POZNAN UNIVERSITY OF TECHNOLOGY

FACULTY OF COMPUTING AND TELECOMMUNICATION
Institute of Computing Science

Master's thesis

LEARNING CLASS EXPRESSION EMBEDDINGS WITH A TRANSFERABLE DEEP NEURAL REASONER

Dariusz Max Adamski, nr indeksu 136674

Supervisor
dr inż. Jędrzej Potoniec

POZNAŃ 2022

Abstract

In our work we design, implement and evaluate a novel method of learning concept embeddings in knowledge bases for the ALC description logic, using a transferable deep neural reasoner. Our method of learning embeddings ensures that the resulting embeddings are entirely data-driven, which unlike manually-designed concept vectorization schemes, captures as many useful properties of the given training data as possible. The deep neural reasoner consists of two modules - a reasoner head, that is a deep neural network classifier, trained to classify whether subsumption axioms hold for a given knowledge base, and an embedding layer that can construct embedding vectors for arbitrarily complex ALC concepts. The reasoner head is transferable, because the embedding layers learn how to embed concepts in a space that minimizes the classifier loss, and is shared between all knowledge bases. In our work we hypothesize, and experimentally show support for the idea that concepts from different knowledge bases can be represented in a shared embedding space, with a topology that lends itself for approximate reasoning by entailment classifiers based on deep neural networks.

Karta
pracy dyplomowej magisterskiej

Uczelnia:	Politechnika Poznańska	Profil studiów:	Ogólnoakademicki
Kierunek:	Informatyka	Forma studiów:	Stacjonarne
Studia w zakresie:	Sztuczna Inteligencja	Poziom studiów:	II stopnia

Zobowiązuję/zobowiązujemy się samodzielnie wykonać pracę w zakresie wyspecyfikowanym niżej. Wszystkie elementy (m.in. rysunki, tabele, cytaty, programy komputerowe, urządzenia itp.), które zostaną wykorzystane w pracy, a nie będą mojego/naszego autorstwa będą w odpowiedni sposób zaznaczone i będzie podane źródło ich pochodzenia.

Jeżeli w wyniku realizacji pracy zostanie dokonany wynalazek, wzór użytkowy, wzór przemysłowy, znak towarowy, prawa do rozwiązań przysługiwać będą Politechnice Poznańskiej. Prawo to zostanie uregulowane odrębną umową.
Oświadczam, iż o wyniku prac wskazanych powyżej, a także o innych, w tym tych, które mogą być przedmiotem tajemnicy Politechniki Poznańskiej, niezwłocznie powiadomię promotora pracy.
Zobowiązuję się ponadto do zachowania w tajemnicy wszystkich informacji technicznych, technologicznych, organizacyjnych, uzyskanych w Politechnice Poznańskiej w okresie od daty rozpoczęcia realizacji prac do 5 lat od daty zakończenia wykonania prac.

	Imię i nazwisko	Nr albumu	Data i podpis
Student:	Dariusz Max Adamski	136674	

Tytuł pracy:	Uczenie się zanurzeń wyrażeń klasowych z wykorzystaniem transferowalnego mechanizmu wnioskowania opartego na głębokich sieciach neuronowych
Wersja angielska tytułu:	*Learning Class Expression Embeddings with a Transferable Deep Neural Reasoner*
Dane wejściowe:	Wstępne wyniki badawcze promotora. Dokumentacja wykorzystywanych bibliotek programistycznych i narzędzi. Literatura naukowa poświęcona zagadnieniom integracji neurosymbolicznej.
Zakres pracy:	1. Utworzenie generatora zbioru danych i analiza powstałego zbioru 2. Opracowanie i implementacja architektury neuronowego mechanizmu wnioskowania 3. Uczenie mechanizmu wnioskowania i ewaluacja na zbiorze testowym, przy użyciu wybranych miar 4. Analiza powstałych zanurzeń dla wybranych ontologii
Termin oddania pracy:	30.06.2022
Promotor:	dr inż. Jędrzej Potoniec
Jednostka organizacyjna promotora:	Instytut Informatyki

podpis dyrektora/kierownika jednostki organizacyjnej promotora

data i podpis Dziekana

Contents

Chapter 1

Introduction

In our work we design, implement and evaluate a novel method of learning concept embeddings in knowledge bases for the $\mathcal{ALC}$ description logic, using a transferable deep neural reasoner. Our method of learning embeddings ensures that the resulting embeddings are entirely data-driven, which unlike manually-designed concept vectorization schemes, captures as many useful properties of the given training data as possible. The deep neural reasoner consists of two modules – a reasoner head, that is a deep neural network classifier, trained to classify whether subsumption axioms hold for a given knowledge base, and an embedding layer that can construct embedding vectors for arbitrarily complex $\mathcal{ALC}$ concepts. The reasoner head is transferable, because the embedding layers learn how to embed concepts in a space that minimizes the classifier loss, and is shared between all knowledge bases. In our work we hypothesize, and experimentally show support for the idea that concepts from different knowledge bases can be represented in a shared embedding space, with a topology that lends itself for approximate reasoning by entailment classifiers based on deep neural networks.

Our work is structured as follows. In chapter 2, we introduce description logics, by first providing an intuitive explanation, and then introducing them more formally, along with the related notation that we use throughout this text. We also introduce the field of neuro-symbolic artificial intelligence, and describe its goals, and some approaches for integrating deep neural networks with description logics. Chapter 3 describes the design of our deep neural reasoner in detail. We then describe the results of an experiment that we conducted, to check how good our reasoner is at classifying whether subsumption axioms hold for a given knowledge base, and whether the reasoner head is transferable as expected. Since we used a synthetic data set for the experiment in chapter 3, we could not directly analyze or interpret the learned embeddings. Thus, in chapter 4 we use our method to learn concept embeddings in a real-world knowledge base, and visualize them to subjectively assess their quality in addition to metrics introduced in chapter 3. We also conduct an experiment to see how good the reasoner is at classifying entailment in real-world knowledge bases, and check if reasoner heads that were pre-trained on the synthetic data set can be used to effectively learn embedding layers for real-world knowledge bases. Lastly, in chapter 5 we summarize the results of our work and point our potential improvements, extensions, and applications of our deep neural reasoner and concept embeddings.

Chapter 2

Background

At the time of writing, the field of artificial intelligence (AI) is mostly focused on deep learning (machine learning using deep neural networks), so we assume that the reader is familiar with the process of supervised training of deep neural networks. However, since description logics are relatively little known, when compared to the ubiquitous deep learning, we provide a short introduction to description logics. We also describe the emerging field of neuro-symbolic AI, that attempts to bridge the gap between the neural and symbolic AI paradigms, and name the role that deep neural reasoners play in this field.

2.1 Description logics

Description logics (DLs) are a family of logics, that are widely used as a formal way to represent knowledge in the form of knowledge bases (KBs). An advantageous property of DLs is that reasoning in most of them is decidable, since they are fragments of first-order predicate logic [1]. The formalism of DLs has been used as a basis for the Web Ontology Language (OWL) standard, which as part of the Semantic Web is responsible for describing the semantics, or meaning, of data on the Internet [2]. DLs and the foundational technologies of the Semantic Web are considered mature, and are used both in research and the industry [3]. An illustrative example of the potential applications of DLs is SNOMED CT – a knowledge base that is considered to be the most comprehensive multilingual clinical healthcare terminology in the world [4].

As mentioned, DLs are a family of logics, which means that there are many DLs of varying expressiveness. Since in our work we focus on the $\mathcal{ALC}$ DL (abbreviated from attributive language with complements), we also introduce DLs from the perspective of $\mathcal{ALC}$. There exist DLs that are simpler than $\mathcal{ALC}$, for example $\mathcal{EL}$, and there are also vastly more expressive DLs like $\mathcal{SHOIN}^{(\mathcal{D})}$ and $\mathcal{SROIQ}^{(\mathcal{D})}$, that are the basis of OWL DL and OWL 2 DL respectively. We do not deal with $\mathcal{EL}$ or $\mathcal{SROIQ}^{(\mathcal{D})}$, but we do discuss a procedure for transforming $\mathcal{SHOIN}^{(\mathcal{D})}$ knowledge bases to $\mathcal{ALC}$ knowledge bases in chapter 4, by removing parts that $\mathcal{ALC}$ does not support.

For readers that are unfamiliar with DLs we begin with an intuitive overview. In later sections we describe the notation we use, and define the syntax and semantics of the $\mathcal{ALC}$ DL. The introduction to DLs is based on [5].

2.1.1 Overview

At a high level, a KB divides knowledge into *terminology* (also called TBox) and *assertions* (also called ABox). One may think that the terminology describes general knowledge (e.g. "Dogs are mammals"), and assertions state facts (e.g. "Fido is a dog" or "Fido likes Alice"). Those elements

of KB terminology and assertions are called *axioms*. Axioms are formulated using a *vocabulary*, which consists of *individuals* (e.g. "Fido" or "Alice"), *concept names* (e.g. "dog" or "mammal"), and *role names* (e.g. "likes"). Besides concept names in the vocabulary, complex concepts can be constructed from other concepts using *concept constructors*. Available constructors are different depending on the chosen DL. For example in the $\mathcal{ALC}$ DL there are concept complement, intersection, union, universal restriction, and existential restriction constructors. Compare that to the simpler $\mathcal{EL}$ DL, which only supports concept intersection and existential restriction constructors.

The TBox contains *subsumption axioms*, that allow one to state that one concept is a subset of another concept. There are two kinds of assertion axioms in ABox. A *concept assertion axiom* allows one to state that an individual is an instance of a concept, and a *role assertion axiom* allows one to relate two individuals by a binary predicate.

Semantics of DLs are defined in a model-theoretic way, by providing an *interpretation* that represents the KB vocabulary in terms of a set called the *domain*. In particular, an interpretation maps individuals to elements of the domain, concepts to sets of individuals, and roles to binary relations between individuals. There are also special concepts, called the *bottom concept* and *top concept*, which correspond to the empty set and the domain, respectively.

The semantics of DLs provide an *entailment* relation, so that given a set of axioms, logical consequences may be inferred. In other words, KB entails a given axiom if that axiom follows from KB's axioms.

2.1.2 Notation

Formally, a KB in the $\mathcal{ALC}$ DL is a pair $\mathcal{K} = (\mathcal{T}, \mathcal{A})$, where $\mathcal{T}$ is the set of terminological axioms and $\mathcal{A}$ is the set of assertions. The vocabulary of a KB is defined as a triple (N_C, N_R, N_I), where N_C is the set of concept names, N_R is the set of role names, and N_I is the set of individual names. Given an arbitrary ordering of the set of concept names N_C and an arbitrary ordering of the set of role names N_R, we define A_i as the i-th concept name, and R_i as the i-th role name, so that $A_i \in N_C$ and $R_i \in N_R$. In text, we write individuals and concept names in `PascalCase`, and role names in `camelCase`.

2.1.3 Semantics

The interpretation is defined as a pair $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$, where $\Delta^{\mathcal{I}}$ is a set called the domain, and $\cdot^{\mathcal{I}}$ is the interpretation function. The interpretation function $\cdot^{\mathcal{I}}$ maps each concept C to a subset of $\Delta^{\mathcal{I}}$, each role R to a subset of $\Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$, and each individual to an element of $\Delta^{\mathcal{I}}$. The syntax and interpretation of concepts in $\mathcal{ALC}$ is summarized in Table 2.1.

The semantics of description logics provide a consequence relation $\models$. Given an axiom α, $\mathcal{K} \models \alpha$ means that $\mathcal{K}$ entails α, or in other words that α is a logical consequence of $\mathcal{K}$. The entailment $\mathcal{K} \models \alpha$ holds iff for every interpretation $\mathcal{I}$, where $\mathcal{I} \models \mathcal{K}$, the entailment $\mathcal{I} \models \alpha$ also holds. In turn, $\mathcal{I} \models \mathcal{K}$ holds iff $\mathcal{I} \models \mathcal{T}$ (that is iff $\mathcal{I} \models \alpha$ holds for every axiom $\alpha \in \mathcal{T}$) and $\mathcal{I} \models \mathcal{A}$ (that is iff $\mathcal{I} \models \alpha$ holds for every axiom $\alpha \in \mathcal{A}$).

The set of terminological axioms $\mathcal{T}$ contains subsumption axioms $C \sqsubseteq D$, that are read C is subsumed by D, where C and D are concepts. The entailment $\mathcal{I} \models C \sqsubseteq D$ holds iff the interpretation of C is a subset of the interpretation of D, written $C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$. In other words, if $C \sqsubseteq D$, then every element of C is also an element of D. Sometimes $\mathcal{T}$ also contains *equivalence axioms* $C \equiv D$, although such axioms can be expressed in terms of two subsumption axioms $C \sqsubseteq D$ and $D \sqsubseteq C$.

TABLE 2.1: Syntax and semantics of $\mathcal{ALC}$

Description	Syntax	Semantics
top	$\top$	$\Delta^{\mathcal{I}}$
bottom	$\bot$	$\emptyset$
intersection	$C \sqcap D$	$C^{\mathcal{I}} \cap D^{\mathcal{I}}$
union	$C \sqcup D$	$C^{\mathcal{I}} \cup D^{\mathcal{I}}$
complement	$\neg C$	$\Delta^{\mathcal{I}} \setminus C^{\mathcal{I}}$
existential restriction	$\exists R.C$	$\{x \in \Delta^{\mathcal{I}} \mid \exists_{y \in \Delta^{\mathcal{I}}}((x,y) \in R^{\mathcal{I}} \wedge y \in C^{\mathcal{I}})\}$
universal restriction	$\forall R.C$	$\{x \in \Delta^{\mathcal{I}} \mid \forall_{y \in \Delta^{\mathcal{I}}}((x,y) \in R^{\mathcal{I}} \rightarrow y \in C^{\mathcal{I}})\}$

The set of assertions $\mathcal{A}$ contains concept assertion axioms $C(a)$, and role assertion axioms $R(a,b)$, where C is a concept, R is a role and a and b are individuals. The entailment $\mathcal{I} \models C(a)$ holds iff the interpretation of a is an element of the interpretation of C, written $a^{\mathcal{I}} \subseteq C^{\mathcal{I}}$. The entailment $\mathcal{I} \models R(a,b)$ holds iff the ordered pair of interpretations of a and b is contained in the interpretation of R, written $(a^{\mathcal{I}}, b^{\mathcal{I}}) \in R^{\mathcal{I}}$.

In the context of description logics, *classification* refers to checking entailment $\mathcal{K} \models C \sqsubseteq D$ for all pairs of concept names $C, D \in N_C$. To avoid confusion with classification in the context of machine learning, we will only refer to classification in the context of description logics as *knowledge base classification*, or *ontology classification* when dealing with an OWL ontology.

2.1.4 OWL ontologies

The Web Ontology Language (OWL) is based on description logics, but uses a different nomenclature. In particular, knowledge bases are called *ontologies* in OWL. We also use the term ontology, but only when referring to a KB described in OWL. Concept names and role names are called class names and object property names, respectively. In general, concepts are referred to as classes or class expressions.

Terminological axioms are called class axioms, and a subsumption $C \sqsubseteq D$ is read "C is a subclass of D". While we did not discuss role axioms as $\mathcal{ALC}$ does not support them, OWL ontologies do support object property axioms, which are the same thing. In addition to class, object property, and individual axioms, OWL ontologies can also contain annotation properties and annotation axioms, that allow one to provide additional information about classes. For example, one very common annotation property is `rdfs:label`, which is used to provide human-readable labels.

2.1.5 Semantic reasoners

Many *semantic reasoners* for DLs are available. The role of a semantic reasoner is to provide reasoning services for knowledge bases. To us, the ability to efficiently perform entailment checking was of the greatest interest. Interestingly, any semantic reasoner that can perform satisfiability, equivalence or subsumption checking can also check for the other two [6].

Another important property of semantic reasoners is their support for different description logics. For example, in our work we consider the $\mathcal{ALC}$ DL, but we could also use a semantic reasoner for a more expressive DL. Indeed, because of their availability and ease-of-use, we considered three semantic reasoners: HermiT [7], Pellet [8], and FaCT++ [9]. HermiT and Pellet fully support reasoning in the OWL 2 ontology language, which corresponds to the $\mathcal{SROIQ}^{(\mathcal{D})}$ DL, and FaCT++ fully supports the OWL DL, which corresponds to the less expressive $\mathcal{SHOIN}^{(\mathcal{D})}$ DL. A reasoner

that supports reasoning in $\mathcal{SHOIN}^{(\mathcal{D})}$ or $\mathcal{SROIQ}^{(\mathcal{D})}$ can also perform reasoning in $\mathcal{ALC}$, so the mentioned semantic reasoners were appropriate for our use-case.

2.2 Neuro-symbolic integration

The neural and symbolic paradigms of artificial intelligence are vastly different. On one hand, the currently dominant neural paradigm shows how well neural networks perform on large-scale data sets, ranging from simple classification and regression, to language models so powerful, that their output is almost indistinguishable from human-written text [10], and generative image models that can synthesize photorealistic images from text prompts [11]. However, neural models often produce nonsensical results, for example paragraphs of text that contradict themselves, and because neural models are not easily interpretable, it is difficult to find the cause of such problems. On the other hand, the symbolic paradigm offers methods for explicitly describing knowledge, and reliably performing reasoning over that knowledge. Symbolic methods can provide step-by-step explanations of their inferences, because they use deductive reasoning. Unfortunately, symbolic methods are not well suited for learning from data, as real-life knowledge is often seemingly contradictory. Symbolic methods also have trouble with processing large-scale data sets, because of high time complexities of used algorithms. The research field of neuro-symbolic integration aims to combine the large-scale learning ability of neural models, and the ability of symbolic methods to express knowledge and perform reasoning, all while keeping interpretability, thus combining the benefits and avoiding the pitfalls of both paradigms [12].

2.2.1 Ontology embedding methods

One way of bridging the neural and symbolic paradigms is representing symbolic knowledge in terms of vectors in a high-dimensional real vector space. Such vectors can then be used as additional inputs to machine learning models based on neural networks, to improve their performance by allowing them to use an approximation of expert knowledge [13]. A good method of learning embeddings from symbolic knowledge should ideally leverage the structural information present in relations between abstract concepts, and should not try to learn embeddings for abstract concepts with the aid of word embeddings, due to the ambiguity of language, its limited abstraction, and other problems [14].

TransE is one of many methods of learning concept embeddings from structural information in knowledge bases [15, 16]. However, TransE and similar methods rely on assumptions, such as that relations between concepts are modeled as translations in the embedding space. These assumptions make the resulting embeddings less data-driven, which resulted in a large number of similar embedding methods that make slightly different assumptions. TransE and related methods do not use the semantics of description logics to their advantage, which limits the usefulness of the learned embeddings.

2.2.2 Deep deductive reasoners

Extensive work has been done on integrating the neural and symbolic paradigms further than by just learning embeddings. There exist deep deductive reasoners, that aim to actually perform reasoning using deep neural networks [12].

Many deep deductive reasoners have been proposed, including ones using long short-term memory networks (LSTMs) for reasoning in the $\mathcal{EL}+$ description logic [17], ones for reasoning in the

first-order logic (FOL) with logic tensor networks (LTNs) [18, 19], or even ones created specifically for solving the SAT problem [20]. Deep neural reasoners are often entailment classifiers, but they can also classify satisfiability, because as we mentioned before, checking one can be reduced to checking the other. The mentioned deep neural reasoners are careful to ensure that they are transferable, as training a reasoner from scratch for each new knowledge base would be infeasible.

In our work we introduce a novel deductive reasoner that classifies entailment axioms for the $\mathcal{ALC}$ description logic, with primary focus on learning concept embeddings. We chose the $\mathcal{ALC}$ description logic, because we felt, that it was under-represented in the space of deep deductive reasoners. Additionally, a deep deductive reasoner for $\mathcal{ALC}$ can be used for reasoning in OWL ontologies, if care is taken to transform the OWL ontologies to $\mathcal{ALC}$ knowledge bases.

Chapter 3

Deep neural reasoner

After introducing description logics, we are ready to describe the architecture of our deep neural reasoner, and describe the process of using it to learn concept embeddings for knowledge bases described using the $\mathcal{ALC}$ description logic. In this chapter we also focus on an experiment, where we evaluated how accurate is the deep neural reasoner, how good are the learned embeddings, and if the reasoner is actually transferable.

However, before introducing our method, we briefly list the technologies we used in our work, and describe non-trivial technical aspects of our work.

3.1 Technologies

All of our code is written in Python 3.9.7 and Cython (a superset of Python that compiles to C or C++). The deep neural reasoner is implemented in PyTorch 1.10.1, and the results of experiments are saved in Jupyter Notebooks. Other used libraries include NumPy, Pandas, Scikit-learn, Matplotlib, and Seaborn.

To run our experiments we used a computer with an Intel Core i5-4670K CPU (4 cores, 4 threads, 4.4GHz frequency), 32 GB of DDR3 RAM (1600MHz, dual-channel), and no dedicated GPU. To accelerate deep learning, PyTorch used the AVX2 instructions supported by our CPU. We tested our code on the Void Linux distribution (kernel version 5.15).

We use pseudo-randomly generated numbers to create our data sets and in training. For data generation we use the NumPy implementation of the PCG pseudo-random number generator, which efficiently generates high-quality pseudo-random numbers [21]. The internal PyTorch pseudo-random number generator is used during training. To ensure reproducibility of our experiments, we set the initial states of the pseudo-random number generators to a known initial value.

During training, our reasoner uses inferences made by a semantic reasoner as the expected class in classification. Initially we used HermiT to perform inferences in KBs, because it implements a particularly efficient reasoning algorithm. Unfortunately, technical issues prevented us from using either HermiT or Pellet. Both semantic reasoners run on the Java Virtual Machine (JVM), which is not a problem when using them from Java code. However, we used Python and PyTorch to implement our deep neural reasoner, and interfacing with the JVM from Python is impractical – the only option we had was using the semantic reasoner from the command-line, which required IO operations and starting the JVM every time we needed to make a single inference. Given that we needed to perform up to tens of thousands of inferences per second, the overhead was unacceptable.

Fortunately, the FaCT++ semantic reasoner was written in C++, which means that it is easy

and efficient to interface with it from Python via a Cython extension. The fact that FaCT++ only fully supports OWL DL was not a problem, because we did not need support for any features of OWL 2.

The only Python library for interfacing with FaCT++ was not as fast as we wanted, and lacked access to important functions of the semantic reasoner, so we modified it to suit our needs[1]. The modifications that we introduced addressed performance issues (we removed unnecessary memory allocations), and added support for missing concept constructors (concept complement, existential and universal restrictions) and inference functions (we added fast functions for checking entailment of subsumption, equivalence and disjointness axioms).

Since we wrote a custom interface to FaCT++ we were able to choose a KB representation that was both easy-to-use for us and efficient for FaCT++ and our reasoner. In our code, a KB is represented by the set of terminological axioms $\mathcal{T}$, an array of $|N_C|$ strings, where the i-th element corresponds to concept name A_i, and an array of $|N_R|$ strings, where the i-th element corresponds to role name R_i. Axioms and concept expressions are represented with plain Python tuples, where the first element is a globally unique integer representing one of the following symbols: $\sqsubseteq$, $\equiv$, $\neg$, $\sqcap$, $\sqcup$, $\forall$, or $\exists$, and the next elements are the operands. Similarly, the top and bottom concepts are represented by globally unique integers. In a concept expression, both the i-th concept name A_i and the i-th role name R_i are simply represented by the integer i, which is more space efficient than representing concept names and role names with strings [22]. Concept names and role names can be distinguished from each other because the latter only occur as the second element of tuples where the first element is the symbol $\forall$ or $\exists$. The string value of a concept or role name can be obtained by indexing the N_C or N_R array, respectively. As a consequence, concepts and roles can be renamed in constant time, without modifying any axioms or concept expressions. It is easy to see that our representation almost directly mirrors the syntax of the $\mathcal{ALC}$ DL.

In chapter 4 we work with an OWL ontology. Initially we used Owlready2 [23] to parse OWL ontologies in the RDF/XML format, but after we found that it does not parse some OWL ontologies correctly, we wrote our own parser for the OWL functional-style syntax [24], which closely corresponds to the syntax of $\mathcal{ALC}$, and by extension to our in-memory representation of KBs. Our parser greatly simplified the ontology loading code. We also improved the performance of loading ontologies by writing our parser in Cython and by directly storing ontologies in our efficient KB representation (we increased ontology loading speeds by at least 400%, without even counting the time spent on converting from the Owlready2 ontology representation to our KB representation). To convert ontologies from other OWL formats to the OWL functional-style syntax, we use the ROBOT command-line tool [25].

3.2 Architecture

Our deep neural reasoner (hereafter, *reasoner*) is a classifier, which given a subsumption axiom, outputs the probability that the axiom is entailed by a given knowledge base. The reasoner consists of the generic *reasoner head*, and an interchangeable *embedding layer* specific to a given given knowledge base. The generic reasoner head can classify entailment, and construct embeddings for concept complements and intersection of concepts. While an embedding layer contains embeddings for a given knowledge base – that is, it stores embedding vectors for concept names and can construct embeddings for existential restrictions for a given role name. A diagram of the architecture of our deep neural reasoner is shown in Figure 3.1.

[1]The original source code of the Python library for FaCT++ is available at `https://bitbucket.org/wrobell/factplusplus/src/factpp/factpp/`

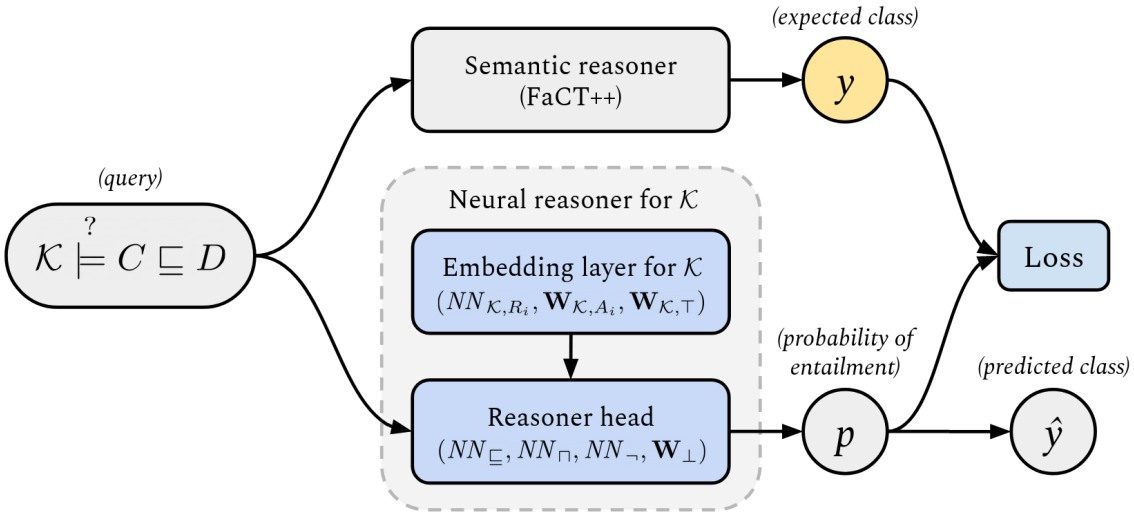

FIGURE 3.1: A high-level overview of the neural reasoner architecture. An embedding layer for knowledge base $\mathcal{K}$ and a reasoner head are combined to create a neural reasoner for $\mathcal{K}$. The reasoner predicts whether $\mathcal{K}$ entails subsumption axioms, written $\mathcal{K} \models C \sqsubseteq D$, where C and D are $\mathcal{ALC}$ concepts. We train the reasoner by contrasting the predicted probability of entailment with the answer provided by a semantic reasoner. The reasoner head generalizes across all $\mathcal{ALC}$ knowledge bases, but for each $\mathcal{K}$ we have to train a separate embedding layer. The reasoner head may be frozen while training an embedding layer for a new knowledge base $\mathcal{K}$, to make learning faster.

The main purpose of the classifier is to facilitate the computation of the gradient of the loss function with respect to the weights of the embedding layer. In other words, when the reasoner is given an axiom, it builds its representation bottom-up from the KB-specific embeddings in the embedding layer. That representation is then used by a neural network in the reasoner head to classify whether the axiom is entailed by that KB. The classification output is contrasted with the target output computed by a semantic reasoner, and the value of the loss function is used to adjust weights of the embedding layer with backpropagation through structure [26].

Remember that even though we try to train the best classifier possible, the goal of the entailment classifier is learning good concept embeddings, and the resulting neural reasoner is a useful byproduct.

3.2.1 Reasoner head

In our work we chose the reasoner to be an entailment classifier for subsumption axioms in $\mathcal{ALC}$ – that is, given an axiom $C \sqsubseteq D$, the reasoner head outputs the probability that the axiom is entailed by a given knowledge base $\mathcal{K} \models C \sqsubseteq D$. We identified integrating the ABox into reasoning as out-of-scope, so assuming that $\mathcal{A}$ is empty, $\mathcal{K} \models C \sqsubseteq D$ holds iff $\mathcal{T} \models C \sqsubseteq D$ holds.

Assume that $h_{\mathcal{K}}$ is a function mapping $\mathcal{ALC}$ concepts in knowledge base $\mathcal{K}$ to embedding vectors in real vector space $\mathbb{R}^{N_e}$, where N_e is the embedding dimension.

The classification output for an entailment query $P(\mathcal{K} \models C \sqsubseteq D) \in (0, 1)$ (see Equation 3.1) is defined as the output of the feedforward neural network $NN_{\sqsubseteq}$, which accepts an *interaction map* of the embeddings $h_{\mathcal{K}}(C)$ and $h_{\mathcal{K}}(D)$, and has a single output neuron with the sigmoid activation function. In our experiments, $NN_{\sqsubseteq}$ also has one hidden layer with 16 neurons, which is followed by the ELU activation function [27]. Additional hidden layers may be added, or the number of neurons per layer may be increased if one so wishes.

$$P(\mathcal{K} \models C \sqsubseteq D) = \sigma(NN_{\sqsubseteq}(h_{\mathcal{K}}(C), h_{\mathcal{K}}(D))) \tag{3.1}$$

An interaction map of a pair of embedding vectors $IM(e_C, e_D)$, as defined by Equation 3.2, is

a concatenation of the embedding vector e_C, the embedding vector e_D and their outer product $e_C \otimes e_D$. Each element of the outer product of two vectors $(\mathbf{u} \otimes \mathbf{v})_{ij}$ is defined as $\mathbf{u}_i \cdot \mathbf{v}_j$. Before concatenation, the outer product is reshaped to a row vector.

$$IM(e_C, e_D) = [e_C; e_D; \text{flatten}(e_C \otimes e_D)] \tag{3.2}$$

Initially, $NN_{\sqsubseteq}$ simply accepted the concatenation of embeddings $h_{\mathcal{K}}(C)$ and $h_{\mathcal{K}}(D)$. However, we found that a small neural network was unable to learn to classify entailment even for the simplest axioms, like $A_i \sqsubseteq A_j$, where A_i and A_j are concept names.

We suspected that our classifier had problems learning to classify entailment because it was unable to capture pairwise correlations between embedding dimensions. We researched possible solutions and found that using the outer product to create an interaction map between embedding vectors helps neural networks learn high-order correlations [28, 29].

In our case, adding the outer product between embeddings to the classifier inputs, allowed even a small network to perfectly memorize a training data set consisting only of entailment queries for these simple axioms, when it could not do so with a simple concatenation of embeddings.

It must be pointed out that the reasoner head does not explicitly use terminological axioms for entailment classification. Instead, the terminological axioms indirectly shape the learned concept embeddings, so even though it may look like the reasoner head is simply memorizing answers, some form of deductive reasoning is actually performed.

3.2.2 Embedding layer

Recursive neural networks have been successfully used for encoding expression trees as fixed-size vectors, that could be used as inputs to machine learning models [26]. The recursively defined DL concepts also have a tree structure [30], so it is appropriate to use a recursive neural network as the embedding layer in our reasoner architecture. The key hypothesis that defines our reasoner is that for each KB, we can train an embedding layer that embeds concepts from that KB in an embedding space with a topology that makes it easy for the reasoner head to classify entailment. An embedding topology that is beneficial to entailment classification is formed by jointly training the reasoner head and embedding layers for multiple KBs, which forces the embedding head to generalize, and the embedding layers to output embeddings in the shared embedding space.

In general, concept embeddings are represented by vectors in real vector space $\mathbb{R}^{N_e}$, where N_e is the embedding dimension. We now define the function $h_{\mathcal{K}} : \mathcal{ALC} \to \mathbb{R}^{N_e}$ that maps $\mathcal{ALC}$ concepts for knowledge base $\mathcal{K}$ to embedding vectors in real vector space $\mathbb{R}^{N_e}$.

To obtain the embedding for the complement of a given concept, one first recursively obtains the embedding of that concept, and then passes it as an input to the *complement constructor network* $NN_{\neg}$. The complement constructor network is a two-layer neural network, with N_e input neurons and N_e output neurons with the tanh activation function.

To obtain the embedding for an intersection of concepts, one first recursively obtains the embeddings of the two concepts, computes the interaction map for these embeddings, and passes it as the input to the *intersection constructor network* $NN_{\sqcap}$, that maps the interaction map of two concept embeddings, to the embedding of their intersection. The function $NN_{\sqcap}$ is a two-layer neural network, with $2N_e + N_e^2$ input neurons (because an interaction map is a concatenation of two embeddings of size N_e, and of their flattened outer product, which has size N_e^2), and N_e output neurons, followed by the tanh activation function.

To obtain the embedding for an existential restriction, one first obtains the embedding of the given concept, and passes it to the *existential restriction constructor network* $NN_{\mathcal{K},R_i}$. The

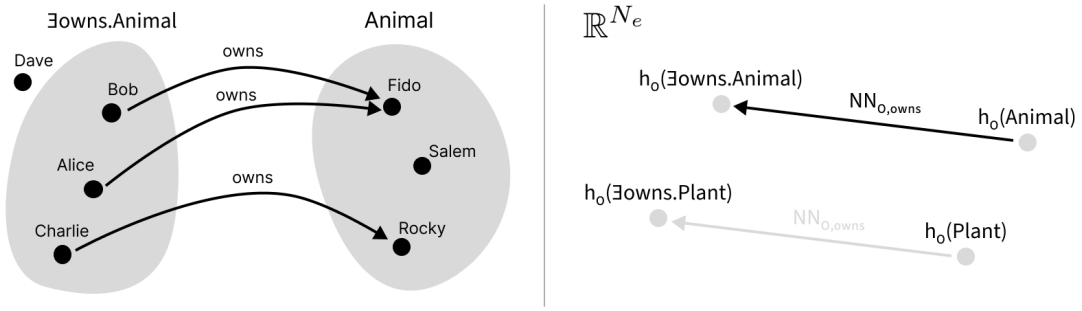

FIGURE 3.2: An example interpretation of a knowledge base with role assertions is shown on the left side. The individuals `Fido`, `Salem`, and `Rocky` are elements of the set $\texttt{Animal}^{\mathcal{I}}$. Individuals `Alice`, `Bob`, and `Charlie` are elements of set $(\exists\texttt{owns.Animal})^{\mathcal{I}}$. An illustration of transformations from concept embeddings to existential restriction embeddings is shown on the right side. The embedding of $\exists\texttt{owns}.C$ may be obtained by applying $NN_{\mathcal{K},\texttt{owns}}$ to the embedding of concept C.

existential restriction constructor network is an a two-layer network, with an input layer with N_e neurons, and an output layer with N_e neurons followed by the tanh activation function.

To understand the intuition behind this choice consider the following example. On the left side of Figure 3.2 we show an interpretation of a KB with a set of human individuals, and a set of individuals that are instances of concept `Animal`. Some individuals own an animal, which is described with role assertions $\texttt{owns}(\texttt{Alice},\texttt{Fido}), \texttt{owns}(\texttt{Bob},\texttt{Fido}), \texttt{owns}(\texttt{Charlie},\texttt{Rocky})$ (shown as arrows). `Alice`, `Bob` and `Charlie` may be described as instances of $\exists\texttt{owns.Animal}$. Notice that if we reversed all `owns` arrows, then an `owns` arrow would always start in set `Animal`, and end in set $\exists\texttt{owns.Animal}$. By definition $(\exists R.C)^{\mathcal{I}} = \{x \in \Delta^{\mathcal{I}} \mid \exists_{y\in\Delta^{\mathcal{I}}}((x,y) \in R^{\mathcal{I}} \wedge y \in C^{\mathcal{I}})\}$, all elements of $(\exists R.C)^{\mathcal{I}}$ are related to some element of $C^{\mathcal{I}}$. So in general, if we interpret role assertions as arrows, for any interpretation $\mathcal{I}$, if we reverse role R_i arrows, they always begin in set $C^{\mathcal{I}}$ and end in set $(\exists R_i.C)^{\mathcal{I}}$. Now consider the right side of the figure. Our embedding layer represents concepts as points in the embedding space $\mathbb{R}^{N_e}$ and does not support individuals at all, so we "squish" the sets $\exists\texttt{owns.Animal}$ and `Animal` into points in $\mathbb{R}^{N_e}$, but we keep the (reversed) arrows, since their start and end points always occur inside the sets. Because of our previous observation about roles, we think that it is appropriate to model the existential restriction constructor as a function mapping concept embeddings $h_{\mathcal{K}}(C)$ to embeddings of the existential restriction $h_{\mathcal{K}}(\exists R_i.C)$. Functions $NN_{\mathcal{K},R_i}$ need to be learned per knowledge base $\mathcal{K}$, because different KBs have different numbers of roles, and roles may have different meanings, so there is no obvious way of sharing existential restriction constructors between KBs.

The embedding for each concept name A_i is simply stored in a vector $\mathbf{W}_{\mathcal{K},A_i}$ of dimension N_e. The concept name embeddings are KB-specific, since different KBs have a different number named concepts with different meanings.

The embedding of the top concept is stored as an KB-specific vector $\mathbf{W}_{\mathcal{K},\top}$, because the most general concepts in a given KB should be close to the top concept in the embedding space, but depending on the vocabulary it might make sense for these general concepts to occupy a different region of the embedding space.

The embedding of the bottom concept is stored by the vector $\mathbf{W}_{\bot}$. We do not see a reason to make the embedding of the bottom concept $\mathbf{W}_{\bot}$ dependent on the KB.

To ensure that we do not combine multiple linear transformations, the outputs of the constructor networks are followed by the tanh activation function. To keep concept embedding values in the same range, regardless of whether the input concept was a concept name, top concept, bottom concept, or a concept expression, we also apply the tanh function to $\mathbf{W}_{\mathcal{K},A_i}$, $\mathbf{W}_{\mathcal{K},\top}$, and $\mathbf{W}_{\bot}$.

$$
\begin{aligned}
h_{\mathcal{K}}(\neg C) &= \tanh NN_{\neg}(h_{\mathcal{K}}(C)) \\
h_{\mathcal{K}}(C \sqcap D) &= \tanh NN_{\sqcap}(IM(h_{\mathcal{K}}(C), h_{\mathcal{K}}(D))) \\
h_{\mathcal{K}}(\exists R_i.C) &= \tanh NN_{\mathcal{K},R_i}(h_{\mathcal{K}}(C)) \\
h_{\mathcal{K}}(A_i) &= \tanh \mathbf{W}_{\mathcal{K},A_i} \\
h_{\mathcal{K}}(\top) &= \tanh \mathbf{W}_{\mathcal{K},\top} \\
h_{\mathcal{K}}(\bot) &= \tanh \mathbf{W}_{\bot}
\end{aligned}
\tag{3.3}
$$

It is not necessary for the embedding layer to support the constructors for concept unions, and universal restrictions. Since the de Morgan laws hold in $\mathcal{ALC}$, one can express concept union in terms of concept complement and intersection. One can also use the quantifier negation law to express universal restrictions in terms of concept complement and existential restrictions [5].

$$
\begin{aligned}
C \sqcup D &\equiv \neg(\neg C \sqcap \neg D) \\
\forall R.C &\equiv \neg\exists R.\neg C
\end{aligned}
\tag{3.4}
$$

Using the above equivalences, we define the embeddings for concept unions and universal restrictions as follows.

$$
\begin{aligned}
h_{\mathcal{K}}(C \sqcup D) &= h_{\mathcal{K}}(\neg(\neg C \sqcap \neg D)) \\
h_{\mathcal{K}}(\forall R_i.C) &= h_{\mathcal{K}}(\neg\exists R_i.\neg C)
\end{aligned}
\tag{3.5}
$$

3.2.3 Relaxed architecture

It is important to note that in the embedding layer, that we introduced in the previous section, the weights of the neural networks $NN_{\neg}$ and $NN_{\sqcap}$ are not dependent on a specific knowledge base. Since the semantics of concept complement and intersection constructors do not change depending on a KB, we think that their neural equivalents should also not change. We call a reasoner, where $NN_{\neg}$ and $NN_{\sqcap}$ network weights are stored in the reasoner head the *restricted reasoner*. Unless stated otherwise, we use restricted reasoners.

However, we did initially experiment with a variant of the reasoner, that allows the embedding layer to learn KB-specific concept constructor networks $NN_{\mathcal{K},\neg}$ and $NN_{\mathcal{K},\sqcap}$, and a KB-specific embedding of the bottom concept $\mathbf{W}_{\mathcal{K},\bot}$. We call this variant the *relaxed reasoner*. All changes to the function $h_{\mathcal{K}}$ in the relaxed reasoner are marked below in red.

$$
\begin{aligned}
h_{\mathcal{K}}(\neg C) &= \tanh NN_{\mathcal{K},\neg}(h_{\mathcal{K}}(C)) \\
h_{\mathcal{K}}(C \sqcap D) &= \tanh NN_{\mathcal{K},\sqcap}(IM(h_{\mathcal{K}}(C), h_{\mathcal{K}}(D))) \\
h_{\mathcal{K}}(\bot) &= \tanh \mathbf{W}_{\mathcal{K},\bot}
\end{aligned}
\tag{3.6}
$$

In the relaxed architecture, the only weights shared between different KBs are the ones learned by the classifier $NN_{\sqsubseteq}$, which we expected would allow the embedding layers to learn more fitting embeddings, at the cost of limiting generalization and opportunities for transfer learning.

3.2.4 Training procedure

The training procedure for our reasoner consists of two steps.

In the first step we train both the reasoner head and embedding layers for as many diverse KBs as possible. This results in a reasoner head that learned to classify whether subsumption axioms hold in any KB, given that an appropriate embedding layer is provided. By appropriate embedding layer we mean a layer that learned to embed KB-specific concepts in a space that minimizes the classifier loss. We simply call this step *training the reasoner head*, and we consider the data used in this step as the *training data set*. Note that as a result of training the reasoner, we obtain trained embedding layers for KBs in the training data set. If obtaining the trained embedding layers was the goal, then the next step is not necessary.

In the second step, we freeze the reasoner head and train the embedding layers for KBs that were not seen in the first step. This results in embedding layers that can embed concepts in a space in which the reasoner is good at classification. We think that if our reasoner can accurately classify whether subsumption axioms are entailed by a KB, then the embeddings used as the input to the classifier $NN_{\sqsubseteq}$ faithfully capture the semantics of the KB, *even though we did not train the reasoner head at all*. We call this step *training the embedding layers* or *testing the reasoner head*, and we consider the data set used in this step as the *test data set*.

3.2.5 Axiom generator

Our reasoner learns embeddings by learning to classify entailment queries $\mathcal{K} \models C \sqsubseteq D$. A set of query axioms could be created manually, but of course, learning will be successful only if a large number of subsumption axioms is provided. One way of obtaining a large number of axioms would be to generate all possible axioms for a given KB, up to some maximum expression tree depth. This approach would suffice for shallow expression trees and KBs with very small vocabularies, but is impractical otherwise. A more practical method of obtaining a large set of axioms is to pseudo-randomly generate them. The most important advantages of pseudo-randomly generating expression trees are that it is fast and that the number of training examples is unlimited. Another advantage is the ability to manually set the initial state of the pseudo-random number generator. One can easily recreate a data set by setting the seed value, which makes storing data sets on disk unnecessary. One drawback of pseudo-randomly generating training data is the possibility of generating duplicate axioms which may need to be discarded, depending on how the reasoner is trained.

One algorithm for pseudo-randomly generating $\mathcal{ALC}$ expressions (including axioms) was introduced by Eberhart et al. [31]. We could not use their implementation, because it uses the Java Virtual Machine, so we implemented our own custom generator inspired by their algorithm.

Our axiom generator has four parameters: the set of concept names N_C, the set of role names N_R, the probability p_A of generating a concept name instead of a top or bottom concept, and the maximum depth of recursion d_{max}. Expressions are generated recursively and the current recursion depth d is tracked.

Axioms are generated by starting with the grammar rule **A** (Equation 3.7), which returns a subsumption axiom or a disjointness axiom with equal probability. The second alternative is intentionally redundant, because disjointness axioms were generated too rarely without it, even though they are common in real-world KBs (generating more disjointness axioms helped us achieve good results in chapter 4, where the analyzed KB had many disjoint concepts). Regardless of which alternative was returned, two concepts are generated according to grammar rule **C** (Equation 3.8). The maximum depth of recursion for the first concept $d_{max,1}$ is chosen from the uniform discrete distribution $\mathcal{U}\{1, d_{max}\}$, and the maximum depth of recursion for the second concept $d_{max,2}$ is chosen from $\mathcal{U}\{1, \max\{1, d_{max} - d_{max,1}\}\}$. This technique of limiting the expression depth results

in axioms where one of the concepts is deeper than the other, or axioms where both concepts are relatively shallow. If both concepts in an axiom are deep, then inference can be very slow in FaCT++, which is problematic when computing answers for tens of thousands of axioms.

$$\mathbf{A} ::= \mathbf{C} \sqsubseteq \mathbf{C} \mid \mathbf{C} \sqcap \mathbf{C} \sqsubseteq \bot \tag{3.7}$$

The grammar rule **C** returns one of the following alternatives with equal probability: a concept according to grammar rule **T** (Equation 3.9), a concept complement, a concept intersection, a concept union, an existential restriction or a universal restriction. For existential and universal restrictions, the role name R_i is chosen randomly from N_R, with equal probability for each role name. If the recursion depth is equal to or greater than d_{max}, then a concept according to grammar rule **T** is always returned.

$$\mathbf{C} ::= \mathbf{T} \mid \neg\mathbf{C} \mid \mathbf{C} \sqcap \mathbf{C} \mid \mathbf{C} \sqcup \mathbf{C} \mid \exists R_i.\mathbf{C} \mid \forall R_i.\mathbf{C} \tag{3.8}$$

The **T** grammar rule returns one of the following alternatives: with probability p_A, a concept name A_i chosen uniformly at random from N_C, the top concept with probability $\frac{1-p_A}{2}$, or the bottom concept with probability $\frac{1-p_A}{2}$. It is crucial that the probability of returning $\top$ or $\bot$ does not depend on the number of concept names $|N_C|$.

$$\mathbf{T} ::= A_i \mid \top \mid \bot \tag{3.9}$$

3.3 Experiment 1 – Transfer test

To test our reasoner we first examine the ability of the reasoner head to correctly predict whether a KB entails subsumption axioms, and by extension, how good are the embeddings learned by the embedding layer.

Then we test if a reasoner head trained on KBs in the training data set, generalizes to unseen KBs from the test data set. The transfer test is described in the next section.

In this experiment we examine both the relaxed reasoner architecture and the restricted reasoner architecture, and compare the advantages and disadvantages of the two variants.

3.3.1 Transfer test

We designed a simple test to check whether a trained reasoner head actually learns to reason in the $\mathcal{ALC}$ description logic, and high classification metric values are not just the effect of embedding layers overfitting to KBs.

We test our architecture by first training the reasoner head and embedding layers on the training data set (as usual), which results in a skillful reasoner. Then we create a no-skill reasoner head, that is not trained, but only randomly initialized. For each of the two reasoner heads, we train embedding layers on the test data set, but keep the reasoner head weights frozen. After training the embedding layers on the test data set for a set number of epochs, if the classification metrics for the reasoner with the trained head are significantly greater than for the reasoner with the random head, then the trained reasoner head learned useful relations in the $\mathcal{ALC}$ embedding space that generalize to new KBs. In short, the reasoner head is transferable.

Conversely, if the differences between classification metrics of both reasoner heads are not significant, then the trained reasoner actually has little or no skill, and the only skill in classification comes from the embedding layer, which would mean that the reasoner head is not transferable.

3.3.2 Synthetic data set

Finding a large number of $\mathcal{ALC}$ KBs with a small number of concept and role names proved difficult, so for this experiment we generated a synthetic data set.

The data set for this consists of 60 randomly generated KBs. We wanted the KBs to be non-trivial, so the number of concept names $|N_C|$ for each KB is chosen randomly from the discrete uniform distribution $\mathcal{U}\{80, 120\}$, and the number of role names $|N_R|$ is chosen randomly from the discrete uniform distribution $\mathcal{U}\{1, 5\}$. The number of role names is very small, because we observed that the number of role names in real-world KBs is typically much smaller than the number of concept names. The actual concept and role names are simply ordinal numbers, and are not considered in any way during learning.

For each KB we also randomly choose the number of terminological axioms $|\mathcal{T}|$ to generate, from the normal distribution $\lfloor\mathcal{N}(200, 10)\rfloor$. We decided that the average number of axioms per KB should be about 2 times bigger than the number of concept names, but should vary a bit. That said, the number of axioms should not be too small nor too big, because when we generate entailment queries later, the probability of randomly generating an axiom entailed by a KB would be too big for KBs with a small proportion of axioms per concept name, or too small for KBs with a large number of axioms per concept name, and that would lead to a strong class imbalance, which is of course undesirable.

Terminological axioms are randomly generated according to the procedure that we described in subsection 3.2.5. We set the maximum depth of axioms d_{max} to 3, because we wanted to keep axioms simple, and thus more general, and with more influence on the interpretation of the KB. The parameter p_A is chosen randomly for each KB from the uniform distribution $\mathcal{U}[0.9, 1]$, as the probability of $\top$ and $\bot$ in a concept expression should be low.

Every time a new axiom is generated for a KB we test if adding it to $\mathcal{T}$ would make the KB inconsistent or if it would make more than 10% of named concepts unsatisfiable (unsatisfiable concepts do occur in real-world KBs, but reasoning in KBs with too many unsatisfiable concepts would be too easy, so we set an upper limit). If so, the problematic axiom is discarded, and we try generating a new axiom.

For each knowledge base $\mathcal{K}$ we additionally generate 2000 unique random queries $\mathcal{K} \models \alpha$, where α is a random subsumption axiom. The same parameters of the random axiom generator are used, as during generating terminological axioms for that KB. For each query we perform the entailment check $\mathcal{K} \models \alpha$ with FaCT++. If the KB entails α, then the expected class of the query is set to $y = 1$. Otherwise the expected class of the query is set to $y = 0$.

After generating KBs and their query sets, we split the data set into the training data set and the test data set. The first 40 KBs and their queries are assigned to the training data set, and the last 20 KBs and their queries are placed in the test data set. We then create a validation data set from 20% of the queries from the training data set. In total, there are 64000 queries in the training data set, 16000 queries in the validation data set, and 400000 queries in the test data set. In every data set, approximately 21.5% of queries have class $y = 1$ and the remaining queries have class $y = 0$. This class imbalance does not significantly affect our experiments, so we do not use any methods for dealing with imbalanced data sets.

KB parameters are sampled from probability distributions instead being set to constant values, to reduce bias in the data set. We do admit that we parameterized the probability distributions mostly according to our intuition, so perhaps a harder data set could be created. However, in retrospect, we think that the data set was hard enough to provide some challenge for reasoners in our experiments.

The data set can be reproduced by setting the initial state of the random number generator to a value specified in the attached code.

3.3.3 Training details

The embedding dimension is set to $N_e = 10$, which is a much smaller number than $|N_C|$. This forces the embeddings for different concept names to share embedding dimensions.

As the loss for our classifier we use binary cross entropy. We create one AdamW optimizer [32] per reasoner head with learning rate set to $\eta_c = 0.0001$, and one AdamW optimizer per embedding layer with learning rate set to $\eta_e = 0.0002$. The AdamW optimizer was the first optimizer we tested, and learning progressed smoothly, so we did not see the need to change it. The learning rates were tuned manually. In total, for each reasoner variant we create 1 optimizer for the reasoner head, and 60 optimizers for the embedding layers.

We train both reasoner variants with mini-batch gradient descent for 15 epochs, which was enough for the validation loss to stop decreasing. During testing, we train the embedding layers (while the reasoner head is frozen) for 10 epochs, as the test loss stabilized after that. We set the batch size to 32, as small batch sizes have been shown to improve generalization [33].

Unless stated otherwise, all weights are randomly initialized using the Xavier initialization [34].

3.3.4 Evaluation metrics

Even though the quality of the embeddings learned by the embedding layer cannot be measured directly, we can still compute classification metrics for a trained reasoner. As mentioned, we think that if the classification metrics indicate good performance, then the embeddings learned by the embedding layer must capture the semantics of a given KB well. If that was not the case and the embeddings did not encode useful information for inference in the $\mathcal{ALC}$ DL, then the classifier should not be able to reliably classify entailment of axioms.

Our classifier learns binary classification, so we chose appropriate metrics [35]. Firstly, we include accuracy (Equation 3.10) in the set of evaluation metrics. Because the data set that we generated for the experiments is slightly imbalanced we also compute precision (Equation 3.11), recall (Equation 3.12), and the F1-score (Equation 3.13).

For simplicity, in threshold-sensitive metrics we choose a threshold of 0.5. In other words, for query $\mathcal{K} \models \alpha$, if the probability of entailment according to the reasoner is greater than the threshold $P(\mathcal{K} \models \alpha) \geq 0.5$, then the predicted class for the query is $\hat{y} = 1$. Otherwise, the predicted class is $\hat{y} = 0$. Given that y is the expected class, in the equations below, TP is defined as the number of true positives (number of queries, where $\hat{y} = y = 1$), TN is the number of true negatives (number of queries, where $\hat{y} = y = 0$), FP is the number of false positives (number of queries, where $\hat{y} = 1$ and $y = 0$), and FN is the number of false negatives (number of queries, where $\hat{y} = 0$ and $y = 1$).

$$\text{Accuracy} = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{TN} + \text{FP} + \text{FN}} \tag{3.10}$$

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}} \tag{3.11}$$

$$\text{Recall} = \text{TPR} = \frac{\text{TP}}{\text{TP} + \text{FN}} \tag{3.12}$$

$$\text{F1} = \frac{2 \cdot \text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}} \tag{3.13}$$

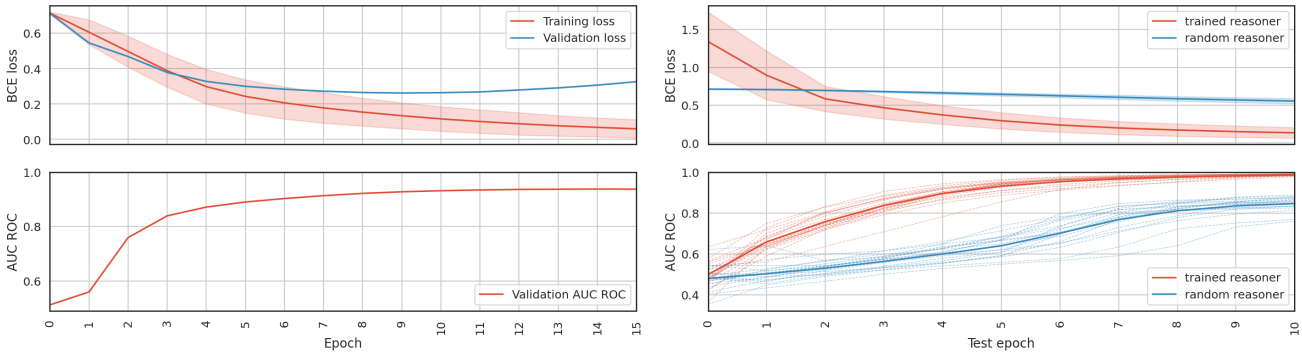

FIGURE 3.3: Training and test progress of the relaxed reasoner. The reported training loss for each epoch is the average mini-batch loss in that epoch. We also show the standard deviation of the mini-batch loss for each epoch. In test progress, the trained reasoner is shown in red, while the random reasoner is shown in blue. The reasoner was *not* trained during epoch 0 of training and testing – during epoch 0 we only compute the initial loss and metric values. Dashed lines show the AUC ROC for each KB in the test data set, while the thicker blue and red curves are the averaged AUC ROC across KBs in the test data set.

Additionally, we visualize the receiver operating characteristic (ROC) curve, the precision-recall (PR) curve, and the confusion matrix (the number for each cell of the confusion matrix is normalized by dividing it by the total number of samples in the data set). The ROC curve shows the performance of a classification model at all classification thresholds, by plotting the true positive rate (TPR; also called recall) against the false positive rate (FPR) (Equation 3.14). Similarly, the PR curve shows the performance of a model at all thresholds, but by plotting precision against recall [36]. For both ROC and PR curves, larger area under the curve suggest a better classifier.

We also compute the value of the area under the ROC curve (AUC ROC), which is a threshold-invariant metric with a minimum value of 0 and a maximum value of 1. Higher AUC ROC values indicate better classification performance [37].

$$\mathrm{FPR} = \frac{\mathrm{FP}}{\mathrm{FP} + \mathrm{TN}} \tag{3.14}$$

The test data set contains 20 KBs, with 2000 queries for every KB. We compute all metrics separately for each query set, and then compute the average and standard deviation across KBs. In the case of the ROC and PR curve plots, in addition to plotting the curves for each query set, we also plot the averaged curves. We do this because the KBs in the test data set may be unequally difficult to classify, so it is beneficial to measure the variance of the reasoner performance across different KBs.

We think that together, AUC ROC, accuracy, precision, recall, F1-score, ROC analysis, PR analysis, and confusion-matrix analysis should provide an accurate image of the performance of the evaluated reasoners.

3.3.5 Evaluation of reasoning ability

We evaluate the reasoning ability of our reasoner by monitoring the training and validation loss and AUC ROC values during training. The only goal of this assessment is to verify that the reasoner can effectively learn to classify entailment in the training data set, and does not suffer from underfitting. Training and validation losses steadily decreasing, and validation AUC ROC increasing with each training epoch, suggests that the reasoner architecture is sufficient for learning to classify entailment in a given data set.

First, we trained the relaxed reasoner on the training data set. The training progress for this variant is shown in Figure 3.3. The training loss decreases and the validation AUC ROC increases

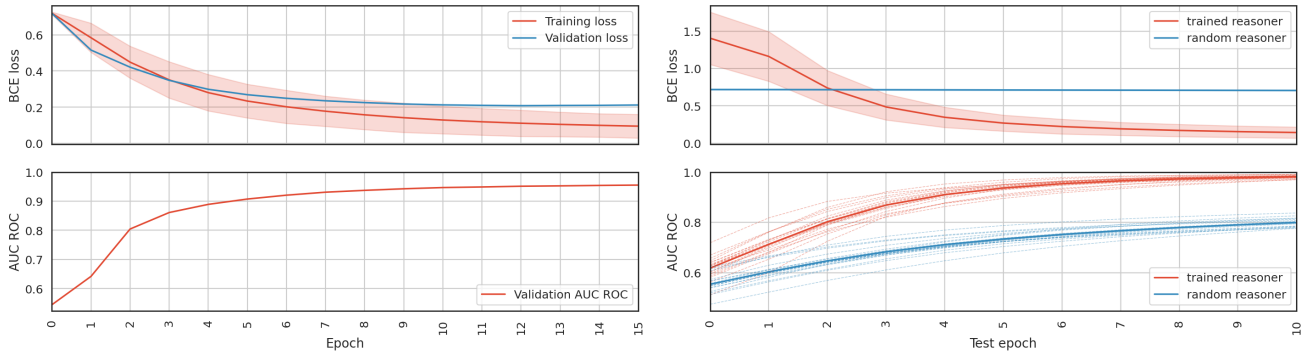

FIGURE 3.4: Training and test progress of the restricted reasoner. Note that the random reasoner loss actually decreases, but a very slow rate.

for the entirety of the training, with smaller gains after epoch 10. The validation loss decreases until epoch 10, and then start increasing, which indicates overfitting.

As shown in Figure 3.4, the AUC ROC metric for the validation set increases when training the restricted reasoner, and does so even faster than when training the relaxed reasoner. We attribute the faster convergence to the shared concept constructor networks $NN_{\neg}$, $NN_{\sqcap}$, which effectively have n times more training samples, where n is the number of training KBs, because they are trained on every sample from every KB, compared to the relaxed architecture, in which KB-specific concept constructor networks are trained on samples from one KB. Furthermore, the validation loss does not start increasing in later epochs, which suggests that the restricted reasoner variant is less prone to overfitting than the relaxed reasoner. The constraint that the concept constructor networks must be the same for each KB does not lower the classification metrics much, while leading to better generalization by virtue of constructors not being KB-dependent.

In this experiment our reasoner (in both variants) was able to learn on the training data set, without extensive hyperparameter tuning. However, if one should notice that the reasoner is underfitting on their data set, and if tuning learning rates does not help, then one should increase the number of hidden layers, or the number of neurons per hidden layer in the $NN_{\sqsubseteq}$ network to make the reasoner head more powerful. If modifying $NN_{\sqsubseteq}$ does not yield better results, then one should try to gradually increase the embedding dimension N_e, up to the maximum number of concept names for a KB in the data set.

3.3.6 Evaluation of knowledge transfer

After training the relaxed reasoner, we froze the reasoner head, and trained embedding layers on the test data set. We also trained embedding layers in conjunction with a randomly initialized reasoner head. The test progress is shown on the right side of Figure 3.3. At the beginning, the test loss for the trained reasoner head was higher than the test loss for the randomly initialized reasoner head. The test loss decreased quickly for the reasoner with the trained head, while the loss for the random head decreased very slowly. For the reasoner with the trained head, the test AUC ROC quickly increased to almost 0.8 after epoch 2, and approached 1 after the last epoch. The average test AUC ROC for the random head slowly increased from around 0.5 in the beginning to around 0.6 in epoch 5. After epoch 5, the AUC ROC for the random head had a relatively big increase to around 0.8 in the last epoch.

Overall, training embedding layers for the relaxed reasoner with the trained head was much faster, than for the reasoner with the randomly initialized head, as the reasoner with the trained head achieved average AUC ROC greater than 0.8 after epoch 3, while it took the reasoner with

TABLE 3.1: Combined test data set metrics for the relaxed and restricted reasoners. Metric values were averaged across different KBs in the test data set. In addition to averages, we standard deviation values are shown.

Model	AUC ROC	F1	Accuracy	Precision	Recall
Relaxed (trained head)	0.9867 ± 0.0042	0.8998 ± 0.0172	0.9572 ± 0.0074	0.9715 ± 0.0191	0.8385 ± 0.0259
Relaxed (random head)	0.8461 ± 0.0343	0.6776 ± 0.0464	0.8423 ± 0.0372	0.6544 ± 0.0773	0.7131 ± 0.0602
Restricted (trained head)	0.9816 ± 0.0064	0.8891 ± 0.0219	0.9525 ± 0.0106	0.9628 ± 0.0144	0.8262 ± 0.0298
Restricted (random head)	0.7995 ± 0.0164	0.3876 ± 0.0174	0.2769 ± 0.0144	0.2406 ± 0.0134	0.9971 ± 0.0024

the random head 10 epochs to do the same. Moreover, the trained head allowed the reasoner to achieve average AUC ROC close to 1 on the test data set after 10 epochs, while the reasoner with the random head only achieved average AUC ROC of around 0.8.

Similarly to the relaxed reasoner, after training the restricted variant, we froze the reasoner head, and trained embedding layers on the test data set. We also trained embedding layers in conjunction with a randomly initialized restricted reasoner head. The test progress for the restricted reasoner is shown on the right side of Figure 3.4. The test loss for the restricted reasoner with the trained head decreased similarly to the relaxed reasoner with the trained head. However, the test loss for the restricted reasoner with the randomly initialized head decreased so slowly, that the test loss curve seems stationary on the plot (we checked that the loss indeed decreased, but extremely slowly). Even though the test loss decreased very slowly for the random head, the test AUC ROC visibly increased to about 0.8 after the last epoch, which is similar to the final result of the relaxed reasoner with the random head. The average AUC ROC increased very quickly for the reasoner with the trained head, achieving average AUC ROC of about 0.8 after epoch 2, more than 0.9 after epoch 5, and approaching 1 after the last epoch.

3.3.7 Comparative results

The combined test data set metrics, that our reasoners achieved in this experiment, are shown in Table 3.1. We also visualize the ROC and PR curves, and confusion matrices for the test data set for the relaxed reasoner in Figure 3.5, and for the restricted reasoner in Figure 3.6.

As expected, for both relaxed and restricted reasoners, the reasoners with trained heads achieved superior performance on the test data set, which shows that both variants of our reasoner are indeed transferable. For the relaxed architecture, the trained model is strictly better than the randomly initialized one, as all metric values are higher for the former. Moreover, the trained model has lower variance of metrics across KBs in the test data set, than the random model. For the restricted architecture, the trained model is significantly better than the randomly initialized one on all metrics except recall. The extremely high recall of the random reasoner is of course worthless, given its very low precision. The restricted trained reasoner has lower variance than the restricted random reasoner for AUC ROC and accuracy. However, the restricted random reasoner has lower variance for F1-score, precision and recall.

Between the two random reasoners, the relaxed variant achieves better metric values, except recall, due to the very high recall of the restricted random reasoner. This was expected, since in the relaxed variant, the complement and intersection constructor networks can adjust to the randomly classifier to minimize the classification error, while in the restricted variant this is not possible.

When comparing the trained reasoners, the relaxed variant achieves slightly higher values with lower variance for all metrics (except precision, for which the restricted variant has slightly lower variance). We were also not surprised by these results, as we expected that sharing constructor

networks between all KBs would force the reasoner to generalize, which would also increase variance in metric values for KBs in the test data set. The restricted reasoner has a big advantage, that we think negates its slightly lower metric values and slightly higher variance in comparison to the relaxed variant – it has less learnable parameters.

Since one of the main goals of our reasoner is transfer learning, the lower number of parameters in the embedding layer speeds up training for new KBs. The average embedding layer training time per epoch was 21.72 seconds for the restricted variant, and 26.84 seconds per epoch for the relaxed variant, which is 23% slower. For the relatively simple constructor networks that we use in our work, this is not very significant, but we expect the time savings to greatly increase, when these networks are scaled-up to deal with bigger data sets and more complex KBs.

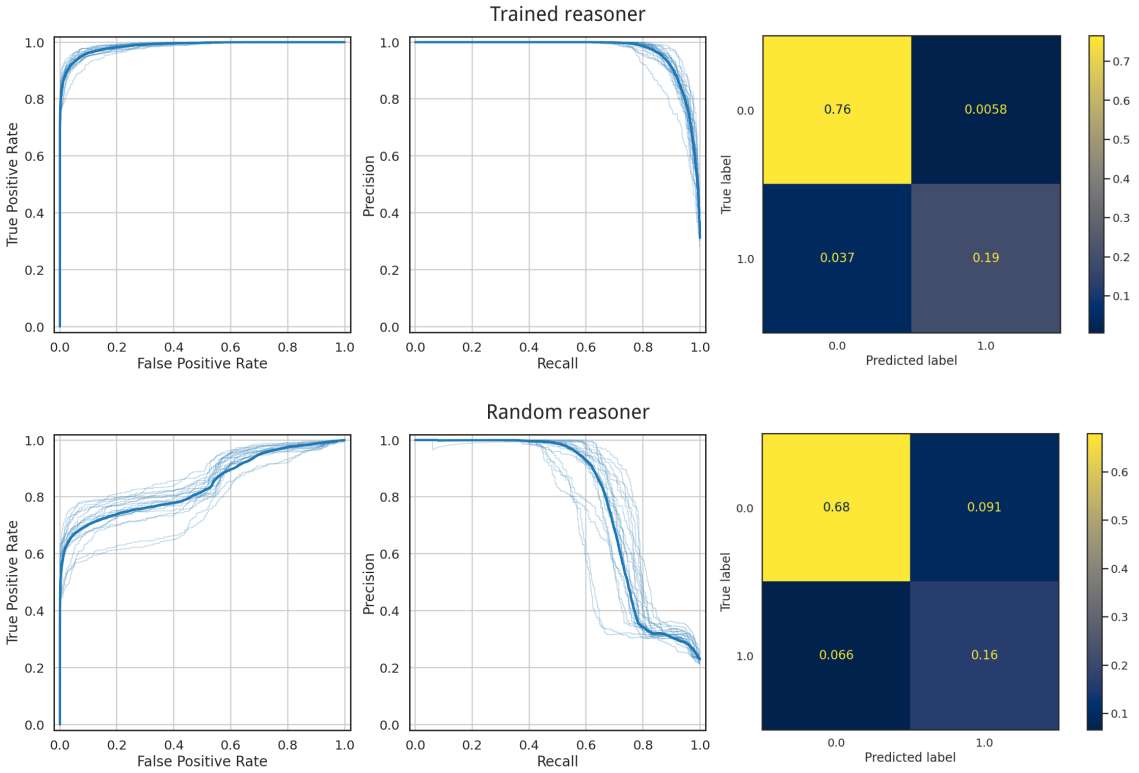

FIGURE 3.5: ROC curves and confusion matrices for the relaxed reasoner. The visualizations reflect classification performance on the test data set. Confusion matrices are normalized by the number of all samples in the test data set.

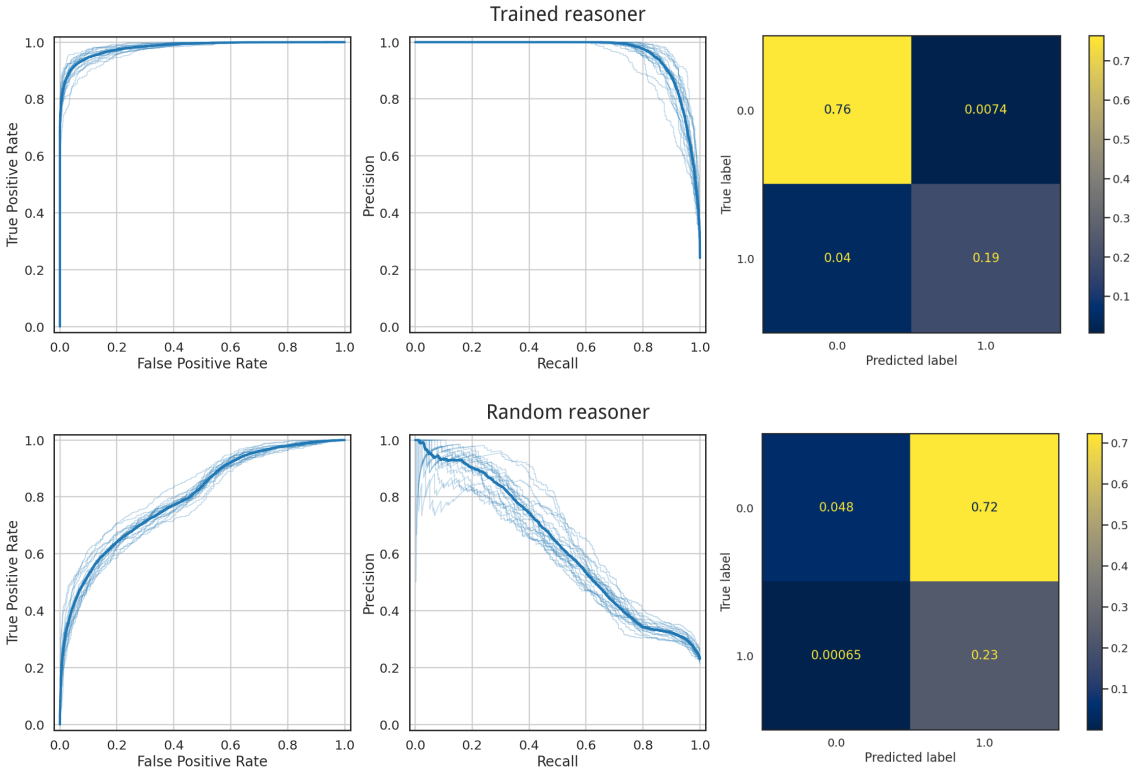

FIGURE 3.6: ROC curves and confusion matrices for the restricted reasoner.

Chapter 4

Case study – pizza ontology

In chapter 3 we evaluated our deep neural reasoner in two ways. Firstly, we showed that our reasoner can learn to accurately classify whether subsumption axioms are entailed by a given KB. Secondly, we demonstrated that pre-training the reasoner head results in shorter fine-tuning time and greater classification accuracy for unseen KBs, when compared to a randomly initialized reasoner head with no pre-training, which meant that reasoner heads are transferable. Because the classification metrics were high, we concluded that the learned embeddings encode information that is useful for classifying entailment. However, since we trained our reasoner on a synthetic data set, the embeddings were not interpretable.

In this chapter we use the pizza ontology[1,2] from the Manchester University OWL Tutorial [38] We chose the pizza ontology, because it has a similar number of axioms, concept and role names to the randomly generated KBs we used in experiment 1. The relatively small number of concept names made it easy to visualize all the learned embeddings, while keeping the visualization readable. The ontology contains various axioms (equivalence and disjointness axioms, universal and existential restrictions, concept intersection, union, and complement), so we could test all capabilities of our reasoner. We describe the pizza ontology in detail in section 4.1.

The pizza ontology is described with OWL DL, so to use our reasoner to learn concept embeddings, it was necessary to remove small parts of the ontology because $\mathcal{SHOIN}^{(\mathcal{D})}$ (the underlying logic of OWL DL) is more expressive than $\mathcal{ALC}$. We describe the process of reading from the RDF/XML format and the steps to transform OWL DL ontologies to $\mathcal{ALC}$ KBs in subsection 4.1.3.

After introducing the pizza ontology, we describe experiments 2 and 3. In experiment 2 we check whether our reasoner is able to learn concept name embeddings of the pizza ontology by learning to classify entailment of subsumption axioms, where both operands are concept names. In experiment 3 we check whether our reasoner is able to learn complex concept embeddings, by learning to classify entailment of subsumption axioms, where both operands are arbitrary concepts. We also checked whether using a pre-trained reasoner head would have a beneficial effect on the learned embeddings. In both experiments, we evaluated the reasoning ability of the trained reasoners according to selected metrics. We also analyze the learned embeddings by visualizing them and subjectively assessing their quality.

[1] As a reminder, *ontology* is an OWL term for a knowledge base. Since the pizza ontology is an OWL document, we use the term *ontology* when referring to it. We still use the term *knowledge base*, when referring to KBs in general.

[2] The pizza ontology in the RDF/XML format is available at `http://owl.cs.manchester.ac.uk/publications/talks-and-tutorials/protg-owl-tutorial/`

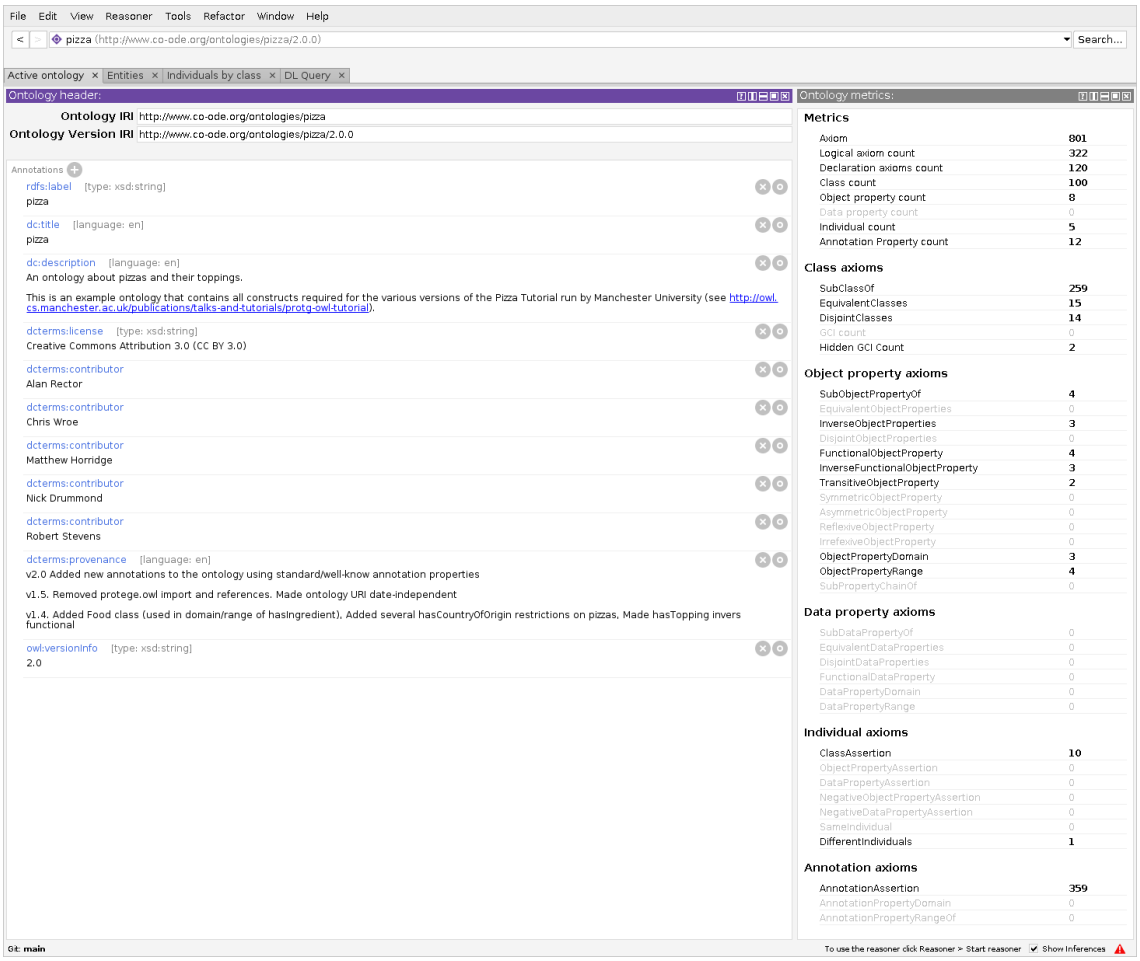

FIGURE 4.1: Summary of the pizza ontology in Protégé. Note the ontology metrics on the right

4.1 Summary of the pizza ontology

To explore the pizza ontology, we used Protégé – the most widely used OWL ontology editor [39]. Specifically, we used version 5.5.0 of Protégé, and version 1.6.5 of the FaCT++ reasoner plugin[3].

4.1.1 Ontology metrics

The summary of the version 2.0 of the pizza ontology is shown in Figure 4.1. We pay attention to metrics, as they are useful for quickly grasping the contents of an ontology. According to Protégé there are 100 classes and 8 object properties which are equivalent to concept names and role names, respectively. There are 15 equivalence axioms and 15 class disjointness axioms (in OWL DL, a `DisjointClasses` axiom allows one to specify that given a set of classes, every pair of classes from that set is disjoint).

The pizza ontology contains 23 object property axioms. Most importantly, properties `hasBase` and `hasTopping` are subclasses of `hasIngredient`. There are also properties `isBaseOf`, `isToppingOf` and `isIngredientOf`, which are inverses of the aforementioned properties. The properties `hasBase`, `hasTopping`, and `hasIngredient` are transitive, and their domain and range are specified. Another often-used property in the pizza ontology is `hasSpiciness`, which is used

[3]Version 1.6.5 of the FaCT++ plugin for Protégé 5 is available at `https://bitbucket.org/dtsarkov/factplusplus/downloads/uk.ac.manchester.cs.owl.factplusplus-P5.x-v1.6.5.jar`

to relate classes to spiciness levels. Some properties are also defined as functional, or inverse functional, which is irrelevant in our case, since these characteristics cannot be expressed in $\mathcal{ALC}$.

There are 12 annotation properties and 359 annotation assertions. We use the `rdfs:label` annotation property in English to obtain the name of a class when parsing the ontology in the RDF/XML format. We also examined the `rdfs:comment` annotations written by the ontology authors, as some of them contained useful explanations. In particular, the authors explain how `IceCream` and `CheesyVegetableTopping` are purposefully made to be unsatisfiable (equivalent to $\bot$). The first class is unsatisfiable because `IceCream` was defined as a subclass of $\exists$`hasTopping`.`FruitToping`, which implies that `IceCream` is a subclass of `Pizza`, since the domain of `hasTopping` was defined to be `Pizza`. However, `IceCream` was also defined as disjoint with `Pizza`, `PizzaBase` and `PizzaToping`, leading to an inconsistency. The class `CheesyVegetableTopping` is unsatisfiable because it is defined to be a subclass of both `CheeseTopping` and `VegetableTopping`, but these toppings were defined to be disjoint, so their intersection is equivalent to $\bot$. Starting the FaCT++ reasoner in Protégé confirms that both classes are equivalent to the bottom concept.

There are also 5 individuals, 10 class assertions, and one set of different individuals. The individuals are `America`, `England`, `France`, `Germany`, and `Italy`, which are instances of the class `Country`. Similarly to disjointness axioms, OWL DL allows one to specify that none of the individuals in a given set are the same with an `DifferentIndividuals` axiom. All individuals in the pizza ontology are defined as different. The `American` and `AmericanHot` pizzas are defined as subclasses of value restriction `hasCountryOfOrigin value America`. Similarly, the object property `hasCountryOfOrigin` is used to relate the individual `Italy` with `Napoletana`, `Veneziana`, `MozzarellaTopping`, and `RealItalianPizza`.

4.1.2 Class hierarchy

We also familiarized ourselves with the class hierarchy of the ontology (shown in Figure 4.2). All classes are of course subclasses of the top concept (`Thing` in OWL nomenclature), but the only direct subclasses of $\top$ are `DomainThing` and `ValuePartition`, which are disjoint.

`Spiciness` is a direct subclass of `ValuePartition`, and is defined as equivalent to the union of three disjoint classes: `Hot`, `Medium`, and `Mild`, which effectively partitions `Spiciness` into three non-overlapping subsets.

`Country` and `Food` are the only direct subclasses `DomainThing` class. `Country` is defined as the set of 5 countries (individuals) in this ontology. `Food` is divided into `IceCream`, `Pizza`, `PizzaBase` and `PizzaTopping`. Not including `DomainThing` and `Food`, the `Pizza` and `PizzaTopping` classes contain the most subclasses in the pizza ontology, with slightly more pizza toppings than pizzas.

`NamedPizza` is a subclass of `Pizza` and has 23 real-world pizzas as subclasses. Other subclasses of `Pizza` are defined with equivalence axioms, so their subclasses are inferred by a reasoner. Notably, there are `VegetarianPizza`, `VegetarianPizza1` and `VegetarianPizza2` classes, which define vegetarian pizzas differently. `VegetarianPizza1` and `VegetarianPizza2` are actually equivalent, and they are inferred to be subclasses of `VegetarianPizza`, but they are not equivalent to it. Another interesting pizza is `UnclosedPizza`, which is defined as a subclass of $\exists$`hasTopping`.`MozzarellaTopping`. A reasoner cannot actually infer that `UnclosedPizza` is a subclass of `VegetarianPizza` or `NonVegetarianPizza`, because due to the open-world assumption (OWA), it may have other, possibly non-vegetarian toppings. If `UnclosedPizza` would be defined as $\forall$`hasTopping`.`MozzarellaTopping`, then it could be classified as a vegetarian pizza. In

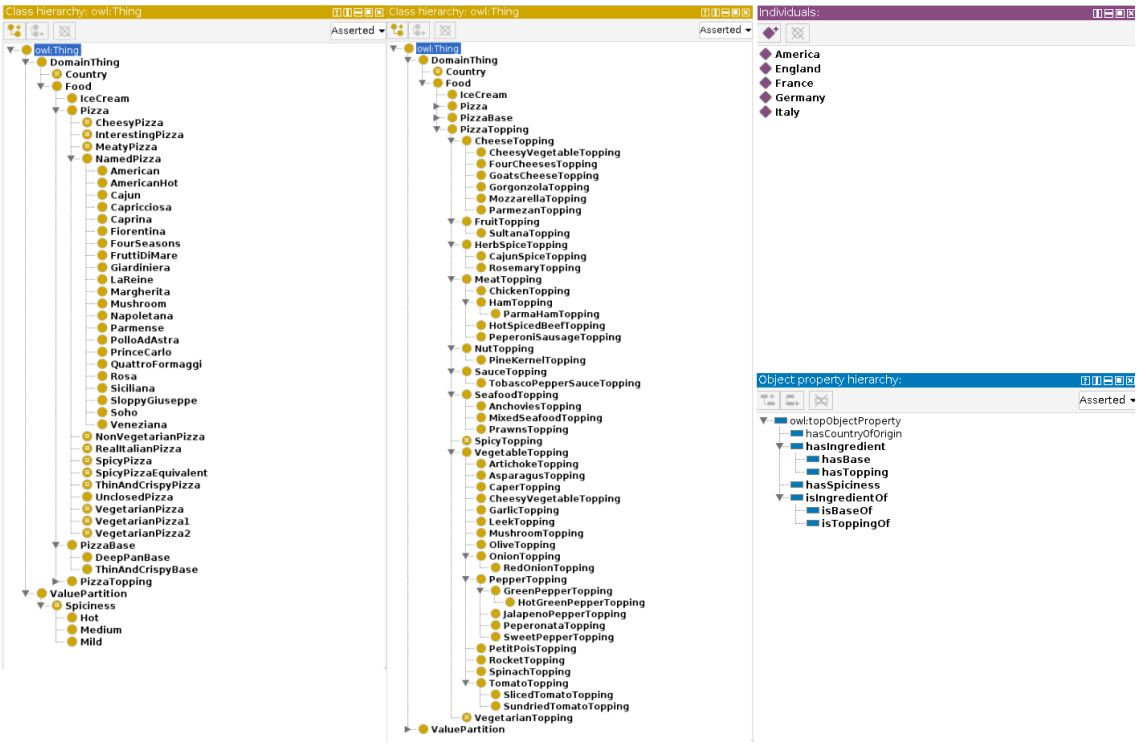

FIGURE 4.2: Class hierarchy, role hierarchy, and individuals in the pizza ontology, shown in Protégé. The entire vocabulary of the ontology is visible.

total, there are 35 subclasses of `Pizza`, 14 subclasses of `VegetarianPizza`, and 15 subclasses of `NonVegetarianPizza`.

The topping hierarchy is relatively simple, as it consists of cheese, fruit, herb/spice, meat, nut, sauce, seafood, and vegetable toppings. The only toppings defined using equivalence axioms are `VegetarianTopping` and `SpicyTopping`. Vegetarian toppings are defined as a union of cheese, fruit, herb/spice, nut, sauce, and vegetable toppings. Spicy toppings are defined as toppings that are a subclass of $\exists$`hasSpiciness.Hot`. In total, there are 51 subclasses of `PizzaTopping`, 39 subclasses of `VegetarianTopping`, 7 subclassess of `SpicyTopping`.

In addition to `PizzaBase`, there are only two disjoint pizza bases: `DeepPanBase` and `ThinAndCrispyBase`. These are rarely used, as `ThinAndCrispyBase` is only used to define `RealItalianPizza` and `ThinAndCrispyPizza`, and `DeepPanBase` is not used in any pizza-related axioms at all.

4.1.3 Ontology processing

The pizza ontology is available online in RDF/XML format. Although we could easily view the ontology in Protégé, using our reasoner to learn concept embeddings requires that the ontology is stored in a format closer to the syntax of $\mathcal{ALC}$. As we mentioned, it is also necessary to remove parts of the ontology to make it compatible with our neural reasoner, because the pizza ontology is described with $\mathcal{SHOIN}^{(\mathcal{D})}$, which is the underlying logic of OWL DL, and is more expressive than $\mathcal{ALC}$.

To parse the pizza ontology, we first use the ROBOT command-line tool to convert it from RDF/XML to the OWL functional-style syntax. After the conversion, we use our parser to load the ontology into our in-memory KB representation. We described our parser and the KB representation in section 3.1.

We do the following pre-processing steps to make OWL ontologies compatible with our reasoner. Pre-processing is done automatically after parsing the ontology file, so one does not need to edit the ontology manually.

- As the string value of concept names, we use the value of the `rdfs:label` annotation property for the English language.
- We ignore object property axioms, since role axioms are not expressible in $\mathcal{ALC}$ (including role hierarchies, inverse roles, transitive roles, and (inverse) functional roles).
- We ignore axioms with number or value restrictions, since they are not expressible in $\mathcal{ALC}$ with KBs where ABox is empty.
- We ignore individuals and ABox axioms, since our reasoner does not support ABox reasoning.
- Since our reasoner does not directly support equivalence axioms $C \equiv D$, we split such axioms into two subsumption axioms $C \sqsubseteq D$ and $D \sqsubseteq C$, as equivalence can be reduced to subsumption [1].
- Our reasoner also does not directly support disjointness axioms, so given a disjointness axiom $\texttt{DisjointClasses}(A_1, A_2, \ldots, A_n)$ in the OWL functional-style syntax, we generate axioms $A_i \sqcap A_j \sqsubseteq \bot$, for each possible pair of concept names (A_i, A_j) from the set of disjoint concepts, where $A_i \neq A_j$.
- We remove roles that do not appear in any TBox axiom kept after pre-processing. This is done because of a limitation in FaCT++, which makes it raise an exception, when constructing a concept with an unused role name.

After processing the pizza ontology, `InterestingPizza` is equivalent to the top concept, because the axiom defining it as a pizza with at least 3 toppings contained a number restriction, so it was removed. The equivalence axiom that defines `RealItalianPizza` as a pizza with Italy as its country of origin was removed because it contained a value restriction. Since `RealItalianPizza` was also defined as a subclass of $\exists$`hasBase.ThinAndCrispyBase`, any pizza with a thin and crispy base is inferred to be a subclass of `RealItalianPizza`. Value restrictions for `American`, `AmericanHot`, `Napoletana`, `Veneziana`, and `MozzarellaTopping` were removed, but that did not have a significant effect on reasoning.

After loading the pizza ontology, we observed that only 99 classes are actually defined in the ontology, instead of 100 reported by Protégé. This can be easily confirmed by saving the pizza ontology with Protégé in the OWL functional-style syntax format and counting the number of `Declaration(Class(...))` expressions in the resulting file. We do not know what in the pizza ontology is counted as the extra class by Protégé, but in this chapter we assume that the number of concept names is $|N_c| = 99$, which is consistent with the source file of the pizza ontology and our ontology processing code.

We used FaCT++ to classify the ontology after processing, and visualized the result as a heat map (shown in Figure 4.3).

4.2 Experiment 2 – Pizza taxonomy

After parsing and processing the pizza ontology, it can be used to test our reasoner. In this experiment, we show that given a simple data set, our reasoner can easily learn good embeddings for concept names.

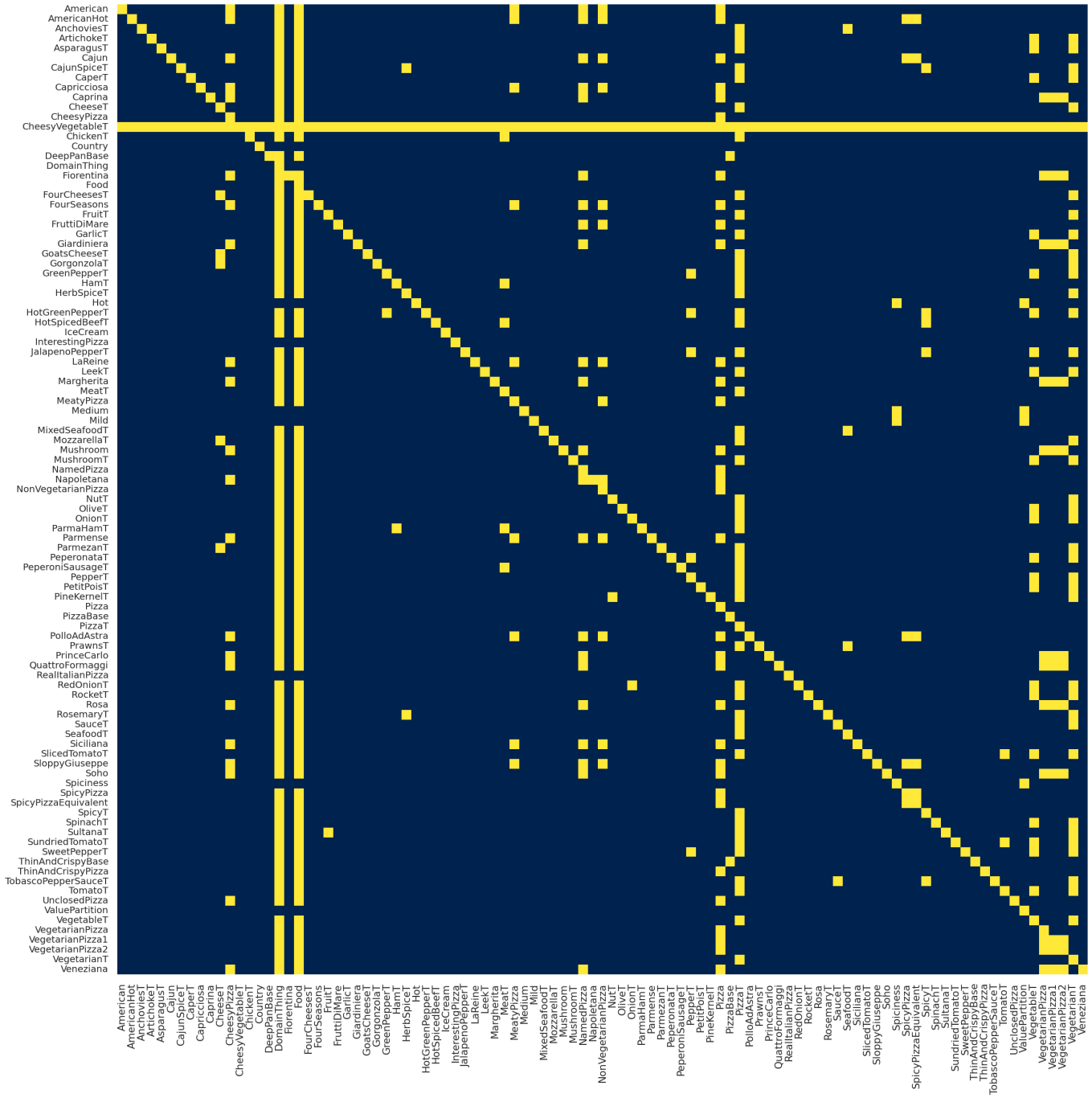

FIGURE 4.3: Results of ontology classification with FaCT++. Each heat map cell shows the result of query $\mathcal{K} \models A_i \sqsubseteq A_j$, where i is the heat map row, j is the heat map column and A_k is the k-th concept name in the pizza ontology in lexicographical order (A_1 being `American` and A_{99} being `Veneziana`). We replaced the word "Topping" in concept names with "T" to save space. A concept that is subsumed by every concept (a concept equivalent to $\bot$) is shown as a yellow horizontal line, and a concept that subsumes every concept (a concept equivalent to $\top$) would be shown as a yellow vertical line.

4.2.1 Data set generator

The data set for this experiment is very simple. It consists of all possible queries $\mathcal{K} \models C \sqsubseteq D$, where C and D are concept names or $\top$ or $\bot$. Since the pizza ontology contains 99 classes, in addition to $\top$ and $\bot$, the number of queries in this simple data set is equal to $(|N_C|+2)^2 = (99+2)^2 = 10201$. The answers to all queries were obtained using the FaCT++ reasoner. We recognize that for any concept C, queries $C \sqsubseteq C$, $C \sqsubseteq \top$, and $\bot \sqsubseteq C$ can be trivially answered. However, because we are training the reasoner for the pizza ontology from scratch, we want it to learn to correctly classify those queries too.

The resulting data set is not balanced, as only about 868 queries ($\approx 8.5\%$) are assigned to class 1, and the remaining 9333 queries ($\approx 91.5\%$) are assigned to class 0. We do not perform undersampling or oversampling, as even with this unbalanced data set, our reasoner learns embeddings that allow it to classify all entailment queries almost perfectly.

4.2.2 Model and training procedure

In this experiment, we train a reasoner using the restricted architecture. We train the reasoner from scratch – that is, we randomly initialize the reasoner head and embedding layer. We pick a small embedding dimension $N_e = 10$, which is significantly smaller than the number of classes in the pizza ontology and forces the reasoner to share embedding dimensions between concepts. We set the size of the hidden layer of the entailment checking neural network $NN_{\sqsubseteq}$ to 16, which is the same as in experiment 1.

With the aforementioned hyperparameters, the reasoner head has 1963 learnable parameters in total ($(2N_e+N_e^2+1)\cdot 16+(16+1)\cdot 1 = 1953$ parameters in $NN_{\sqsubseteq}$, and N_e parameters for the bottom concept), and the embedding layers have 1000 parameters (N_e parameters per concept name, and N_e for the top concept), which is 2953 parameters in total. We did not include parameters of role embeddings $NN_{\mathcal{K},R_i}$, and the intersection and negation constructor networks $NN_{\sqcap}$ and $NN_{\neg}$, as the data set in this experiment does not use them.

We do not split the data set, since we want the learned concept embeddings to fit the data set as well as possible. We train the reasoner and embeddings for 30 epochs. After each training epoch, we calculated classification metrics for the entire data set. We use the AdamW optimizer with learning rate set to the same value for both reasoner head and embedding layer optimizers $\eta_c = \eta_e = 0.002$.

4.2.3 Evaluation of reasoning ability

After training for 30 epochs, the reasoner achieves perfect classification metrics. In total, at the threshold of 0.5, the trained reasoner made 0 mistakes. Because the error rate is 0, we do not show the heat map of the predicted probabilities, as it looked exactly the same as the heat map in Figure 4.3.

Of course, we realize that the $NN_{\sqsubseteq}$ network in the reasoner head simply memorized answers to the queries from the data set. Nevertheless, it was useful to verify that the reasoner can at least find embeddings that are helpful in learning the entailment relation for a simple taxonomy.

4.2.4 Embedding analysis

In addition to evaluating the reasoning ability of our reasoner, which showed that it can successfully learn to classify entailment of subsumption axioms in a "real-world" KB, we visualize the learned embeddings.

A 2D visualization that tries to preserve the distances between concepts in the high-dimensional embedding space enables visual assessment of the quality of the learned embeddings. We think that if semantically similar concepts are placed closer to each other than to dissimilar concepts, then the learned embeddings capture the semantics of a given KB.

Dimensionality reduction

Because we chose 10 as the size of concept embeddings, first we needed to reduce the dimensionality of the embeddings. We used Uniform Manifold Approximation and Projection for Dimension Reduction (UMAP), because it is a nonlinear dimensionality reduction algorithm with interpretable hyperparameters and high performance [40]. We used the Python implementation by the authors of UMAP and other open source contributors[4].

According to UMAP's documentation, the two main hyperparameters are `n_neighbors` and `min_dist`. The first one specifies the number of neighbors to consider during the initial construction of the high-dimensional graph. Low values of `n_neighbors` emphasize the local structure of the data, and high values emphasize the global structure of the data. The second hyperparameter specifies the minimum distance between points in the low-dimensional space. Low values of `min_dist` will result in visualizations where the clusters are tightly packed and high values will spread out the data points.

For our purposes, we wanted to emphasize the global structure of the learned embeddings, so we set `n_neighbors` at a very high value of 50 (considering that we have 101 points in total). To choose the value of `min_dist` we began at 0.99 and decreased the value until the visualization became readable and clusters started to separate. In the end, we set the value of `min_dist` at 0.01.

Coloring scheme

If we visualized all concept embeddings as points sharing the same shape and color, our visualization would not be readable, and it would be hard to judge, whether semantically similar concepts form clusters. To create an effective visualization of the learned embeddings, we created a coloring scheme for concepts in the pizza ontology that tries to make the most important attributes of concepts immediately recognizable. The following description specifies how we use different shapes, sizes, and colors of markers:

- By default concepts are square and gray
- Top concept, bottom concept and concept expression are cyan
- Toppings are shaped like diamonds, and pizzas are round
- Vegetarian pizzas and toppings are light green (except vegetable toppings, which are dark green)
- Seafood toppings are pink
- Non-vegetarian pizzas and meat toppings are dark red
- Cheesy pizzas, and cheese toppings are respectively marked with a yellow disk or yellow diamond inside
- Pepper toppings are marked with an orange diamond inside

[4]Source code for the Python UMAP implementation is available at `https://github.com/lmcinnes/umap`

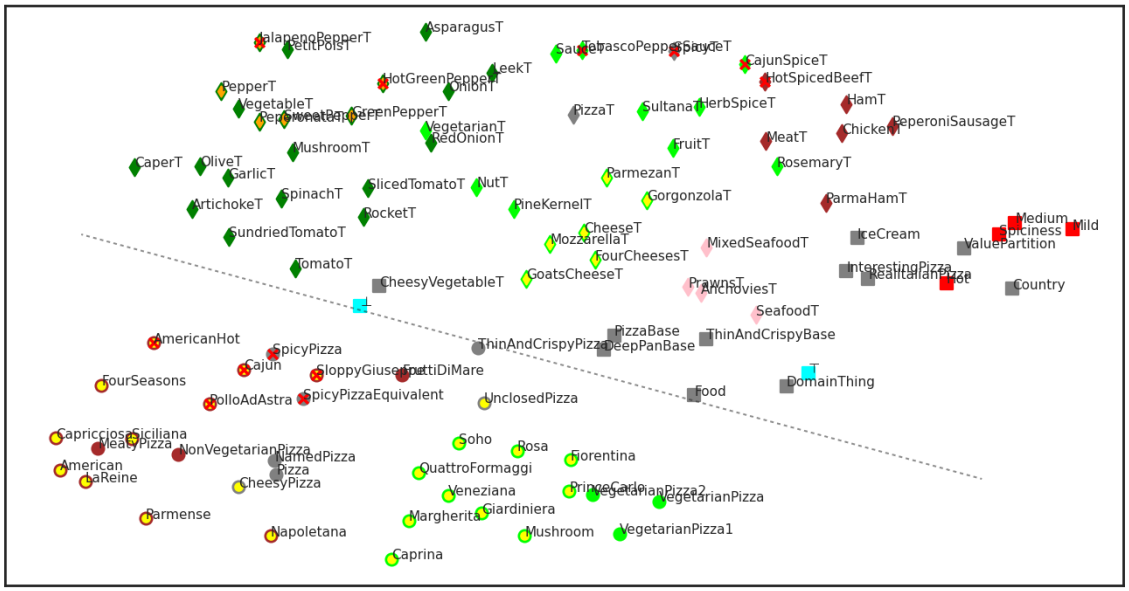

FIGURE 4.4: UMAP visualization of the learned concept embeddings. The shapes and colors of the point markers are decided according to the scheme described in subsection 4.2.4. We replaced "Topping" in concept names with "T" to improve readability. The dashed line separating pizzas from other concept names was added for emphasis.

- Spicy things are marked with a red "x" inside
- Spiciness levels are light red

Visual assessment

The UMAP visualization of the concept embeddings is shown in Figure 4.4. As expected, the unsatisfiable `CheesyVegetableTopping` is close to $\bot$, and the general `DomainThing` concept is close to $\top$. Spiciness and pizza bases form their own clusters. Pizzas and toppings are separated, with vegetarian pizzas forming one cluster, and non-vegetarian pizzas forming another cluster together with spicy pizzas. All spicy pizzas in the pizza ontology are non-vegetarian, so it is understandable that `SpicyPizza` and `SpicyPizzaEquivalent` are close to non-vegetarian pizzas.

Meat toppings are close to each other and relatively close to seafood toppings, which is good. Cheesy toppings, vegetable toppings, spicy toppings, and pepper toppings are also close to related concepts. `CajunSpiceTopping` is close to `HotSpicedBeef`, which is understandable, because they are both spicy toppings. `MeatTopping` and `RosemaryTopping` are also close to each other, which is unexpected.

Overall, based on the visualization, we think the embeddings capture the semantics well, given the simple data set.

4.3 Experiment 3 – Full data set

The previous experiment showed that our reasoner can learn good concept embeddings for simple taxonomies, so the next step is to test whether it can learn good embeddings for arbitrary concepts. In this experiment we do that by training the neural reasoner to classify entailment, given randomly generated queries about the pizza ontology as the data set. We repeat this experiment three times. In the first run, we let the reasoner learn without any pre-training. In the second run, we initialize the reasoner head with weights of the restricted reasoner, that we trained in experiment 1, then

freeze it, and only allow the embedding layer to learn. In the third run, we randomly initialize the reasoner head, and also freeze it to only allow the embedding layer to learn KB-specific embedding.

4.3.1 Training data set

The data set for this experiment consists of 30 000 unique random queries that we generate using the algorithm described in subsection 3.2.5. We set the maximum axiom depth to 4, and the probability of concept names to $p_A = 0.95$. The answer to each query is obtained using FaCT++. Similarly to the last experiment, we do not split the data set because we want the learned embeddings to fit the data set as well as possible. The data set is slightly imbalanced, with 36.97% of queries belonging to class $y = 1$, and the rest to class $y = 0$, although the class imbalance did not significantly affect classification.

4.3.2 Training procedure

As mentioned, we repeat learning three times, which results in three reasoners:

- *Unfrozen reasoner head* – Both the reasoner head and KB-specific embedding layers were trained.
- *Frozen pre-trained reasoner head* – Only KB-specific embedding layers were trained. The reasoner head head was initialized with the weights of the restricted reasoner, that we obtained in experiment 1.
- *Frozen random reasoner head* – Only KB-specific embeddings layers were trained. The reasoner head was initialized randomly.

Each reasoner head has 3283 learnable parameters in total ($(2N_e+N_e^2+1)\cdot 16+(16+1)\cdot 1 = 1953$ parameters for $NN_{\sqsubseteq}$, $(2N_e+N_e^2+1)\cdot N_e = 1210$ parameters for the concept intersection constructor $NN_{\sqcap}$, $(N_e + 1) \cdot N_e = 110$ parameters for the concept complement constructor $NN_{\neg}$, and N_e parameters for the bottom concept), and the embeddings have 1330 parameters (N_e parameters per concept name, N_e for the top concept, and $(N_e + 1) \cdot N_e$ per role), which is 4613 parameters in total.

In every run of the experiment we trained the model for 30 epochs, with learning rate set to $\eta_c = \eta_e = 0.002$ both for the optimizer of the reasoner head and the embedding layers. Similarly to the previous experiments, we use the AdamW optimizer, and train the reasoners with batch size of 32. Because we initialize the frozen pre-trained reasoner head with weights of the restricted reasoner head, that we trained in experiment 1, the embedding dimension N_e and the layers of the neural network $NN_{\sqsubseteq}$ needed to be exactly the same as in that experiment. Thus, for all reasoners in this experiment we set $N_e = 10$ and defined $NN_{\sqsubseteq}$ to be a neural network with one hidden layer with 16 neurons and ELU activation function. The embedding dimension is adequate, since the pizza ontology has a similar number of concepts to the ontologies in the synthetic data set, that we used in experiment 1.

We expected the reasoner with the unfrozen head to achieve best metric values and learn the best embeddings out of the three reasoners in the embedding analysis, because no weights are frozen, which means that the reasoner with the unfrozen head can fit to the data set more than the reasoners with frozen heads. The only obstacle to learning good embeddings for the reasoner with unfrozen head are the randomly generated queries, that may not contain useful entailments for the pizza ontology, although the other two reasoners learn with the same data set, so the comparison is at least fair.

Table 4.1: Classification metrics for reasoners in experiment 3 after training for 30 epochs.

Model	AUC ROC	F1	Accuracy	Precision	Recall	Training time
Unfrozen reasoner head	0.9997	0.9906	0.9931	0.9915	0.9897	349.83s
Frozen pre-trained reasoner head	0.9868	0.9472	0.9612	0.9566	0.9379	290.14s
Frozen random reasoner head	0.7816	0.7372	0.7568	0.6128	0.9249	287.21s

Based on the results of experiment 1, we expected the reasoner with transfer to achieve higher classification metrics than the reasoner with randomly initialized frozen head. In experiment 1, the trained reasoner head was better at classifying queries for the unseen KBs from the test data set, than the randomly initialized reasoner head, so we expected the same to be true for the pizza ontology.

4.3.3 Evaluation of reasoning ability

The classification metrics of the three reasoners are shown in Table 4.1. The expected ontology classification heat map and heat maps for the three reasoners are shown in Figure 4.5. The ROC curves, PR curves, and confusion matrices for each reasoner are shown in Figure 4.6.

As expected, the unfrozen reasoner achieves strictly better classification performance than the reasoners with frozen heads, as the values of all metrics are higher than for other reasoners. In particular, the unfrozen reasoner achieves near perfect AUC ROC and accuracy scores, and very good F1-score, precision and recall scores. The ROC and PR curves for the unfrozen reasoner are nearly perfect, which we also expected. The ontology classification heat map is close to the expected one, but the unfrozen reasoner has trouble classifying trivial axioms $C \sqsubseteq D$, where $C = D$. Perhaps reasoners would classify trivial axioms better if $NN_{\sqsubseteq}$ had more hidden layers, or the layers were wider. We discuss this and other opportunities for improvement in section 5.1.

The reasoner with the frozen pre-trained head is the second-best reasoner after the unfrozen reasoner. It achieves very high AUC ROC, F1-score, accuracy, and precision, although recall could be improved. The ROC and PR curves for the frozen pre-trained reasoner head also indicate a good fit to the data set. The ontology classification heat map for this reasoner sheds some light on the limitations of knowledge transfer in this learning context. Because we chose a small embedding dimension $N_e = 10$, relative to the number of concepts $|N_C| = 99$, if the reasoner head is frozen, then the rather sparse ontology classification map cannot be learned perfectly. For the unfrozen reasoner, learning the ontology classification was easy, because the weights of the deep neural network $NN_{\sqsubseteq}$ could be adjusted, which compensated for the small embedding dimension. For the reasoner with a frozen head, the only option is to adjust the concept embedding vectors and role constructor networks $NN_{\mathcal{K},R_i}$. Given the high classification accuracy and a good ontology classification heat map, we think that the restricted reasoner in experiment 1 learned good concept complement and intersection constructors $NN_{\neg}$ and $NN_{\sqcap}$. However, those constructors are limited, so we interpret the recurring vertical and horizontal lines on the heat map, as attempts to place a concept name embedding closer to the embedding of $\top$ and $\bot$, respectively. This may be done because some concepts like `Food` are general, so for classification it may be useful to just equate it with $\top$, if a better embedding cannot be learned. We also suspect that for reasoners with frozen heads, the ontology classification may be easier to learn in ontologies that have a larger number of role names $|N_R|$, than for simple taxonomies, because the embedding layer has more parameters, allowing the reasoner to achieve a better fit to the training data.

The reasoner with the randomly initialized frozen head was the worst of the three reasoners,

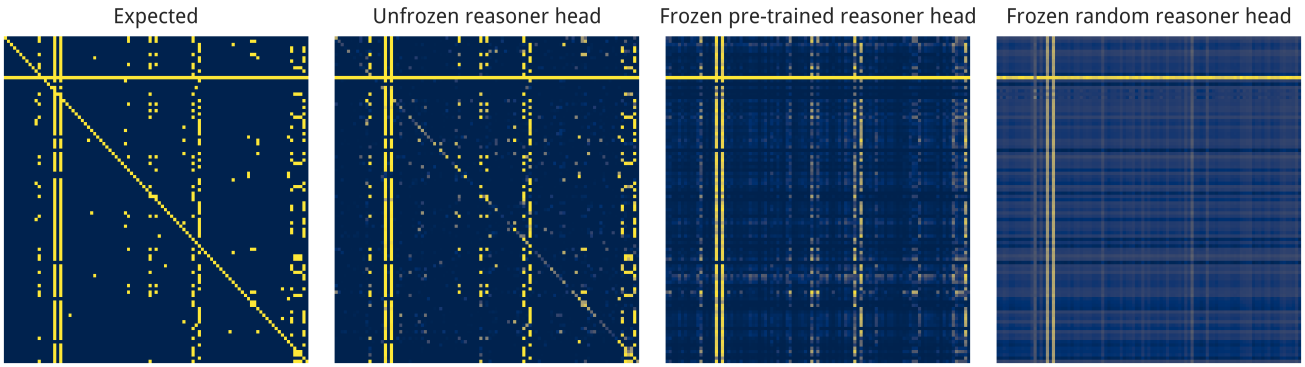

FIGURE 4.5: Expected ontology classification and classification heat maps obtained with the reasoners in experiment 3. The expected classification is the same as in Figure 4.3, and is the ideal result. We do not show the concept names, because they are not necessary for comparing heat maps. A concept that is subsumed by every concept (a concept equivalent to $\bot$) is shown as a yellow horizontal line, and a concept that subsumes every concept (a concept equivalent to $\top$) is shown as a yellow vertical line.

although it is better than a random guesser, with AUC ROC of around 0.78. This reasoner has very high recall, but relatively low precision. Looking at the confusion matrix, one can see that the random head has a much higher number of false positives, which explains the low precision score. The number of false negatives is similar to frozen reasoner head (trained), so the main difference between them is that the randomly initialized reasoner head has much lower precision. By looking at the ontology classification heat map we can see part of the reason for the low precision – it is composed almost entirely of uniform vertical and horizontal lines. This shows that most named concepts are classified similarly to the top and bottom concept.

For all reasoners there are a bit more false negatives than false positives, which we attribute to the slight class imbalance in the training data set. For the randomly initialized reasoner, we do not attribute the high number of false positives to the class imbalance, as class $y = 1$ is the minority one, which would typically result in a high number of false negatives instead. However, the number of false negatives of frozen random reasoner head is similar to the number of false negatives of frozen reasoner head (trained). We do not think that addressing the class imbalance would significantly improve classification results, as those are already very good for reasoners other than the one with the random head.

4.3.4 Evaluation of knowledge transfer

In the last section we discussed the differences between the three reasoners that we trained in experiment 3. Taking into account the differences between the reasoners with frozen heads, we conclude that the transfer of knowledge from the randomly generated KBs in experiment 1 to the pizza ontology was a success. The reasoner with the pre-trained head achieved similar results to the unfrozen reasoner, which is very promising, given that the pizza ontology is certainly different from the randomly generated KBs used as the training data set in experiment 1.

It should be mentioned that the total training time of the reasoners with frozen heads is approximately 17% shorter than the training time of the unfrozen reasoner. The training time is shorter because there is no time spent on updating the reasoner head weights. The differences may be small at this scale, but for reasoner heads with many more parameters, transfer learning may save a lot time.

4.3.5 Embedding analysis

The last part of this experiment is to assess the UMAP visualizations of the embeddings learned by the three reasoners. The UMAP visualizations of the learned embeddings are shown in Figure 4.7.

We set the UMAP parameters in the same way as in experiment 2. We kept the `n_neighbors` set to 50, to emphasize global structure, and set `min_dist` by starting at a large value and decreasing it until clusters started to separate and the visualization became readable. The `min_dist` parameter is set to 0.2, 0.6, and 0.2 for the visualizations for the unfrozen reasoner, the reasoner with the frozen pre-trained head, and the reasoner with the frozen random head, respectively. The larger `min_dist` value was needed for the pre-trained reasoner head, because the cluster of pizzas and the cluster of toppings were so tightly-packed, that the visualization was not readable.

The visualization of embeddings learned by the unfrozen reasoner is very similar to the visualization in experiment 2. The data set in this experiment contained complex axioms with concept expressions using constructors, so the resulting embeddings should capture the ontology semantics better than the embeddings learned in experiment 2. However, we admit that we do not see meaningful differences between this visualization and the one in the previous experiment.

The embeddings learned by the reasoner with the pre-trained head look a bit worse than those of the unfrozen reasoner. The pizzas and the toppings are separated, but inside of the topping cluster the embeddings are not as well organized. For example, some meat toppings are close to vegetarian toppings, and neither meat, nor seafood topping form their own cluster. The cheese toppings are also scattered throughout the topping cluster. Inside the pizza cluster, the cheesy pizzas and vegetarian pizzas are close to each other, which is good. The named vegetarian pizzas are a bit too close to the non-vegetarian pizzas, but this may be because all of these pizzas are cheesy pizzas (except `FruttiDiMare`). The levels of spiciness form a little cluster, which is also good. The top concept is close to `Food` and `DomainThing`, which is expected, but it is also close to `CheesyVegetableTopping`, which should be close to the bottom concept instead. We also think that `HerbSpiceTopping` is too close to the bottom concept.

The embeddings learned by the reasoner with the randomly initialized head look very bad when visualized. Very few concept names form clusters that make sense, and most look like they are randomly scattered. In the upper left corner, a couple of cheesy vegetarian pizzas form a cluster. In addition, general concepts are also close to the top concept, which is good.

In general, we think that the embeddings for the unfrozen reasoner are the best, the embeddings for the reasoner with the frozen pre-trained head are good, and the embeddings for the reasoner with the frozen randomly initialized head are not good at all. Again, the results of the visual assessment of the learned embeddings are consistent with the classification metrics of the reasoners.

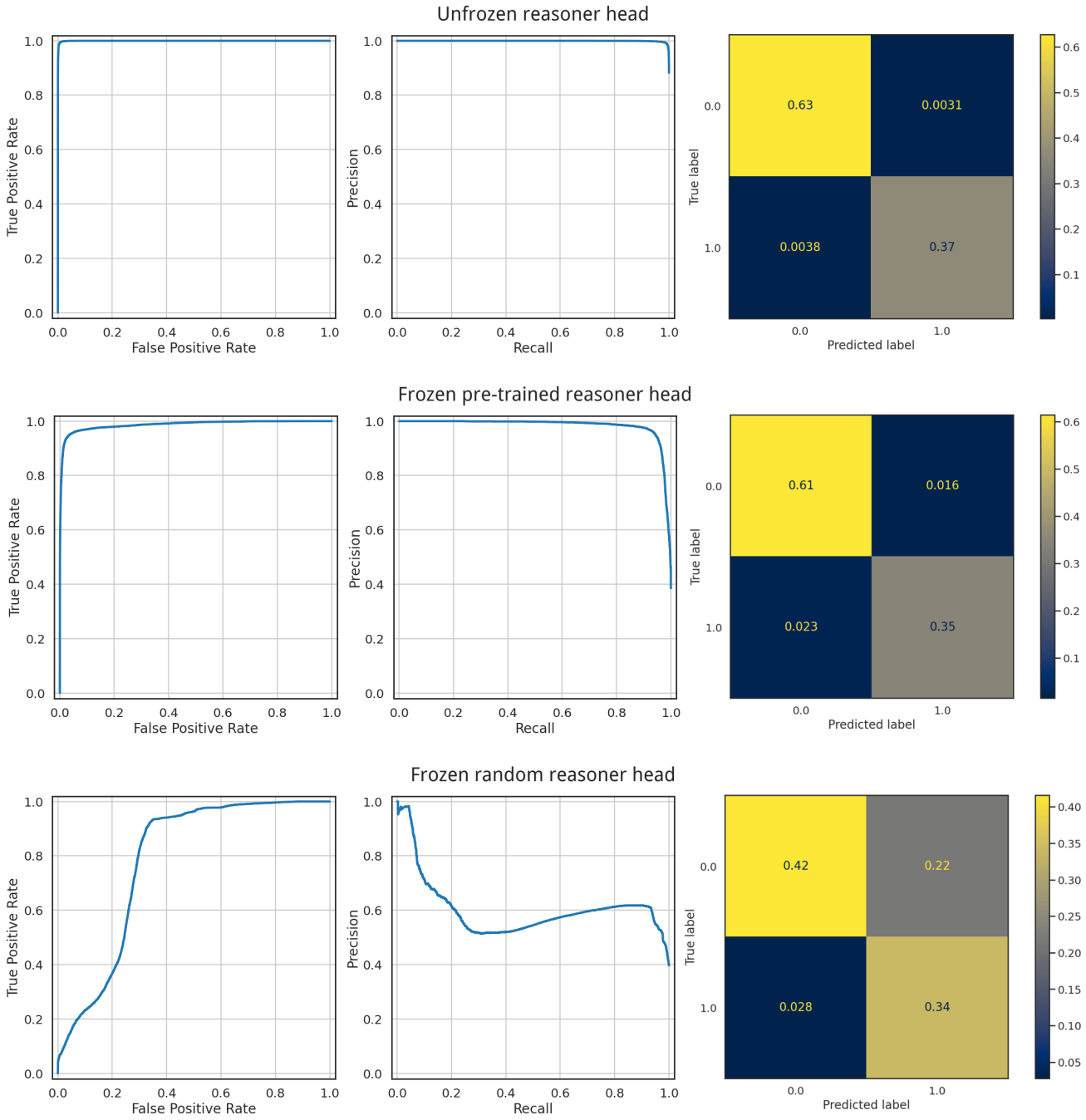

FIGURE 4.6: ROC curves, PR curves, and confusion matrices of reasoners in experiment 3. Confusion matrices are normalized by the number of all samples in the training data set.

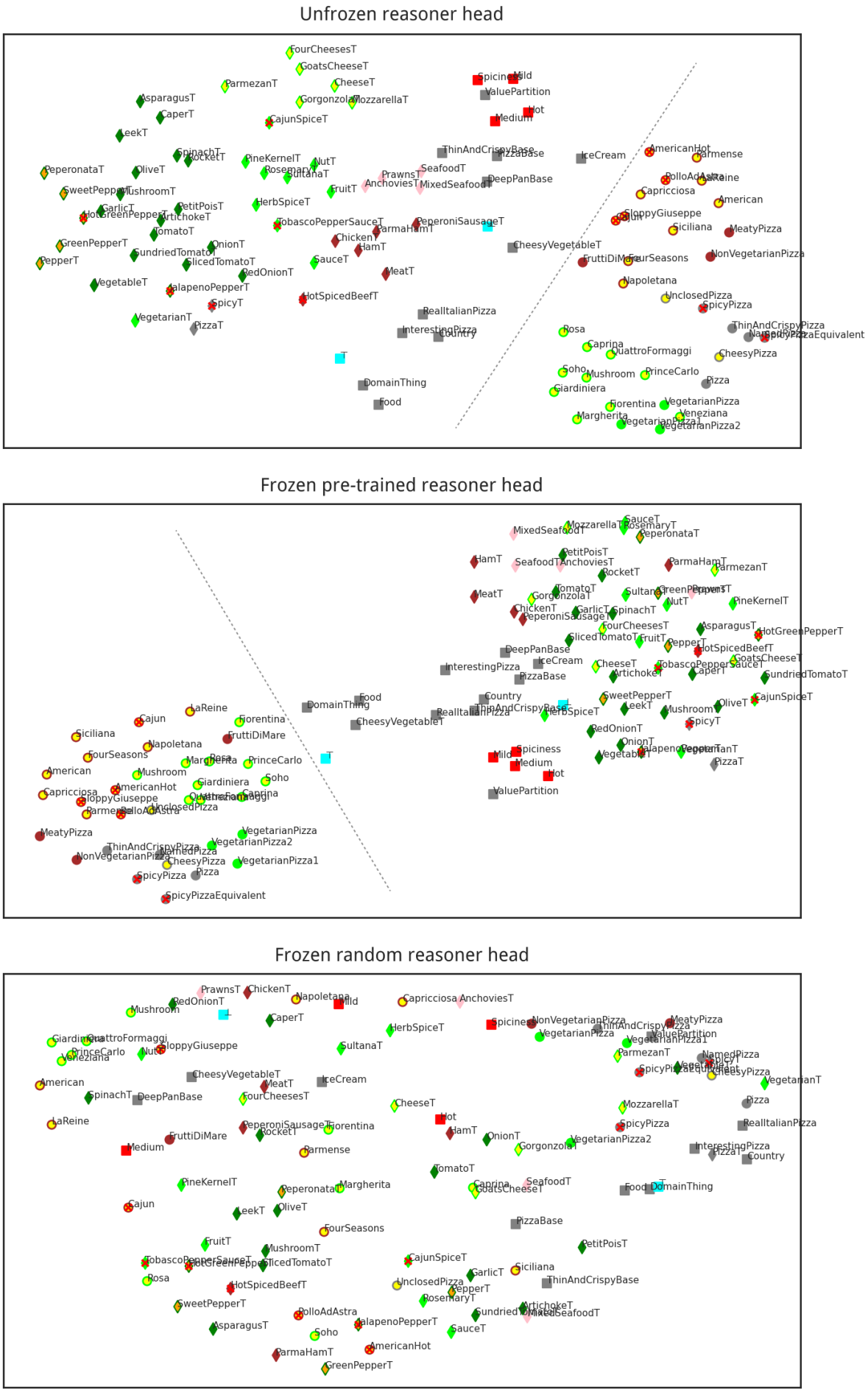

FIGURE 4.7: UMAP visualizations of embeddings learned by reasoners in experiment 3. For emphasis, we added a dashed line separating pizzas and other concept names in visualizations where the two are clearly separated.

Chapter 5

Conclusions

In our work we introduced a novel method of learning data-driven concept embeddings in $\mathcal{ALC}$ knowledge bases. To our best knowledge, our method of learning concept embeddings is the first one using an entailment classifier based on deep neural networks. Thanks to of our unique approach, after learning embeddings, one can use the resulting classifier to perform fast approximate reasoning.

We also show that using recursive neural networks for constructing embeddings of arbitrarily complex concepts obviates the need for manually designing concept vectorization schemes, and avoids the pitfalls of recurrent neural networks operating on a textual representations of concepts. Instead, concept embeddings can be learned in a data-driven way, by simply asking entailment queries for a given knowledge base.

Finally, we show that a significant part of our reasoner is transferable across knowledge bases in the $\mathcal{ALC}$ description logic, including small real-world knowledge bases like the pizza ontology. The advantage of a transferable reasoner is that learning concept embeddings takes less time, and thus is less expensive, when using a pre-trained reasoner head, compared to training an entire reasoner from scratch. Promising initial results in the transferability of deep neural reasoners suggest that it is indeed feasible to embed concepts from multiple domains in a single shared embedding space.

We hope that our deep neural reasoner architecture will allow for greater use of knowledge in models based on neural networks, both by providing an effective way of learning concept embeddings, and learning an accurate entailment classifier for knowledge bases in description logics. Thus, making a small step towards the integration of the neural and symbolic paradigms in artificial intelligence.

5.1 Further work

We identified many opportunities to improve, extend and apply our deep neural reasoner, that were out-of-scope for this work, but look like promising avenues for further research.

In our work we used small neural networks, but deeper and wider concept constructor networks, and subsumption entailment classifier networks could be examined. The number of parameters in the reasoner could also be reduced, while preserving the quality of embeddings and accuracy of entailment classification. Currently, the number of parameters scales quadratically with the embedding dimension, because the reasoner uses the outer product of embeddings as an input to neural networks $NN_{\sqsubseteq}$ and $NN_{\sqcap}$. Instead of passing the entire interaction map as input, convolutional neural networks could be used instead, which could greatly reduce the number of parameters. The number of parameters used by existential restriction constructor networks could possibly be

reduced by representing weights as sparse matrices.

Examining reasoner variants for other description logics like $\mathcal{EL}$, or extending our reasoner to support ABox reasoning might be beneficial. It would also be worthwhile to examine how the reasoner performance scales with the embedding dimension N_e, and how it performs for real-world knowledge bases of different sizes, for example SNOMED CT. After checking if deep neural reasoners can work with large real-world KBs, one could see if using learned embeddings as additional model inputs in other machine learning tasks can improve results.

Finally, it would be interesting to see if recursive neural networks could be applied in reverse to how we use them – to generate concepts, given learned concept embeddings. That would make it possible to not only learn concept embeddings by classifying entailment, but also to induce new concepts by sampling the embedding space. Among other uses, concept induction can be used for explainable AI, but current algorithms are not scalable [41]. If it were possible to traverse the space of concept embeddings, with the gradient descent algorithm for example, then applying concept induction for explainable AI could be made feasible. Of course this is just a vague idea of what may be possible with a bidirectional mapping between concept expressions and the concept embedding space.

5.2 Acknowledgments

We would like to thank out supervisor, dr inż. Jędrzej Potoniec, for the idea of using a transferable classifier as a means for learning concept embeddings for knowledge bases in description logics, and their support and incredibly helpful discussions. We would also like to thank dr hab. inż. Agnieszka Ławrynowicz, our mentor in the “AI Tech” project, for guidance and support in our studies, and helpful feedback on our work.

Bibliography

[1] Franz Baader and Werner Nutt. Basic description logics. In *The description logic handbook: theory, implementation, and applications*, pages 43–95. Cambridge University Press, USA, January 2003.

[2] OWL Web Ontology Language Overview, 2004.

[3] Pascal Hitzler, Markus Krotzsch, and Sebastian Rudolph. *Foundations of Semantic Web Technologies*. Chapman and Hall/CRC, 0 edition, August 2009.

[4] Tim Benson. *Principles of Health Interoperability HL7 and SNOMED*. Health Informatics. Springer London, London, 2010.

[5] Sebastian Rudolph. Foundations of Description Logics. In Axel Polleres, Claudia d'Amato, Marcelo Arenas, Siegfried Handschuh, Paula Kroner, Sascha Ossowski, and Peter Patel-Schneider, editors, *Reasoning Web. Semantic Technologies for the Web of Data*, volume 6848, pages 76–136. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[6] Franz Baader, Ian Horrocks, Carsten Lutz, and Uli Sattler. *An Introduction to Description Logic*. Cambridge University Press, Cambridge, 2017.

[7] Rob Shearer, B. Motik, and I. Horrocks. HermiT: A Highly-Efficient OWL Reasoner. In *OWLED*, 2008.

[8] Evren Sirin, Bijan Parsia, Bernardo Cuenca Grau, Aditya Kalyanpur, and Yarden Katz. Pellet: A practical OWL-DL reasoner. *Journal of Web Semantics*, 5(2):51–53, June 2007.

[9] Dmitry Tsarkov and Ian Horrocks. FaCT++ Description Logic Reasoner: System Description. In Ulrich Furbach and Natarajan Shankar, editors, *Automated Reasoning*, Lecture Notes in Computer Science, pages 292–297, Berlin, Heidelberg, 2006. Springer.

[10] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, J. Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, T. Henighan, Rewon Child, A. Ramesh, Daniel M. Ziegler, Jeff Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language Models are Few-Shot Learners. *NeurIPS*, 2020.

[11] A. Ramesh, Prafulla Dhariwal, Alex Nichol, Casey Chu, and Mark Chen. Hierarchical Text-Conditional Image Generation with CLIP Latents. *ArXiv*, 2022.

[12] Monireh Ebrahimi, Aaron Eberhart, Federico Bianchi, and Pascal Hitzler. Towards bridging the neuro-symbolic gap: deep deductive reasoners. *Applied Intelligence*, 51(9):6326–6348, September 2021.

[13] Geoffrey G. Towell and Jude W. Shavlik. Knowledge-based artificial neural networks. *Artificial Intelligence*, 70(1-2):119–165, October 1994.

[14] Jiaoyan Chen, Yuan He, Ernesto Jimenez-Ruiz, Hang Dong, and Ian Horrocks. Contextual Semantic Embeddings for Ontology Subsumption Prediction. *arXiv:2202.09791 [cs]*, February 2022. arXiv: 2202.09791.

[15] Antoine Bordes, Nicolas Usunier, Alberto Garcia-Durán, Jason Weston, and Oksana Yakhnenko. Translating embeddings for modeling multi-relational data. In *Proceedings of the 26th International Conference on Neural Information Processing Systems - Volume 2*, NIPS'13, pages 2787–2795, Red Hook, NY, USA, December 2013. Curran Associates Inc.

[16] Bishan Yang, Wen-tau Yih, Xiaodong He, Jianfeng Gao, and L. Deng. Embedding Entities and Relations for Learning and Inference in Knowledge Bases. *ICLR*, 2015.

[17] Aaron Eberhart, Monireh Ebrahimi, Lu Zhou, C. Shimizu, and P. Hitzler. Completion Reasoning Emulation for the Description Logic EL+. *AAAI Spring Symposium: Combining Machine Learning with Knowledge Engineering*, 2020.

[18] Samy Badreddine, Artur d'Avila Garcez, Luciano Serafini, and Michael Spranger. Logic Tensor Networks. *arXiv:2012.13635 [cs]*, January 2021. arXiv: 2012.13635.

[19] Federico Bianchi and P. Hitzler. On the Capabilities of Logic Tensor Networks for Deductive Reasoning. *AAAI Spring Symposium: Combining Machine Learning with Knowledge Engineering*, 2019.

[20] Daniel Selsam, Matthew Lamm, Benedikt Bünz, Percy Liang, Leonardo de Moura, and David L. Dill. Learning a SAT Solver from Single-Bit Supervision. *arXiv:1802.03685 [cs]*, March 2019. arXiv: 1802.03685.

[21] Melissa E. O'Neill. Pcg: A family of simple fast space-efficient statistically good algorithms for random number generation. Technical Report HMC-CS-2014-0905, Harvey Mudd College, Claremont, CA, September 2014.

[22] Jedrzej Potoniec. Inductive Learning of OWL 2 Property Chains. *IEEE Access*, 10:25327–25340, 2022.

[23] Jean-Baptiste Lamy. Owlready: Ontology-oriented programming in Python with automatic classification and high level constructs for biomedical ontologies. *Artificial Intelligence in Medicine*, 80:11–28, July 2017.

[24] OWL 2 Web Ontology Language Structural Specification and Functional-Style Syntax (Second Edition), 2012.

[25] Rebecca C. Jackson, James P. Balhoff, Eric Douglass, Nomi L. Harris, Christopher J. Mungall, and James A. Overton. ROBOT: A Tool for Automating Ontology Workflows. *BMC Bioinformatics*, 20(1):407, December 2019.

[26] C. Goller and A. Kuchler. Learning task-dependent distributed representations by backpropagation through structure. In *Proceedings of International Conference on Neural Networks (ICNN'96)*, volume 1, pages 347–352 vol.1, June 1996.

[27] Djork-Arné Clevert, Thomas Unterthiner, and S. Hochreiter. Fast and Accurate Deep Network Learning by Exponential Linear Units (ELUs). *ICLR*, 2016.

[28] Xiangnan He, Xiaoyu Du, Xiang Wang, Feng Tian, Jinhui Tang, and Tat-Seng Chua. Outer Product-based Neural Collaborative Filtering. *arXiv:1808.03912 [cs, stat]*, August 2018. arXiv: 1808.03912.

[29] Michal Bednarek, Piotr Kicki, and Krzysztof Walas. On Robustness of Multi-Modal Fusion—Robotics Perspective. *Electronics*, 9(7):1152, July 2020.

[30] Agnieszka Ławrynowicz, Jedrzej Potoniec, Michał Robaczyk, and Tania Tudorache. Discovery of emerging design patterns in ontologies using tree mining. *Semantic Web*, 9(4):517–544, June 2018.

[31] Aaron Eberhart, Michelle Cheatham, and Pascal Hitzler. Pseudo-Random ALC Syntax Generation. In Aldo Gangemi, Anna Lisa Gentile, Andrea Giovanni Nuzzolese, Sebastian Rudolph, Maria

Maleshkova, Heiko Paulheim, Jeff Z Pan, and Mehwish Alam, editors, *The Semantic Web: ESWC 2018 Satellite Events*, Lecture Notes in Computer Science, pages 19–22, Cham, 2018. Springer International Publishing.

[32] I. Loshchilov and F. Hutter. Decoupled Weight Decay Regularization. In *ICLR*, 2019.

[33] Dominic Masters and Carlo Luschi. Revisiting Small Batch Training for Deep Neural Networks. *arXiv:1804.07612 [cs, stat]*, 2018.

[34] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *AISTATS*, 2010.

[35] Alaa Tharwat. Classification assessment methods. *Applied Computing and Informatics*, 17(1):168–192, January 2021.

[36] Takaya Saito and Marc Rehmsmeier. The Precision-Recall Plot Is More Informative than the ROC Plot When Evaluating Binary Classifiers on Imbalanced Datasets. *PLOS ONE*, 10(3):e0118432, March 2015.

[37] Tom Fawcett. An introduction to ROC analysis. *Pattern Recognition Letters*, 27(8):861–874, June 2006.

[38] Matthew Horridge. A practical guide to building OWL ontologies using protégé 4 and CO-ODE tools edition 1.3. [Online; accessed 2022-06-20] `http://mowl-power.cs.man.ac.uk/protegeowltutorial/resources/ProtegeOWLTutorialP4_v1_3.pdf`.

[39] M. Musen. The protégé project: a look back and a look forward. *SIGAI*, 2015.

[40] Leland McInnes, John Healy, and James Melville. UMAP: Uniform Manifold Approximation and Projection for Dimension Reduction. *arXiv:1802.03426 [cs, stat]*, September 2020. arXiv: 1802.03426.

[41] Md Kamruzzaman Sarker and Pascal Hitzler. Efficient Concept Induction for Description Logics. *Proceedings of the AAAI Conference on Artificial Intelligence*, 33(01):3036–3043, July 2019.

Poznan University of Technology
Faculty of Computing and Telecommunication
Institute of Computing Science

Typeset using LaTeX.