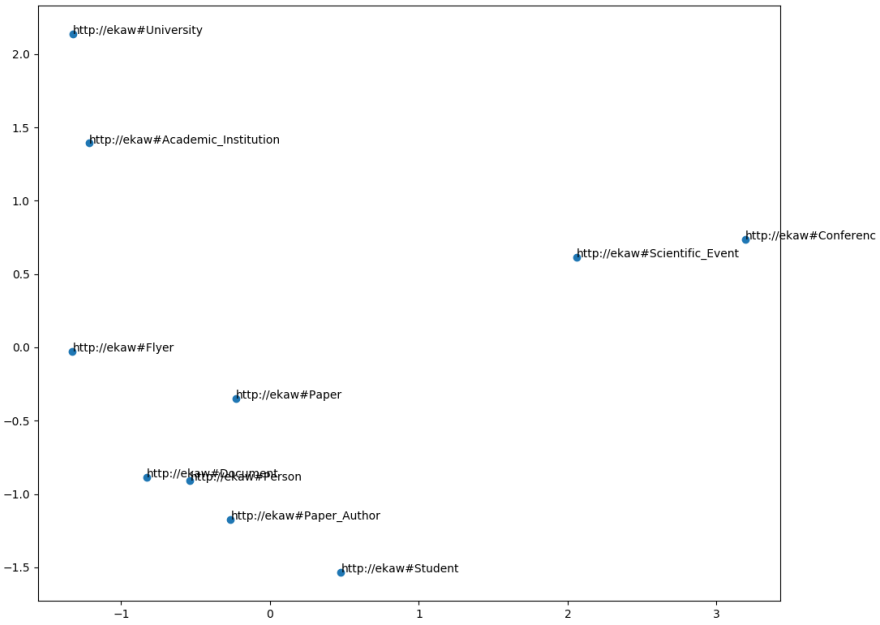

Figure B.3: Plotting the same points using the **RDF2Vec** framework on the projection

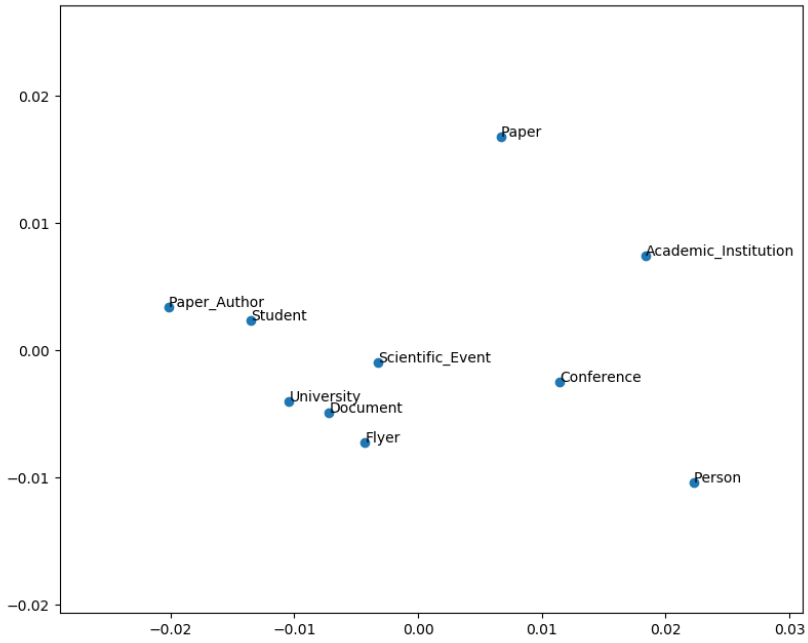

Figure B.4: Plotting the same points using the Onto2Vec framework

B.2 Similarity task

The embeddings for the similarity tasks were trained the same way as for the clustering and visual assesment. The **RDF2Vec** as trained with depth 6 and 100 walks per class. For each concept, we find the most similar concept by cosine distance. It is not easy to quantify if the predictions are good or bad.

Sample	Prediction	cosine
Accepted_Paper	Rejected_Paper	0.922
Workshop_Paper	Poster_Paper	0.968
Regular_Session	Workshop_Session	0.980
Paper	Demo_Paper	0.940
Academic_Institution	Research_Institute	0.977
Conference_Participant	Late-Registered_Participant	0.952
Review	Neutral_Review	0.961
Workshop	Track	0.981
Individual_Presentation	Invited_Talk	0.949
Flyer	Web_Site	0.983
presentationOfPaper	Contributed_Talk	0.595

Table B.1: Some similarity samples using the **OWL2Vec+** system

Sample	Prediction	cosine
Accepted_Paper	Rejected_Paper	0.791
Workshop_Paper	Demo_Paper	0.899
Regular_Session	Demo_Session	0.935
Paper	hasReviewer	0.523
Academic_Institution	scientificallyOrganises	0.541
Conference_Participant	OC_Chair	0.609
Review	reviewOfPaper	0.520
Workshop	Track	0.899
Individual_Presentation	presentationOfPaper	0.548
Flyer	Multi-author_Volume	0.672
presentationOfPaper	reviewWrittenBy	0.672

Table B.2: Similarity samples using **RDF2Vec** with projection

Sample	Prediction	cosine
Accepted_Paper	updatedVersionOf	0.877
Workshop_Paper	Poster_Paper	0.935
Regular_Session	Poster_Session	0.999
Paper	Multi-author_Volume	0.926
Academic_Institution	Proceedings_Publisher	0.804
Conference_Participant	OC_Chair	0.833
Review	Multi-author_Volume	0.952
Workshop	Tutorial	0.984
Individual_Presentation	hasEvent	0.926
Flyer	Multi-author_Volume	0.999
presentationOfPaper	exactly	0.896

Table B.3: Similarity samples using Onto2Vec

Evaluated_Paper	**Invited_Talk**	**Flyer**
DisjointWith	Session	Paper
SubClassOf	Track	Review
Camera_Ready_Paper	Conference	Web_Site
Accepted_Paper	Invited_Talk	Flyer
some	Tutorial	Programme_Brochure
Evaluated_Paper	Workshop	Multi-author_Volume
Submitted_Paper	Social_Event	Industrial_Paper
Assigned_Paper	Individual_Presentation	Poster_Paper
min	Scientific_Event	Demo_Paper
Rejected_Paper	Demo_Session	Regular_Paper
3	Poster_Session	Positive_Reviw
hasReview	Regular_Session	Workshop_Paper
hasReviewer	Workshop_Session	Negative_Reviw
writtenBy	PartOfEvent	Neutral_Review
updatedVersionOf	Conference_Banquet	Conference_Paper
	Conference_Trip	
	Industrial_Session	
	hasEvent	

Student	**(Accepted_Paper**	**Session_Chair**
Person	or	Conference_Participant
Document	Invited_Talk_Abstract	PC_Member
Organisation	(Conference	PC_Chair
Event	Workshop)	Tutorial_Chair
Academic_Institution	(Conference_Paper	Workshop_Chair
Organisation_Agency	(Accepted_Paper	Early-Registered_Participant
Abstract		Late-Registered_Participant
Location		Possible_Reviewer
Student		OC_Member
Proceedings_Publisher		Presenter
Tutorial_Abstract		Session_Chair
Paper_Author		Demo_Chair
Invited_Talks_Abstract		OC_Chair
University		SC_Member
Agency_Staff_Member		Invited_Speaker
Reserach_Institute		
Research_Topic		

1
Contributed_Talk
Proceedings
Proceedings
Thing
volumeContainsPaper
only
Conference_Proceedings
1
presentationOfPaper
exactly

Table B.4: The 7 clusters of ekaw using Onto2Vec

Poster_Paper	**Proceedings_Publisher**	**Demo_Session**
Evaluated_Paper	Proceedings_Publisher	Individual_Presentation
volumeContainsPaper	Academic_Institution	Scientific_Event
Proceedings	Social_Event	Session
Assigned_Paper	PC_Member	Conference
Conference_Paper	scientificallyOrganises	Track
Camera_Ready_Paper	Multi-author_Volume	Workshop
Submitted_Paper	technicallyOrganises	Conference_Session
Industrial_Paper	Flyer	Demo_Session
Workshop_Paper	scientificallyOrganisedBy	Regular_Session
Poster_Paper	Agency_Staff_Member	Poster_Session
Accepted_Paper	Organising_Agency	Workshop_Session
Regular_Paper	Proceedings_Publisher	Industrial_Session
Conference_Proceedings	Student	
Invited_Talk_Abstract	Paper_Author	
Demo_Paper	University	
Rejected_Paper	Research_Institute	
Tutorial_Abstract		

owl#Thing	**PC_Chair**	**Tutorial**
owl#Thing	Presenter	Positive_Review
subClassOf	OC_Member	Contributed_Talk
Event	Workshop_Chair	Neutral_Review
Paper	Late-Registered_Participant	Negative_Review
Person	Tutorial_Chair	Tutorial
Document	PC_Chair	Invited_Talk
organises	Demo_Chair	
hasEvent	Early-Registered_Participant	
presentationOfPaper	Session_Chair	
partOfEvent	Invited_Speaker	
coversTopic	OC_Chair	
organisedBy		
hasReview	**Conference_Banquet**	
topicCoveredBy	Conference_Trip	
paperPresentedAs	Conference_Banquet	
authorOf		
listsEvent		
reviewerOfPaper		
Organisation		
eventOnList		
hasReviewer		
Review		
Possible_Reviewer		
heldIn		
updatedVersionOf		
hasUpdatedVersion		
reviewOfPaper		
locationOf		
publisherOf		
writtenBy		
inverse_of_partOf_7		
Abstract		
Programme_Brochure		
Web_Site		
reviewWrittenBy		
Research_Topic		
Location		

Table B.5: The 7 clusters of ekaw using **RDF2Vec**

Paper	**Document**	**Session**
Paper	Document	Scientific_Event
Camera_Ready_Paper	Review	Session
Regular_Paper	Abstract	Individual_Presentation
Conference_Paper	Positive_Review	Conference
Poster_Paper	Invited_Talk_Abstract	Workshop
Workshop_Paper	Multi-author_Volume	Conference_Session
Demo_Paper	Neutral_Review	Track
Industrial_Paper	Negative_Review	Tutorial
Location	Tutorial_Abstract	Invited_Talk
Research_Topic	Programme_Brochure	Contributed_Talk
	Web_Site	Demo_Session
	Flyer	Poster_Session
	Proceedings	Regular_Session
	Conference_Proceedings	Industrial_Session
Academic_Institution	**Conference_Participant**	**Social_Event**
Organisation	Conference_Participant	Social_Event
Academic_Institution	PC_Member	Event
Proceedings_Publisher	Person	Conference_Trip
Research_Institute	Presenter	Conference_Banquet
Organising_Agency	OC_Member	**Evaluated_Paper**
University	Session_Chair	Evaluated_Paper
	Workshop_Chair	Assigned_Paper
	Tutorial_Chair	Submitted_Paper
	Possible_Reviewer	Accepted_Paper
	PC_Chair	Rejected_Paper
	Invited_Speaker	
	OC_Chair	
	Agency_Staff_Member	
	Early-Registered_Participant	
	Student	
	Paper_Author	
	Late-Registered_Participant	
	Demo_Chair	
	SC_Member	

Table B.6: The 7 clusters of ekaw using **OWL2Vec**

Appendix C

Ontology matching experiments Ekaw-Ekaw

C.1 Semantic embeddings

Anchors	Precision	Recall	F-measure
100	0.87	0.75	0.81
80	0.87	0.65	0.74
60	0.85	0.48	0.61
40	0.86	0.31	0.46
20	0.70	0.12	0.20
0	0.40	0.01	0.01

Table C.1: The **OWL2Vec structural** system using *best candidate* on matching **ekaw-ekaw** conference track using decreasing % anchors and walks

Anchors	Precision	Recall	F-measure
100	0.97	1.00	0.99
80	0.95	0.97	0.96
60	0.85	0.84	0.84
40	0.72	0.62	0.67
20	0.52	0.34	0.41
0	0.71	0.03	0.06

Table C.2: The **OWL2Vec structural** system using *disambiguate* on matching **ekaw-ekaw** conference track using decreasing % anchors and walks

Anchors	Precision	Recall	F-measure
100	0.85	0.81	0.83
80	0.80	0.75	0.77
60	0.83	0.61	0.70
40	0.74	0.42	0.54
20	0.85	0.29	0.43
0	0.86	0.13	0.23

Table C.3: The **OWL2Vec synonyms** system with *best candidate* on matching **ekaw-ekaw** conference track using decreasing % anchors

Anchors	Precision	Recall	F-measure
100	0.96	0.97	0.97
80	0.94	0.91	0.92
60	0.93	0.85	0.89
40	0.89	0.68	0.76
20	0.85	0.62	0.72
0	0.84	0.34	0.48

Table C.4: The **OWL2Vec synonyms** with *disambiguate* on matching **ekaw-ekaw** conference track using decreasing % anchors

Anchors	Precision	Recall	F-measure
100	0.44	0.92	0.59
80	0.45	0.82	0.58
60	0.51	0.72	0.59
40	0.50	0.52	0.50
20	0.56	0.29	0.37
0	0.69	0.15	0.24

Table C.5: The **OWL2Vec synonyms** with *all relations* on matching **ekaw-ekaw** conference track using decreasing % anchors and walks with synonyms

Anchors	Precision	Recall	F-measure
100	1.00	0.90	0.95
80	1.00	0.90	0.95
60	1.00	0.77	0.87
40	1.00	0.49	0.66
20	0.87	0.23	0.37
0	0.04	0.01	0.02

Table C.6: The **OWL2Vec synonyms** with *transformation matrix* on matching **ekaw-ekaw** conference track using decreasing % anchors and walks

Anchors	Precision	Recall	F-measure
100	0.83	0.63	0.72
80	0.81	0.60	0.69
60	0.60	0.44	0.50
40	0.73	0.27	0.39
20	0.66	0.14	0.23
0	0.0	0.0	0.0

Table C.7: The **RDF2Vec** system on *best candidate* on matching **ekaw-ekaw** conference track using decreasing % anchors and walks

Anchors	Precision	Recall	F-measure
100	0.78	0.60	0.68
80	0.80	0.69	0.74
60	0.73	0.68	0.71
40	0.67	0.65	0.66
20	0.47	0.48	0.47
0	0.04	0.04	0.04

Table C.8: The **OWL2Vec subClass** on *best candidate* on matching ekaw-ekaw conference track using decreasing % anchors

Anchors	Precision	Recall	F-measure
100	0.57	0.48	0.52
80	0.56	0.47	0.51
60	0.63	0.52	0.57
40	0.71	0.56	0.63
20	0.75	0.56	0.64
0	0.76	0.29	0.41

Table C.9: The **OWL2Vec 2doc** with *two documents* on matching **ekaw-ekaw** conference track using decreasing % anchors

Anchors	Precision	Recall	F-measure
100	0.91	0.87	0.89
80	0.88	0.79	0.83
60	0.83	0.66	0.74
40	0.71	0.41	0.52
20	0.62	0.21	0.32
0	0.70	0.01	0.03

Table C.10: The **OWL2Vec+** with *best candidate* on matching **ekaw-ekaw** conference track matching using decreasing % anchors

Anchors	Precision	Recall	F-measure
100	0.97	0.97	0.97
80	0.97	0.91	0.94
60	0.94	0.79	0.86
40	0.92	0.57	0.70
20	0.87	0.29	0.44
0	0.40	0.02	0.03

Table C.11: The **OWL2Vec+** with synonyms in the document with *best candidate* on matching **ekaw-ekaw** conference track matching using decreasing % anchors and walks

Anchors	Precision	Recall	F-measure
100	0.97	1.00	0.99
80	0.97	0.99	0.98
60	0.97	0.99	0.98
40	0.95	0.90	0.92
20	0.87	0.59	0.70
0	0.75	0.03	0.05

Table C.12: The **OWL2Vec+** with synonyms in the document and `fastText` using *best candidate* on matching **ekaw-ekaw** conference track matching using decreasing % anchors and walks

Appendix D

Ontology matching experiments Cmt-Ekaw

D.1 Pretrained embeddings

cmt	ekaw	confidence
Author	Paper_Author	0.80
Meta-Review	Review	0.82
Document	Document	1.00
ConferenceChair	Conference	0.87
Review	Review	1.00
ProgramCommitteeMember	Agency_Staff_Member	0.87
Conference	Conference	1.00
Paper	Paper	1.00
Person	Person	1.00

Table D.1: The alignment found between cmt and ekaw using pretrained word embeddings

D.2 Semantic embeddings

cmt	**ekaw**	**confidence**
Meta-Review	Neutral_Review	0.96
User	Student	0.89
ConferenceChair	Late-Registered_Participant	0.78
AuthorNotReviewer	Paper_Author	0.82
PaperFullVersion	Regular_Paper	0.93
ConferenceMember	Conference_Participant	0.78
PaperAbstract	Regular_Paper	0.93
AssociatedChair	Late-Registered_Participant	0.77
Preference	Poster_Paper	0.70

Table D.2: An example of the output of **RDF2Vec** with *best candidate* given 100 % of the anchors

Anchors	Precision	Recall	F-measure
100	0.70	0.60	0.64
80	0.60	0.48	0.53
60	0.64	0.30	0.39
40	0.35	0.23	0.27
20	0.57	0.10	0.17
0	0.00	0.00	0.00

Table D.3: The **OWL2Vec structural** with *best candidate* on matching **ekaw-cmt** conference track using decreasing % anchors

Anchors	Precision	Recall	F-measure
100	0.26	0.85	0.39
80	0.21	0.70	0.32
60	0.16	0.53	0.24
40	0.14	0.43	0.21
20	0.12	0.33	0.18
0	0.11	0.08	0.09

Table D.4: The **OWL2Vec structural** with *disambiguate* on matching **ekaw-cmt** conference track using decreasing % anchors

Anchors	Precision	Recall	F-measure
100	0.74	0.45	0.56
80	0.55	0.33	0.40
60	0.73	0.20	0.31
40	0.83	0.23	0.34
20	0.30	0.05	0.08
0	0.30	0.05	0.08

Table D.5: The **OWL2Vec synonyms** system with *best candidate* on matching **ekaw-cmt** conference track using decreasing % anchors

Anchors	Precision	Recall	F-measure
100	0.38	0.68	0.48
80	0.40	0.65	0.49
60	0.36	0.45	0.40
40	0.32	0.23	0.26
20	0.26	0.15	0.19
0	0.30	0.05	0.08

Table D.6: The **OWL2Vec synonyms** system with *disambiguate* on matching **ekaw-cmt** conference track using decreasing % anchors

Anchors	Precision	Recall	F-measure
100	0.63	0.40	0.48
80	0.44	0.30	0.35
60	0.77	0.33	0.45
40	0.49	0.18	0.23
20	0.67	0.18	0.26
0	0.20	0.03	0.04

Table D.7: The **OWL2Vec synonyms** with *all relations* on matching **ekaw-cmt** conference track using decreasing % anchors

Anchors	Precision	Recall	F-measure
100	0.53	1.00	0.69
80	0.37	0.75	0.49
60	0.30	0.63	0.41
40	0.15	0.38	0.21
20	0.14	0.38	0.20
0	0.08	0.13	0.10

Table D.8: The **OWL2Vec synonyms** system with *transformation matrix* on matching **ekaw-cmt** conference track using decreasing % anchors

Anchors	Precision	Recall	F-measure
100	0.17	0.13	0.14
80	0.0	0.0	0.0
60	0.0	0.0	0.0
40	0.0	0.0	0.0
20	0.0	0.0	0.0
0	0.0	0.0	0.0

Table D.9: The **RDF2Vec** with *best candidate* on matching **ekaw-cmt** conference track using decreasing % anchors

Anchors	Precision	Recall	F-measure
100	0.60	0.95	0.74
80	0.44	0.80	0.57
60	0.33	0.75	0.45
40	0.19	0.53	0.28
20	0.14	0.40	0.21
0	0.07	0.23	0.10

Table D.10: The **OWL2Vec subClass** with *best candidate* on matching **ekaw-cmt** conference track using decreasing % anchors

Anchors	Precision	Recall	F-measure
100	0.61	0.75	0.67
80	0.48	0.58	0.52
60	0.60	0.70	0.64
40	0.62	0.70	0.66
20	0.60	0.63	0.61
0	0.37	0.20	0.26

Table D.11: The **OWL2Vec 2doc** with *two documents* on matching **ekaw-cmt** conference track using decreasing % anchors and walks

Anchors	Precision	Recall	F-measure
100	0.69	0.98	0.81
80	0.68	0.78	0.72
60	0.66	0.55	0.59
40	0.55	0.50	0.52
20	0.55	0.23	0.31
0	0.20	0.03	0.04

Table D.12: The **OWL2Vec+** with *best candidate* on matching ekaw-cmt conference track matching using decreasing % anchors

Anchors	Precision	Recall	F-measure
100	0.97	0.78	0.86
80	0.93	0.68	0.78
60	0.93	0.50	0.63
40	1.00	0.43	0.59
20	1.00	0.25	0.39
0	0.60	0.08	0.13

Table D.13: The **OWL2Vec+** with synonyms in the document and *best candidate* on matching ekaw-cmt conference track matching using decreasing % anchors

Anchors	Precision	Recall	F-measure
100	0.60	0.95	0.73
80	0.62	0.83	0.69
60	0.61	0.63	0.62
40	0.65	0.40	0.49
20	0.61	0.23	0.31
0	0.47	0.15	0.23

Table D.14: The **OWL2Vec+** with synonyms in the same document and using `fastText` and *best candidate* on matching ekaw-cmt conference track matching using decreasing % anchors and walks

Anchors	Precision	Recall	F-measure
100	0.72	0.88	0.79
80	0.70	0.75	0.72
60	0.86	0.60	0.71
40	0.85	0.38	0.52
20	0.90	0.38	0.53
0	1.00	0.13	0.22

Table D.15: The **OWL2Vec+** with two documents and *two documents* alignment system on matching ekaw-cmt conference track matching using decreasing % anchors