FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Deep Learning für das Semantic Web

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Computational Intelligence

eingereicht von

Patrick Hohenecker
Matrikelnummer 0925365

an der Fakultät für Informatik der Technischen Universität Wien

Betreuer: Univ.-Doz. Dr. Thomas Lukasiewicz

Wien,
29.08.2016

(Unterschrift Verfasser) (Unterschrift Betreuung)

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ +43-1-58801-0 ▪ www.tuwien.ac.at

FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Deep Learning for the Semantic Web

MASTER'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Computational Intelligence

by

Patrick Hohenecker
Registration Number 0925365

to the Faculty of Informatics at the Vienna University of Technology

Advisor: Univ.-Doz. Dr. Thomas Lukasiewicz

Vienna,
August 29, 2016

(Signature of Author) (Signature of Advisor)

Declaration of Academic Honesty

Patrick Hohenecker, Schulzgasse 3/7, 1210 Vienna, Austria

Hereby, I declare that I have composed the presented thesis independently on my own and without any other resources than the ones indicated. All thoughts—including those illustrated as tables and figures—taken directly or indirectly from external sources are properly denoted as such. This work has neither been previously submitted to another authority, nor has it been published yet.

Vienna, August 29, 2016

(Patrick Hohenecker)

Erklärung zur Verfassung der Arbeit

Patrick Hohenecker, Schulzgasse 3/7, 1210 Wien, Österreich

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit—einschließlich Tabellen, Karten und Abbildungen—, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 29.08.2016

(Patrick Hohenecker)

Acknowledgements

First and foremost, I want to express my sincerest gratituted to my advisor, Dr. Thomas Lukasiewicz, for supporting me during the course of this work and for providing a lot of helpful advice. Also, I am grateful that he strongly encouraged me to continue my academic endavors, and I am looking forward to our collaboration during my upcoming PhD studies.

Furthermore, I want to thank my parents Helga and Christian, my sister Claudia, and all of my friends for a lot of encouragement and for getting my mind off my academic ambitions once in a while. Most importantly, however, I want to thank my girlfriend Natalie for her unconditional support and a lot of patience during the last few months.

Thank you & Danke!

Abstract

The *Semantic Web* marks the next cornerstone in the development of the World Wide Web, and constitutes the ultimate goal of enabling machines to access and interpret data published on the Web automatically. This, in turn, allows applications to grasp the meaning inherent in the data without human assistance, and empowers them to reason about it autonomously. In practice, however, the task of reasoning about Semantic Web data comes with a number of issues attached. Among these are typical problems related to databases, like incomplete information or conflicting data, but also new challenges came up, like reasoning at the scale of the World Wide Web or the effective construction of expressive knowledge bases. It turns out, though, that many of the obstacles encountered are actually related to the methods employed for reasoning with classical logic rather than to the nature of the Semantic Web itself, and so people started to investigate *machine learning* as an alternative approach to reason about Semantic Web data.

This thesis follows that very path and investigates the use of *deep learning* in order to tackle the problem of instance checking, a particular kind of reasoning task commonly encountered in real-world applications of the Semantic Web. In doing so, I develop a new kind of deep model, and compare its performance on this task with state-of-the-art approaches based on machine learning. Thereby, I show that the newly introduced model is indeed competitive and in parts even superior to existing state-of-the-art techniques. However, the main contribution of this thesis is a generalization of recursive neural tensor networks to arbitrary relational datasets, with the single restriction that these data involve binary relations only. In contrast to similar approaches, a distinctive property of this model is that it can be applied to relational datasets out-of-the-box, i.e. we do not have to account for the particular structure of a dataset beforehand.

Deep learning is among the hottest topics within the field of machine learning right now, and has—to the best of my knowledge—not been employed in connection with the Semantic Web so far.

Kurzfassung

Das *Semantic Web* markiert den nächsten Meilenstein in der Entwicklung des World Wide Web und verkörpert das Ziel Maschinen effektiven Zugriff auf und automatisierte Interpretation von Daten im Web zu ermöglichen. Damit soll es Anwendung gestattet werden, die den Daten inhärente Bedeutung ohne menschlicher Hilfestellung zu erfassen und autonom Schlussfolgerungen daraus abzuleiten. In der Praxis hat sich jedoch gezeigt, dass derartiges Schlussfolgern von einer Reihe verschiedener Umstände erschwert wird. Darunter finden sich altbekannte Probleme, welche im Zusammenhang mit Datenbanken beinahe alltäglich auftreten, wie etwa unvollständige Daten oder widersprüchliche Informationen. Daneben sehen wir uns allerdings auch mit neuen Herausforderungen konfrontiert, etwa Schlussfolgern im Angesicht enormer Datenmengen entsprechend jenen des World Wide Web oder die effektive Konstruktion ausdrucksstarker Wissensbasen. Viele der angetroffenen Erschwernisse sind jedoch nicht dem Konzept des Semantic Web an sich geschuldet. Tatsächlich ist es so, dass die Wurzel des Problems oftmals jene Technologien sind, die automatisches Schließen auf Basis klassischer Logik realisieren. Als Folge daraus, kommen nun zunehmend auch alternative Ansätze, wie etwa Methoden aus dem Bereich *Machine Learning*, im Kontext des Semantic Web zum Einsatz.

Die vorliegende Arbeit widmet sich ebendiesem Ansatz und untersucht den Einsatz von *Deep Learning* im Zusammenhang mit Instance Checking. Dabei handelt es sich um ein bestimmtes Problem automatisierten Schlussfolgerns von großer praktischer Relevanz. Zur Lösung dieses Problems entwickle ich ein neuartiges Modell und vergleiche dieses mit dem State-of-the-Art jener Methoden, welche Schlussfolgern ebenso unter Verwendung von Machine Learning umsetzen. Dabei wird sich zeigen, dass dieses neue Modell durchaus wettbewerbsfähig ist und den bestehenden State-of-the-Art teilweise sogar übertrifft. Das wichtigste Ergebnis dieser Arbeit ist jedoch eine Verallgemeinerung von Recursive Neural Tensor Networks auf beliebige relationale Datensätze, mit der einzigen Einschränkung, dass jene lediglich binäre Relationen beinhalten dürfen. Diese Verallgemeinerung unterscheidet sich dadurch von ähnlichen Ansätzen, dass es ohne der Notwendigkeit weiterer Anpassungen auskommt und somit unmittelbar auf Datensätze beliebiger Struktur angewendet werden kann.

Deep Learning ist derzeit zweifelsohne eines der wichtigsten Teilgebiete aus dem Bereich Machine Learning und wurde bislang—nach bestem Wissen und Gewissen—noch nicht im Zusammenhang mit dem Semantic Web eingesetzt.

Contents

List of Figures

List of Tables

1

CHAPTER

Introduction

In 1998, Tim Berners-Lee, visionary and inventor of the World Wide Web (WWW), presented his idea of a *Semantic Web* (SW; Berners-Lee et al., 2001) for the very first time, and started a huge hype across many fields of computer science. His idea was to extend the Web in a way such that machines can actually »*understand*« and process data automatically by attaching machine-readable information to the content published on the Web. Almost 20 years later, the dust has settled, and while the WWW has grown to a size far beyond imagination, the idea of the SW could not really assert itself by now.

However, even though the SW has not been accepted by the general public, there is still a vivid research community in the field, and the amount of machine-readable data available on the Web has literally exploded in the last decade. Huge SW knowledge bases, like DBpedia[1] and Yago[2], have evolved, and are freely accessible today—it seems needless to say that this has opened a multitude of new possibilities for computer science and related fields.

It was realized quickly, however, that the task of reasoning about this kind of data comes with a number of issues attached. Among these are typical problems related to databases, e.g. incomplete information or conflicting data, but also a number of new challenges came up, like reasoning at the scale of the WWW or the effective

[1] `http://wiki.dbpedia.org` (last visited on August 29, 2016)

[2] `http://www.mpi-inf.mpg.de/YAGO` (last visited on August 29, 2016)

construction of expressive SW knowledge bases (KBs). As it turns out, some of the obstacles encountered are actually related to the methods employed for reasoning with classical logic rather than to the nature of the SW itself, and so people started to investigate alternative ways to reason about SW data. One idea is to employ methods from the field of *machine learning* (ML).

1.1 Objectives and Contributions

In this thesis, we want to follow that very path and investigate the use of *deep learning* (DL) in order to tackle the problem of *instance checking*, a particular kind of reasoning task commonly encountered in real-world applications of the SW. In doing so, we will develop a deep model that combines two well-known types of neural networks (NNs), and compare its performance on this task with state-of-the-art approaches based on ML.

However, the main contribution of this thesis is a generalization of *recursive neural tensor networks* (RNTNs) to arbitrary relational datasets, with the single restriction that these data involve binary relations only. In contrast to similar approaches, a distinctive property of this model is that it can be applied to relational datasets out-of-the-box, i.e. we do not have to account for the particular structure of the dataset beforehand.

DL is among the hottest topics within the field of ML right now, and has—to the best of my knowledge—not been employed in connection with the SW so far. There was, however, one article published that discusses the use of techniques of representation learning, which is a sub-field of DL, based on formal ontologies (Phan et al., 2015). Anyhow, this article is not concerned with the SW, and does not touch upon any of the methods that we are going to consider in this work either.

Interestingly, our conducted research could actually turn out to be interesting for the advancement of ML itself as well. SW data exhibits a distinctly richer structure than the typical dataset used for ML, and the successful integration of these fields could allow us to provide learning algorithms with much more complex datasets in an easy way.

1.2 Structure of this Thesis

The remainder of this document is structured as follows:

- Chapter 2 provides a treatment of the technical background necessary in order to understand the approach developed in this thesis. It starts with a rather informal introduction to the field of the SW, which presents the basic intuition as well as an overview of the most important technologies, but does not cover any details regarding the formal semantics of the knowledge representation language(s). Subsequently, I introduce some basic notions from the field of ML followed by an overview on NNs and DL.

- In Chapter 3, we will have a closer look at the issues related to reasoning in the SW, and consider how ML can help us to deal with (some of) them. Furthermore, this chapter gives an overview of the state-of-the-art in this part of the field. It ends with a first glimpse at my own approach.
- Chapter 4 is the most important part of this work. It starts with an informal presentation of the intuition underlying my approach, and elaborates on all of the details subsequently. In doing so, we will first define the problem formally and in a more general setting, and show that reasoning in the SW fits the constructed formulation. After this, we develop a deep model to tackle the general problem independently of the particular scenario SW, and discuss some of its characteristics.
- In Chapter 5, I present some implementation details, and report the results of my experiments. These indicate that the newly introduced approach is competitive and, in parts, even superior compared to the current state-of-the-art.
- Finally, Chapter 6 summarizes the conclusions that I drew, and discusses possible paths to continue the conducted research.

2 CHAPTER Background

2.1 The Semantic Web

The term *Semantic Web*—sometimes also referred to as Web 3.0, Web of Data, or Linked Data Web—marks the next cornerstone in the development of the WWW. As we know it today, the Web is a seemingly unlimited source of information of all kinds, and while it is easily accessible for us human beings, it poses quite a challenge for machines. It turns out that there are a number of reasons why it is hard to extract and process data published on the Web automatically. One of them is the Web's heterogeneous structure, since information is published in a variety of different ways, all tailored to serve human beings. This last note leads us right to another issue. Most of the content on the Web is addressed to humans, and thus specified in natural language. However, natural language processing (NLP) is a very difficult task, and understanding written or spoken language on a level with a real person is

Literature on the Semantic Web. There are a number of great books and papers about the SW, yet I would like to draw your attention to Domingue et al. (2011) in particular. This comprehensive piece of work provides a great and in-depth overview on the SW in general, its technologies, and a number of important applications. For further information about description logics, I recommend Baader et al. (2007).

still an open problem. Another point is that every website embraces some specific part of the world, and while it is just a routine matter for human beings to interpret information in a certain context, this is yet another tough challenge for computers.

Undoubtedly, one could come up with a number of other problems, but the root of the matter should be clear. However, this is the right point to get back to the SW, which constitutes the ultimate goal of extending the WWW suitably in order to remedy these very issues. To that end, we follow a simple idea, namely to attach machine-interpretable information to any kind of data published on the Web. These additional details are specified in a knowledge representation language, which is equipped with some sort of formal semantics, and thus enable machines to access and interpret the annotated content efficiently. The important aspect, to note here, is that this allows applications to grasp the meaning inherent in the data *without human assistance*, and empowers them to reason about it *automatically*.

2.1.1 Ontological Knowledge Bases

So, the bottom line of the first few introductory words is that we add information expressed in »*another language*« to everything published on the internet. However, it is not really obvious what this attached data should look like, since it needs to enable computers to actually understand and reason about nothing less than the real world. In this regard, one of the central concepts that the SW is built upon is that of a *(formal) ontology*:

▶ **Definition 2.1 (Ontology, Domingue et al., 2011, p. 22)** *An ontology is a formal, explicit specification of a shared conceptualization.* △

This definition might seem a little odd at first sight, but it really just says that an ontology is a formal description of a common concept, in our case (parts of) the real world. The word »*formal*« emphasizes, once again, that such a description needs to be specified in a knowledge representation language with clearly defined semantics. Therefore, ontologies allow us to model certain scenarios, and describe relations among different things or persons in this context. Intuitively, this means that they provide means to describe »*the way our world works*« in a formal manner, and thus enable computers to draw conclusions about it.

An important thing to note is that an ontology is situated on the meta-level, i.e. it might specify general concepts or relations, but does not contain any facts. However, in the sequel we will mostly talk about a number of facts supported by some formal ontology, and we call such a setting an *ontological KB*.

Now we are ready to get back to the question raised at the beginning of this section, and answer it as follows: *machine-interpretable data in the SW is given as an ontological KB*. Hence, whenever some information is attached to any content on the Web, then it consists of a number of facts as well as details about the ontology that these facts belong to. However, even though the original vision of the SW is a global ontological KB, it is really more like a collection of independent KBs together with a number of cross-references. Nevertheless, there are several great ontologies available for various domains by now, and so it is quite common to reuse and extend these rather than to develop a new one from scratch.

Before we move on to the next section, I would like to clarify a few matters of speech for the remainder of this thesis. If we talk about an ontology, then we always refer to a *formal* ontology, i.e. one specified in some formal knowledge representation language. Furthermore, I use the terms ontological KB, SW-KB, SW data, and SW dataset synonymously.

2.1.2 Data Model

In this section, we make inroads into the core of the matter, and have a look at what is probably the most important concept—not just for this thesis, but in general—underlying the SW: the *data model*.

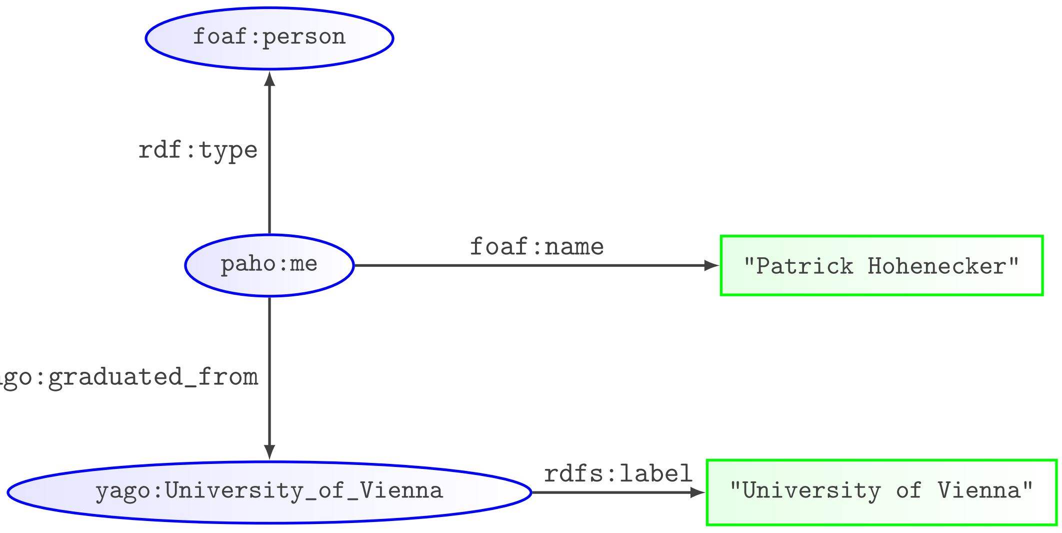
When we think of the usual ways to store large amounts of data, then two approaches pop up in our heads immediately namely relational databases on the one hand and flat data, i.e. plain text files, on the other hand. However, it turns out that the SW adopts neither of these ideas. Instead, data in the SW can—and should—be viewed as a graph, and since all information is specified in the so-called resource description framework (RDF)—we will hear a little more about this in the next section—, people also speak of an RDF graph if they want to address the graph-representation of a SW-KB in particular.

Before we dig a little deeper into this, there is one missing link that we have not touched upon so far. At this point, we know that SW data is given as ontological KBs, yet we have not clarified how to relate these data to things or persons in the real world. However, there is just one reasonable way to achieve this, namely to equip everything, and this means literally everything, with a uniform resource identifier (URI). For the sake of completeness, let me review the definition here:

▶ **Definition 2.2 (URI, Uniform Resource Identifier, Berners-Lee et al., 1998)** *A Uniform Resource Identifier (URI) is a compact string of characters for identifying an abstract or physical resource.* △

So we basically assign a »*primary key*« to everything in the real world, and use it to refer to an entity from within a SW dataset. As the SW is an extension of the WWW, it has become best practice to use URLs for this purpose even though they might not actually refer to a resource on the Web. Therefore, `http://paho.at/me` could, e.g., identify myself as a person, but if you type it into your browser's address bar, then you will receive `HTTP/1.1 404 Not Found`. An important thing to note is that such URIs are not just used to address physical objects, but might refer to immaterial things, like music, laws, or other mental constructs, as well.

Now, after this short digression, we are ready to get back to the already accosted data model. The most straightforward way to explain how data is stored in the SW is by means of an example. Therefore, let's have a look at Figure 2.1, which shows



```
@prefix xsd:  <http://www.w3.org/2001/XMLSchema#> .
@prefix rdf:  <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
@prefix yago: <http://yago-knowledge.org/resource/> .
@prefix paho: <http://paho.at/> .

paho:me  rdf:type  foaf:person .
paho:me  foaf:name  "Patrick Hohenecker"^^xsd:string .
paho:me  yago:graduated_from  yago:University_of_Vienna .
yago:University_of_Vienna  rdfs:label  "University of Vienna"^^xsd:string .
```

Figure 2.1: Two different representations of the same, small sample dataset, just like one might find it on the SW. The graph representation consists of two types of nodes, denoting resources as ovals and literals as boxes, and directed edges which indicate links either between resources or from resources to literals. The text representation on the bottom is called *Turtle format*, and represents exactly the same information as so-called triples, each denoting one edge in the graph.

a tiny sample dataset from two different points of view. The upper half of the figure depicts the data as an RDF graph, and the lower part shows the same information specified in the so-called turtle format (Beckett et al., 2014). Let's disregard the turtle representation for now, and focus on the dataset as graph. If you examine the figure in detail, then you will notice a few things about it:

- The graph is directed, and contains two different kinds of nodes namely ovals on the one hand and boxes on the other one. Ovals denote so-called *resources*, and represent something (or someone) in the real world. Therefore, they are attached with a URI that identifies this very entity. Note that I used the familiar prefix notation, known from XML, in order to keep the URIs short and simple—all of the used prefixes are specified in the turtle code. This way `paho:me` is short for `http://paho.at/me`, and the respective oval represents, once again, myself. Intuitively, one could compare resources in the SW with individuals in the domain of a first-order structure, and in the remainder I use the terms »*resources*« and »*individuals*« interchangeably,
- Boxes denote *literals*, which is just a fancy name to refer to values like character strings, numbers, dates, and so forth. This means that the box labeled with `"University of Vienna"` actually represents the name of a local university as character string.
- The directed edges are called *links*, and denote relationships present in the dataset. Thereby, the type of a certain relationship is—how could it be different—identified by the URI that the according edge is annotated with. By convention, the source node, the type of the link, and the target node are referred to as *subject*, *predicate*, and *object*, respectively. An important thing to note is that literals represent properties of resources, and can thus appear as objects only, i.e. there are no relations with a literal as the subject or between two literals.
- There is one aspect about this example that needs to be emphasized in particular. It turns out that there are resources which represent certain classes or categories of things. Technically speaking, those categories are called RDF classes, and links annotated with `rdf:type` denote the membership of a resource to such an RDF class. In Figure 2.1, e.g., there is one link of this kind which connects `paho:me` to `foaf:person`. This means that the resource `paho:me` belongs to the class `foaf:person`, which denotes all »*things*« that are human beings.

Now this was pretty much everything to say about this simple dataset, and if we wanted to present the same information in natural language, then we would probably come up with a description like this:

> »*There is a person named 'Patrick Hohenecker', who graduated from a university called 'University of Vienna'.*«

In fact, however, this description does *not* directly coincide with what is depicted in Figure 2.1. According to the considered RDF graph, there is no explicit information about `yago:University_of_Vienna` being a university, yet it turns out that this inference is completely justified. The yago ontology specifies that the object of a relation of the type `yago:graduated_from` belongs to the class `wordnet:university`—the prefix `wordnet` references another well-known ontology, but you can safely disregard any details here—, which represents the class of all universities. Hence, a computer could draw the same conclusion as we just did, and infer that `yago:University-_of_Vienna` indeed refers to a university.

You see, an RDF graph is really just a graph representation of explicitly defined information, and does not contain any details that are only inferable from it. This information is usually specified as so-called *triples*, which are 3-tuples of the form (`subject`, `predicate`, `object`). Each triple defines a single edge in a dataset's RDF graph, and one of the most commonly used formats to specify SW data is the already mentioned *turtle format*. For an example, have a last look at the lower part of Figure 2.1.

One last thing, to note, is that the format of the graph in Figure 2.1 is not an arbitrary one. In the context of the SW, graphs are frequently used to depict data, and it has become best practice to use ovals for resources, boxes for literals, and directed edges for links. Therefore, we will stick to this convention for describing all kinds of relational datasets throughout this work.

2.1.3 Core Technologies

Although it is not essential for understanding the sequel, I would like to give you a (very) quick overview on a few important technologies. From a technical point of view, the SW is not a single piece of technology, but rather refers to a whole family of tools and formalisms. At the heart of it are three technical standards, created by the WWW Consortium (W3C):

- **Resource Description Framework (RDF).** This is a set of specifications[1], that determine how to describe information published on the SW—one of these is the turtle format. Therefore, RDF might be considered as an abstract data modelling language, and every piece of data in the SW is represented in it. Note that this means that RDF is not just used for describing facts, but it provides means for specifying ontologies too. Interestingly, ontologies are specified as sets of triples as well, and the relevant, though very basic, classes and links are referred to as RDF schema (RDFS) (Brickley and Guha, 2014). For further details about the RDF, have a look at the official primer by Manola et al. (2014).
- **Web Ontology Language (OWL).** OWL is the knowledge representation language of the SW, and sits on top of the RDFS. It provides advanced tools for modelling problem domains as ontologies, and exhibits different levels of increasing semantic expressiveness. Just like the RDF, OWL is specified as a collection of standards published by the W3C[2], and there is an official primer for it as well: Hitzler et al. (2012).
- **SPARQL Protocol and RDF Query Language (SPARQL).** Last but not least, SPARQL is the query language for the SW. It is reminiscent of SQL,

[1] `https://www.w3.org/standards/techs/rdf` (last visited on August 29, 2016)

[2] `https://www.w3.org/standards/techs/owl` (last visited on August 29, 2016)

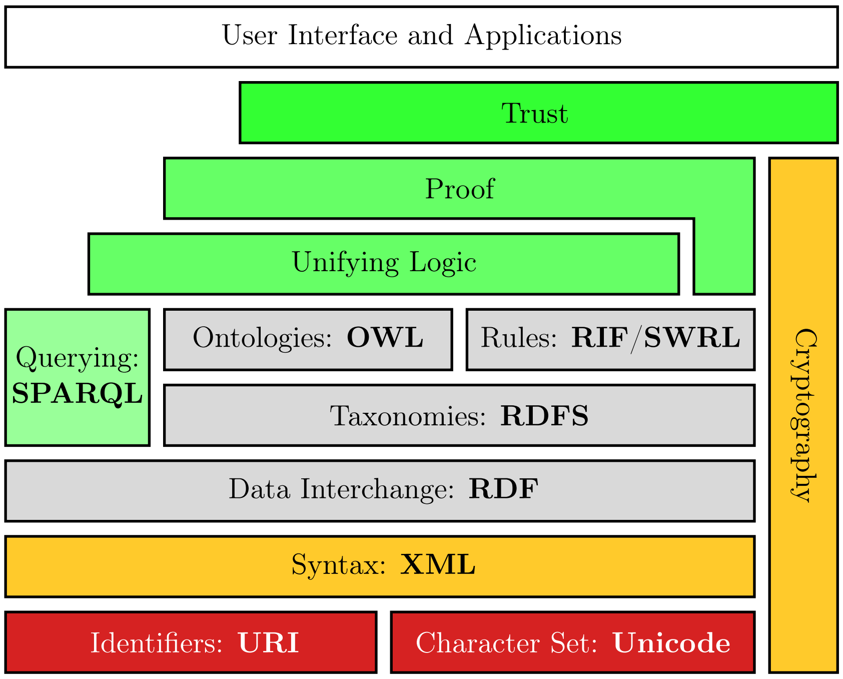


Figure 2.2: This is the SW technology stack, sometimes called »*SW layer cake*«, which illustrates the architecture of the SW. (The depiction follows `https://en.wikipedia.org/wiki/Semantic_Web_Stack`, last visited on August 29, 2016.)

and was designed to meet the specific requirements of the SW, like querying structured data spread across numerous systems.

Figure 2.2 shows the entire technology stack of the SW, and for further information about any of these technologies, I refer the reader to the respective parts of W3C's website[3].

2.1.4 Reasoning Problems

In real-world applications of the SW, there are a number of tasks that we are more or less constantly confronted with, two of which are important for us. Therefore, let's start with defining the, very general, problem of *link prediction*:

▶ **Definition 2.3 (Link Prediction)** *Let $\mathcal{K}$ be a SW-KB and s, p, o three URIs. The problem of assessing whether (s, p, o) holds for $\mathcal{K}$, denoted*

$$\mathcal{K} \models (s, p, o), \tag{2.1}$$

[3] `https://www.w3.org/standards/semanticweb/` (last visited on August 29, 2016)

is called link prediction. △

For the reader with a background in mathematical logic, the use of the entailment relation in this definition, denoted as $\models$, will make perfect sense. However, for understanding the sequel, it is totally sufficient to read Equation 2.1 as »*given knowledge base $\mathcal{K}$, we can infer that (s, p, o) holds.*« This way, we could, e.g., call the KB in Figure 2.1 $\mathcal{K}$, and denote the inference we previously drew about it as

$$\mathcal{K} \models (\texttt{yago:University\_of\_Vienna}, \texttt{rdf:type}, \texttt{wordnet:university}).$$

It is not a coincidence that I employed the symbol for the entailment relation in order to express an inference in this context. In Section 2.1.1 about ontological KBs, we emphasized the use of formal knowledge representation languages in order to define ontologies for the SW. As it turns out, the semantics of OWL[4], the knowledge representation language for the SW, are defined in terms of the description logic SROIQ, which is a fraction of classical first-order logic. Hence, link prediction actually boils down to evaluating whether a certain entailment holds—at least in the original vision of the SW.

With that said, let's reserve the word »*reasoning*« for this kind of inference. This means that whenever we speak about reasoning in the remainder, then this refers to some kind of inference based on mathematical logic.

According to Definition 2.3, link prediction comprises answering any kind of question which can be stated in terms of an RDF triple. Sometimes, however, we are interested in reinforcing the opposite proposition, i.e. whether a certain link has to be absent in a dataset. Often times, even though not in general, this task is considered as link prediction as well, and, employing the terminology used in Definition 2.3, we are going to denote this relation as

$$\mathcal{K} \models \neg(s, p, o).$$

Link prediction is one of the most general problems related with SW-KBs, and subsumes many other tasks that are more specific. Among these is also the following, which will be our main concern in the remainder of this thesis:

▶ **Definition 2.4 (Instance Checking)** *Let $\mathcal{K}$ be a SW-KB, s an arbitrary URI, and c one that refers to an RDF class. Assessing whether the individual identified by s belongs to class c, i.e. whether*

$$\mathcal{K} \models (s, \texttt{rdf}:\texttt{type}, c)$$

holds, is called instance checking. △

This means that instance checking is link prediction confined to those cases where the predicate is `rdf : type` and the object is an RDF class. Therefore, the example

[4] cf. `https://www.w3.org/TR/2012/REC-owl2-direct-semantics-20121211/` (last visited on August 29, 2016)

we gave for link prediction is also one of instance checking. Just like for the general case, one can investigate the question whether an individual does *not* belong to a certain class, and in the following I am going to consider this as instance checking as well.

2.2 Machine Learning

In this section, I review some general notions from the area of ML, and introduce NNs as the basic model underlying all further treatment. Besides that, this part serves as a primer for the notation used throughout this document, as there is unfortunately no consistent terminology for ML and especially not for NNs. Even though I tried to make this work as self-contained as possible, a thorough elaboration of the subsequently presented topics is far beyond the scope of this thesis. Therefore, I will point to numerous references, which provide more comprehensive treatments of the subject matter.

2.2.1 Machine Learning Basics

First and foremost, we need to define what the term *ML* is actually all about. To that end, we have to introduce some terminology first. In what follows, $\mathcal{X}$ and $\mathcal{Y}$ denote two sets called *feature space* and *target space*, respectively. Let me stress at this point that, even though their names might suggest it, neither of them needs to be a vector space. In fact, $\mathcal{X}$ and $\mathcal{Y}$ do not even have to be sets of numbers or vectors, and could just as well consist of symbolic objects. Anyhow, in most scenarios, $\mathcal{X}$ and $\mathcal{Y}$ are indeed sets of vectors, and in this case, we denote the dimensions of their respective elements as p and q.[5] Furthermore, the feature spaces that we consider in this thesis are almost exclusively sets of real vectors, i.e. $\mathcal{X} \subseteq \mathbb{R}^p$, and for the target space, we will mostly even confine ourselves to a two-valued set such that $\mathcal{Y} \subseteq \{-1, 1\}^q$. Another basic notion is the *training set* $\mathcal{T} \subseteq \mathcal{X} \times \mathcal{Y}$, which is usually denoted as $\mathcal{T} = \{(\mathbf{x}^{(1)}, \mathbf{y}^{(1)}), (\mathbf{x}^{(2)}, \mathbf{y}^{(2)}), \dots, (\mathbf{x}^{(m)}, \mathbf{y}^{(m)})\}$ where $\mathbf{x}^{(i)} \in \mathcal{X}$ and $\mathbf{y}^{(i)} \in \mathcal{Y}$ for

Literature on Machine Learning. For the ML novice or people with modest mathematical background, Mitchell (1997) provides a very good introduction to the field. This is one of the classic textbooks on ML, and even though a bit outdated, it is a great point to start from. Another excellent book on ML in general is by Bishop (2006). This text, however, requires basic knowledge of calculus as well as statistics and probability theory. For DL, in particular, I recommend Goodfellow et al. (2016).

[5] As a matter of fact, most ML techniques are defined for numeric inputs only. Therefore, unless stated differently, we always assume that a model works on vectors.

$1 \leq i \leq m$. Note that $\mathcal{T}$ is actually a multiset, i.e. it may contain the same element multiple times, and its cardinality will always be referred to as m.

There are a number of different definitions of the term, but at the bottom line, one could consider ML as some kind of an approximation problem. Anyhow, without making any claim of universal correctness, we use the following, to some degree informal, definition of ML, which is entirely sufficient for this work.

▶ **Definition 2.5 (Machine Learning)** *Let $\mathcal{X}$ and $\mathcal{Y}$ be an arbitrary feature and target space, respectively, and $g : \mathcal{X} \to \mathcal{Y}$ an arbitrary function. Furthermore, let $\mathcal{T}$ be a corresponding training set, i.e. $g(\mathbf{x}^{(i)}) = \mathbf{y}^{(i)}$ for all $(\mathbf{x}^{(i)}, \mathbf{y}^{(i)}) \in \mathcal{T}$. The task of finding a function $h : \mathcal{X} \to \mathcal{Y}$ that resembles g as closely as possible based on $\mathcal{T}$ is called ML.* △

Intuitively, this definition states that there is an unknown function g that associates every element in $\mathcal{X}$ with an according element in $\mathcal{Y}$—not necessarily one-to-one—, and our goal is to find a function h that recreates g at best. However, we only have a limited number of examples that illustrate this mapping—the training set $\mathcal{T}$—, and we want to use these data to figure out »*how g works*« in order to come up with a reasonable choice of h. In the ML literature, this process is called *training a model*.

An important thing to note right here is that the function g is not necessarily deterministic. This means that the outcome of g might be defined in terms of some unknown probability distribution, and in this case we are looking for a function $h(\mathbf{x}) = \mathbb{E}[g(\mathbf{y})|\mathbf{x}]$. Besides that, it is often useful to assume this probabilistic point of view to account for noise in the data. Furthermore, it is, of course, infeasible to consider every possible function in order to determine h. Therefore, the general approach is to confine the search to some set of functions $\mathcal{H}$ right away, and try to optimize as far as possible within this set only. This selection of functions in $\mathcal{H}$ happens implicitly by picking a certain ML model to be trained on $\mathcal{T}$. In this context, $\mathcal{H}$ is sometimes referred to as *hypothesis space*.

Notice further that Definition 2.5 is, at least from a mathematical point of view, clearly not precise, as it simply states that h is supposed the resemble g »*as closely as possible*«. However, there are two reasons why I chose this very wording. First, it turns out that it is not easy to define ML in general, since there are a lot of aspects to be taken into account. And second, the needs of practice often necessitate us to adapt our approach to certain peculiarities, which is why people make use of different notions of »*closeness*« in different situations.

Now let's start with the definition of the problem that we will aim to solve later on.

2.2.2 Classification

In its most general form, the classification problem is defined as follows:

▶ **Definition 2.6 (The Classification Problem)** *Let $\mathcal{C} = \{c_1, c_2, \ldots, c_q\}$ be a finite set of classes, and suppose that every $\mathbf{x} \in \mathcal{X}$ belongs to exactly one of them. The task of associating elements of $\mathcal{X}$ with their corresponding class is referred to as classification.* △

This definition sounds almost frivolous, yet it describes a very import kind of learning tasks. From a ML perspective, one would usually approach this problem as follows. As implied by the definition already, the elements that we want to classify form our feature space $\mathcal{X}$. For the target space $\mathcal{Y}$, however, we do not just label or number the classes in $\mathcal{C}$, but instead encode them in the so-called *one-hot-vector representation*. Therefore, every $\mathbf{y} \in \mathcal{Y}$ is a binary vector, and exhibits one entry for each class in C, i.e. $\mathcal{Y} \subseteq \mathbb{F}_2^q$. Out of these entries, exactly one has a value of 1—the entry for the class referred to by $\mathbf{y}$—while all of the others are 0.

In the course of this thesis, we will often encounter situations where we try to answer a certain yes-no question about individuals in a dataset. In doing so, we treat the considered task as a classification problem with two classes, a *positive class* identifying data points for which the answer is yes and a *negative class* for the opposing case. Such a setting is called a *binary classification problem*, and we usually simplify things a little by defining $\mathcal{Y} = \{-1, 1\}$ with the understanding that for any $(\mathbf{x}, y) \in \mathcal{X} \times \mathcal{Y}$ the input $\mathbf{x}$ belongs to the yes-class whenever $y = 1$ and to the no-class if $y = -1$.[6]

If we are facing a certain classification problem, then, according to Definition 2.6, we employ some ML model to learn a function $h : \mathcal{X} \to \mathcal{Y}$ based on the data given as training set $\mathcal{T}$. This function h is called a *discriminant function*, as it maps elements from the feature space right to the target space, and thus »*discriminates*« them by class. It should be clear, however, that there is hardly a scenario where we are actually able to classify elements from $\mathcal{X}$ perfectly well—at least not if we are dealing with a real-world problem—, and so we usually spend our time looking for a function h that accomplishes the job as good as possible.

Still, it turns out that it is often more helpful to have the probabilities of the different classes for some element available rather than an actual discriminant function. Suppose, e.g., that we are confronted with a binary classification problem, and for $\mathbf{x} \in \mathcal{X}$ we have $\mathbb{P}\{y = 1 \mid \mathbf{x}\} = 0.51$ and thus $\mathbb{P}\{y = -1 \mid \mathbf{x}\} = 0.49$. Let's assume further that erroneously classifying $\mathbf{x}$ as a positive instance is much more severe than accidentally assigning it to the negative class—$y = 1$ could, e.g., mean that a patient described by $\mathbf{x}$ is healthy while $y = -1$ means she has cancer. So, if we would

[6] In some textbooks, you also find the convention to use $\mathcal{Y} = \{1, 0\}$ as class labels for a binary classification problem, since this makes the mathematical formulation of certain learning algorithms a little easier. However, as far as we are concerned, the particular choice of $\mathcal{Y} = \{-1, 1\}$ will prove to be more convenient later on, and so I stick to it throughout this work.

base our classification decision on these probabilities, then we could assign $\mathbf{x}$ to the negative class even though its probability is slightly lower. If we used a discriminant function, however, then this would probably associate $\mathbf{x}$ with the positive class, a decision that is basically a guess, and which might imply serious consequences.

Models that try to learn conditional probability distributions like the one just encountered, i.e. which induce a function

$$h = \begin{cases} \mathcal{X} & \rightarrow & \mathbb{P}\{\mathcal{Y} \mid \mathcal{X}\} \\ \mathbf{x} & \mapsto & \mathbb{P}\{\mathcal{Y} \mid \mathbf{x}\} \end{cases} ,$$

are called *discriminative models*—not to confuse with the formerly introduced discriminant functions—, and will be of major interest in the subsequent chapters. It turns out that there is a third kind of classification algorithms, so-called *generative models*, but for further details I refer the interested reader to Bishop (2006, Section 1.5.4).

2.2.3 Neural Networks

NNs are a family of models used for ML, and were originally inspired by the way that information is processed in the human brain. They are often depicted as directed graphs (cf. Figure 2.3) where the nodes represent processing units, so-called neurons, that perform a single computation step. In the most common network architecture, the single neurons are stacked up in layers, and perform computations based on the inputs received from those units situated at the previous level. Thereby, neurons within one and the same layer are usually not connected, and thus perform their calculations independently of each other.

However, there are two exceptions from this general pattern. The first layer of a NN is called *input layer*, and does not perform any kind of computation at all. Instead, it represents the input to the network, and thus contains one neuron for each dimension of the feature space $\mathcal{X}$, which simply emits the respective value of an input vector $\mathbf{x} \in \mathcal{X}$. Furthermore, the last layer of a network is called *output layer*, and the computed values of its neurons are not provided as inputs to any other units, but are regarded as the final output of the whole network.

All layers between input and output are referred to as *hidden layers*, since the optimal values for their outputs are not exposed by the training data. NNs vary a lot with respect to both the number of hidden layers they contain as well as the number of neurons within every single one of them. Furthermore, in the ML literature, there is no uniform way of counting the layers in a network. Therefore, we will stick to the convention to count all layers except the input, i.e. according to our terminology the NN depicted in Figure 2.3 is a 2-layer network. This way, the number of layers reflects the sequential computation steps performed by a NN.

Now before we move on to consider this kind of model in detail, I would like to mention a few further remarks about NNs in general. Networks that exhibit an architecture like the one just introduced are referred to as *feed-forward NNs*.

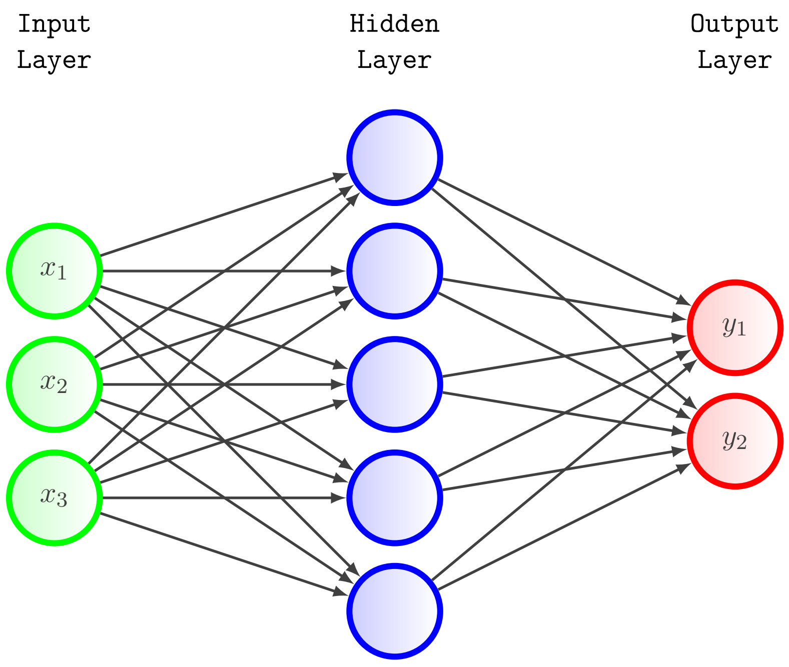


Figure 2.3: This is a depiction of a 2-layer NN that computes a non-linear function h, and maps values from $\mathcal{X} \subseteq \mathbb{R}^3$ to $\mathcal{Y} \subseteq \mathbb{R}^2$. The neurons in the input layer represent the single elements of a data vector for some $\mathbf{x} = (x_1, x_2, x_3)^T \in \mathcal{X}$, and the output layer represents an outcome $h(\mathbf{x}) = \mathbf{y} = (y_1, y_2)^T \in \mathcal{Y}$. Neurons in the hidden layer can be viewed as feature extractors, and perform intermediate computation steps. The directed edges in the diagram depict the dataflow through the NN.

The name stems from the fact that these networks can be depicted as directed acyclic graphs, and thus information »*flows*« through the network in one direction only. In contrast to this, NNs that embody directed cycles are called *recurrent*. Furthermore, when people talk about feed-forward NNs, then they usually refer to the just introduced »*layered*« architecture, even though this is not the only kind of feed-forward model.

Lastly, I want to emphasize that, even though layered feed-forward NNs are the most commonly encountered type in practice, there is a great number of variations of NN models, and the architectures that we present and use in this thesis are really just a few among a number of others. For further details, I would like to refer the interested reader to Goodfellow et al. (2016, Part II).

Computations in a Neural Network

As mentioned already, NNs were inspired by insights from the field of neuroscience. However, the way we usually use them for ML today does not pursue the objective of mimicking the human brain at all, and so it is instructive to view them as what they really are, namely non-linear functions made of simple building blocks. These building blocks are the single neurons that a NN consists of, and in this section, we want to have a look at how they compute a network's final output. Accordingly, the representation of a NN as a graph can be regarded as depiction of the function computed by the network in terms of other, notably very simple, functions.

Let's start by defining some terminology. In the following, we use the letter z to denote the value computed by a single neuron. However, in order to distinguish the outputs of the different neurons in a network, we use superscripts to refer to layers as well as subscripts to identify neurons within a layer. This means that $z_j^{(i)}$, e.g., denotes the output of the j-th neuron in the i-th layer, and sometimes we will denote all of the outputs in a layer more compactly as vector $\mathbf{z}^{(i)} = (z_1^{(i)}, z_2^{(i)}, \dots)^T$. For the input layer we use index zero, and thus $\mathbf{z}^{(0)} = \mathbf{x}$ if a neural network is provided with the input $\mathbf{x} \in \mathcal{X}$.

Using this notation, the rather simple computations performed by a neuron are the following:

$$a = b + \mathbf{w}^T \mathbf{z}^{(i-1)}, \tag{2.2}$$

$$z_j^{(i)} = f(a). \tag{2.3}$$

(Notice that I dropped the indices $^{(i)}$ and $_j$ for a, b, $\mathbf{w}$, and f in order to keep the notation uncluttered.)

Now, there are some things to note about Equation (2.2). The value a is called *activation*, and is simply a linear combination of the inputs received from the neurons of the previous layer plus a *bias term* b. This bias as well as the *weight vector* $\mathbf{w}$ are adjustable parameters of the model. Therefore, all of the learning techniques that are used to train such a model aim to adjust these parameters properly in order to enable a NN to fit a set of training data well. It is important to notice that, at least in the most common architecture, every neuron has its own bias and weight vector, which are independent of any other parameters in the network.

To keep the notation concise, we will write the bias terms of the neurons in layer i jointly as vector $\mathbf{b}^{(i)}$, and integrate the weights of all neurons in that layer into a single *weight matrix* $\mathbf{W}^{(i)}$ such that the weights of the j-th neuron sit at row j, i.e.

$$\mathbf{b}^{(i)} = \begin{bmatrix} b_1^{(i)} \\ b_2^{(i)} \\ \vdots \end{bmatrix} \qquad \text{and} \qquad \mathbf{W}^{(i)} = \begin{bmatrix} \left(\mathbf{w}_1^{(i)}\right)^T \\ \left(\mathbf{w}_2^{(i)}\right)^T \\ \vdots \end{bmatrix}.$$

Using these conventions, we can rewrite Equation 2.3 more compactly as

$$\mathbf{z}^{(i)} = f^{(i)}\left(\mathbf{a}^{(i)}\right) = f^{(i)}\left(\mathbf{b}^{(i)} + \mathbf{W}^{(i)} \mathbf{z}^{(i-1)}\right),$$

where f is applied element-wise, i.e.

$$f^{(i)}\big(\mathbf{a}^{(i)}\big) = \begin{bmatrix} f^{(i)}\big(a_1^{(i)}\big) \\ f^{(i)}\big(a_2^{(i)}\big) \\ \vdots \end{bmatrix}.$$

In the NN terminology, the function f is usually called *activation function* or sometimes just *the nonlinearity*. As it turns out, there are a number of possible activation functions that work well in practice. It is critical, however, that f is actually nonlinear. The reason for this is that a linear combination of linear functions yields, again, a linear function. So, if f was indeed linear, then the same would be true for the whole network, and this, in turn, would clearly limit its capability of mastering complex ML tasks.

In practice, the actually applied nonlinearity strongly depends on the field of application. Furthermore, it is often hardly possible to predict beforehand which of the available options will work best. However, as a default choice for hidden units most practitioners advocate the use of so-called *rectified linear units* (ReLUs). These are neurons that use the activation function

$$f(x) = \max\{0, x\},$$

shown in Figure 2.4a, and exhibit a number of pleasant characteristics (cf. Goodfellow et al., 2016, Section 6.3).

Besides that, one also frequently encounters hidden units that make use of the two nonlinearities that were state-of-the-art before the introduction of ReLUs—in fact, they still are for some applications. These are the *logistic function*

$$\sigma(x) = \frac{1}{1 + e^{-x}},$$

also called *sigmoid function*, on the one hand and the hyperbolic tangent

$$\tanh(x) = \frac{e^{2x} - 1}{e^{2x} + 1}$$

on the other hand. Figure 2.4b and c shows plots of both of them.

For neurons in the output layer, the choice of the activation function is predicated on the particular learning task that a NN is used for. As suggested already, we are mostly interested in discriminative models for binary classification problems, and it turns out that a single output neuron with a sigmoid nonlinearity is exactly what we need. If you examine Figure 2.4b in detail, then you see that the logistic function maps real values to the open interval $(0, 1)$. Hence, we might consider the output of a sigmoid unit as a probability, and indeed, for some input $\mathbf{x} \in \mathcal{X}$ we are going to interpret such a network's one-dimensional output as $\mathbb{P}\{y = 1 \mid \mathbf{x}\}$. Since we are facing a binary classification problem, this implies that $\mathbb{P}\{y = -1 \mid \mathbf{x}\} = 1 - \mathbb{P}\{y = 1 \mid \mathbf{x}\}$, and so we know the entire conditional distribution of the possible classes given the input $\mathbf{x}$, i.e. $\mathbb{P}\{\mathcal{Y} \,|\mathbf{x}\}$. Therefore, we arrived at a discriminative NN model.

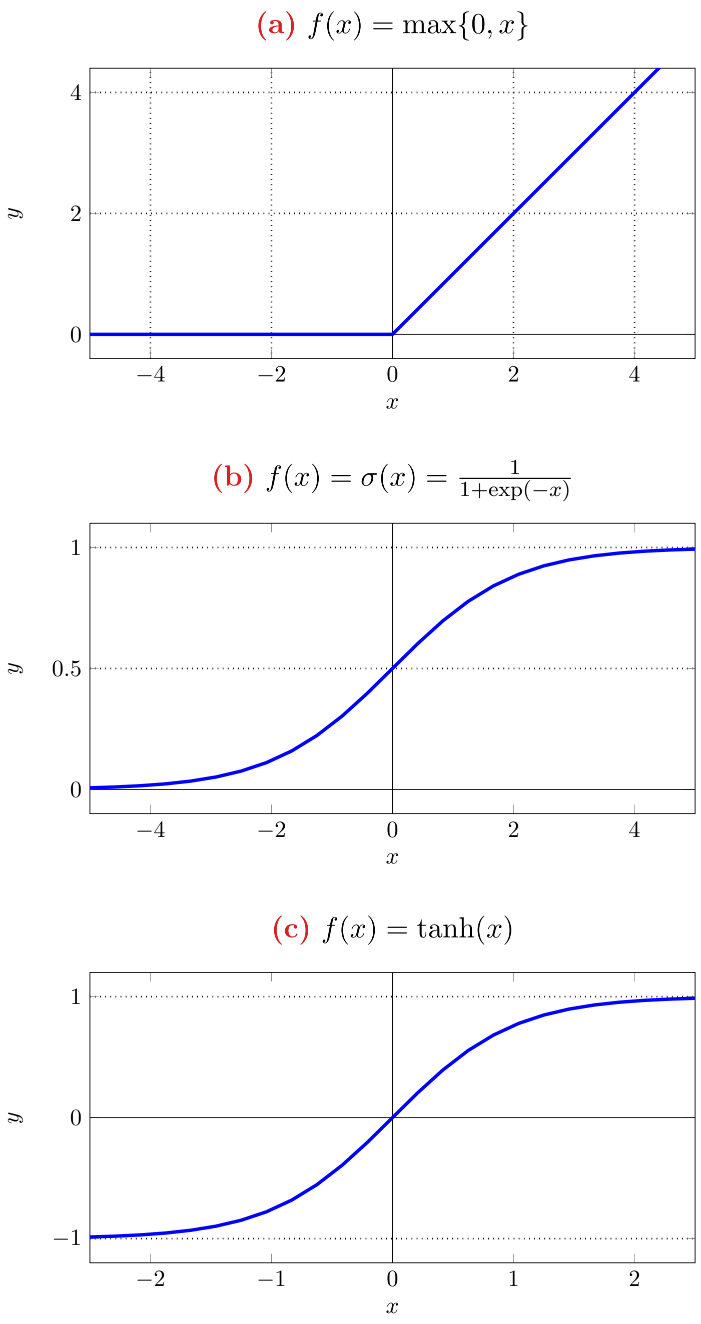


Figure 2.4: The plots of three activation functions f commonly used in feed-forward NNs: **(a)** the activation function of a ReLU, **(b)** the logistic function, and **(c)** the hyperbolic tangent.

Network Training

The most important algorithm for training feed-forward NNs is called *error backpropagation* (Rumelhart et al., 1986; for a nice introduction also cf. Bishop, 2006, sections 5.2 and 5.3), or just *backprop* for short. Although we are going to have a look at the standard version of the algorithm only, there exist a number of extensions of basic backprop, and it has laid the foundation for many more advanced learning strategies. Among these are also algorithms that are used to train other kinds of NNs, e.g. *backpropagation through time* (Werbos, 1990) for recurrent NNs and *backpropagation through structure* (Goller and Küchler, 1996) for recursive NNs.

Like the majority of ML models, backprop follows the principle of *empirical risk minimization* (Vapnik, 1998, Section 1.7). To that end, we define an appropriate error function which provides us with a measure of how good or bad our model works for a given training set $\mathcal{T} = \{(\mathbf{x}^{(1)}, y^{(1)}), (\mathbf{x}^{(2)}, y^{(2)}), \dots, (\mathbf{x}^{(m)}, y^{(m)})\}$. Subsequently, we try to minimize this error as far as possible by adjusting the model's parameters appropriately. A good example of such an error function is the familiar mean-squared error

$$E(\Theta) = \sum_{i=1}^{m} E_i(\Theta) = \frac{1}{2} \sum_{i=1}^{m} \left[y^{(i)} - y\big(\mathbf{x}^{(i)}; \Theta\big) \right]^2 ,$$

where Θ denotes a vector of all parameters in the model and $y(\cdot)$ is the function computed by the considered feed-forward NN. The notation $y(\mathbf{x}^{(i)}; \Theta)$ expresses that y is parameterized by Θ.

In general, the concrete error function depends, once again, on the considered ML task, and for a binary classification problem one usually makes use of the so-called *cross-entropy error function*:

$$E(\Theta) = \sum_{i=1}^{m} E_i(\Theta) = -\sum_{i=1}^{m} \left\{ \frac{y^{(i)} + 1}{2} \ln y\big(\mathbf{x}^{(i)}; \Theta\big) + \frac{1 - y^{(i)}}{2} \ln \left[1 - y\big(\mathbf{x}^{(i)}; \Theta\big) \right] \right\}.$$

In fact, this equation describes the cross-entropy error function *for a single logistic output unit* and the class labels $\{-1, 1\}$, i.e. the error function appropriate for our purposes.

Now, in order to minimize an error function like this, we simply use standard *stochastic gradient descent* (cf. Goodfellow et al., 2016, Section 8.1), usually on minibatches, to adjust the parameters of the model—for a feed-forward NN, these are the weight matrices and the bias terms. This means that we compute the partial derivatives of the error with respect to all of the parameters, and take a step towards the direction of the steepest descent, as indicated by the negations of those derivatives, for each of them. The updates in a single layer can thus be written as

$$\Delta \mathbf{W}^{(i)} = -\eta \frac{\partial E}{\partial \mathbf{W}^{(i)}} \qquad \text{and} \qquad \Delta \mathbf{b}^{(i)} = -\eta \frac{\partial E}{\partial \mathbf{b}^{(i)}},$$

respectively, where η denotes the applied learning rate. However, in calculating these update steps, backprop applies the chain rule for differentiation in a clever way such that we can re-use parts of the derivatives computed for higher layers in computing those for lower ones.

Let's elaborate on this idea by considering the error E_n contributed by the single training instance $(\mathbf{x}^{(n)}, \mathbf{y}^{(n)})$, and we start with evaluating the update for a particular weight $w_{i,j}^{(k)}$. To that end, we observe that by the chain rule for derivatives

$$\frac{\partial E_n}{\partial w_{i,j}^{(k)}} = \frac{\partial E_n}{\partial a_i^{(k)}} \cdot \frac{\partial a_i^{(k)}}{\partial w_{i,j}^{(k)}}.$$

Using the common notation

$$\delta_i^{(k)} = \frac{\partial E_n}{\partial a_i^{(k)}},$$

we can rewrite this equation as

$$\frac{\partial E_n}{\partial w_{i,j}^{(k)}} = \delta_i^{(k)} \frac{\partial a_i^{(k)}}{\partial w_{i,j}^{(k)}}. \tag{2.4}$$

It is easily seen that

$$\frac{\partial a_i^{(k)}}{\partial w_{i,j}^{(k)}} = \begin{cases} x_j^{(n)} & \text{if } k = 1 \text{, i.e. layer } k \text{ is the first layer above the input, and} \\ z_j^{(k-1)} & \text{otherwise,} \end{cases}$$

and so the interesting part in Equation 2.4 is $\delta_i^{(k)}$.

Now suppose that layer k is hidden, i.e. there is at least one more succeeding layer in the network, then it turns out that

$$\delta_i^{(k)} = \big(f_i^{(k)}\big)'\big(a_i^{(k)}\big) \sum_{\ell} \mathbf{W}_{\ell,i}^{(k+1)} \delta_\ell^{(k+1)}.$$

However, this means that we can evaluate $\delta_i^{(k)}$ easily in terms of the deltas of the next layer above, and thus we can calculate the derivatives of all weights in the whole network top-to-bottom in a very efficient way. This is also the reason why the algorithm is called error backpropagation, since we basically »*propagate*« the error derivatives backwards through the network. Using our familiar notation, we can now write the deltas for the whole layer as

$$\boldsymbol{\delta}^{(k)} = \begin{bmatrix} \big(f_1^{(k)}\big)'\big(a_1^{(k)}\big) \\ \big(f_2^{(k)}\big)'\big(a_2^{(k)}\big) \\ \vdots \end{bmatrix} * \Big(\big(\mathbf{W}^{(k+1)}\big)^T \boldsymbol{\delta}^{(k+1)}\Big), \tag{2.5}$$

where $*$ denotes the element-wise product.

If k refers to the output layer rather than to a hidden one, then the concrete form of $\boldsymbol{\delta}^{(k)}$ depends on the particular activation function used in this layer on the one hand as well as on the employed error function on the other hand. However, in the case of a single logistic unit together with the according cross-entropy error, like we are going to use it later on, we get

$$\delta_1^{(k)} = y(\mathbf{x}^{(n)}) - y^{(n)}.$$

As an interesting remark, note that most of the commonly used activation functions have a derivative that can be defined in terms of the function itself, e.g.

$$\sigma'(x) = \sigma(x)\big(1 - \sigma(x)\big) \qquad \text{and} \qquad \tanh'(x) = 1 - \tanh^2(x).$$

This way, Equation 2.5 can be evaluated even faster.

The computation of the derivatives for the biases resembles the steps just presented almost exactly, and the only difference that one has to consider is

$$\frac{\partial a_i^{(k)}}{\partial b_i^{(k)}} = 1.$$

However, I leave it to the reader to figure out the details.

Backprop is a really nice algorithm, still it suffers from a number of flaws. First and foremost, it needs to be mentioned that the previously stated calculations are often not feasible, since the considered error function is—or at least parts of its domain are—simply not differentiable. This can be either due to the fact that (some of) the used activation functions are not differentiable, e.g. if we employ ReLUs, or might be because of the particular choice of an error function itself. It turns out, though, that this issue is not a big problem in practice, and can safely be ignored in many situations. However, in order to account for a non-differentiable error anyway, one can make use of so-called subgradient methods (Ratliff et al., 2006) to calculate directions that resemble the usual gradient.

Furthermore, NNs are highly non-convex functions, and thus it is not trivial to minimize an according error function without getting stuck in some poor local minimum. This is the prime reason why there exist so many extensions of basic backprop, and Goodfellow et al. (2016, Section 8.2) discuss challenges encountered in optimizing NNs more generally.

2.3 Deep Learning

2.3.1 What's Behind All This?

Over the last ten years, the term DL has become a buzzword in the field of computer science, and in a way also embodies the once undreamt-of possibilities that ML provides today. In fact, however, it refers to nothing more than the just introduced NNs, and so the huge hype around DL might seem a little nebulous to some people—at least at first sight. Indeed, the history of the field dates back to the 1940s, and many constituent parts of modern NNs were invented in the 80s or early 90s already. This inevitably leads us to the question of what this recent resurgence of NNs under the designation of DL is actually all about.

The crux of the matter lies in the word »*deep*«, which expresses that DL is concerned with NNs that perform many steps of computation sequentially. To that end, the depth of a model is determined either in terms of the maximal number of sequential steps that are necessary to compute a network's output or the depth of its graphical representation—for layered feed-forward NNs, either way of counting yields, of course, the number of layers. There is no general agreement on this, but many people consider a network with two hidden layers as deep already. In practice, however, we often encounter deep models with about 20 or even more layers in total.

NNs are really powerful tools for ML, and a seemingly simple model like a feed-forward NN with a single hidden layer consisting of a finite number of logistic neurons only is able to approximate *any* continuous function on a compact subset of $\mathbb{R}^n$ already (Cybenko, 1989). By adding additional layers, we can make our models even more general, and so it seems like NNs could be a universal tool for tackling any kind of ML task. However, it turns out that this incredible strength comes with a price to pay. An average deep model today contains tens, or sometimes even hundreds, of thousands of adjustable parameters, and is thus subject to *severe overfitting*. As a result, it is not a trivial task to make a model like this generalize well.

(If you are new to ML, then you can simply picture overfitting as follows: when a student studies for an exam, then she can either learn all of the material by heart or try to actually understand what it is all about. In the latter case, she will be able to answer even yet unseen questions, while she would probably fail miserably on this if she follows the first approach. In the context of ML, the prior is referred to as *overfitting*, and means that a model basically learns the peculiarities of the training data, but misses to figure out the underlying trend. This issue becomes increasingly severe with the growing power of a model, and usually correlates with the number of its adjustable parameters. The ability of computing good predictions for new, i.e. yet unseen, data is called *generalization*.)

Now we are ready to answer the question raised at the beginning of this section, namely why NNs have once again become that popular in recent years. The answer is surprisingly simple: *we learned how to control these models*. In the late 80s, people realized the limitations of deep NNs, and thus many researchers abandoned them in favor of other approaches. However, in 2006, Hinton et al. published a very influential article in which they demonstrated how to pretrain NNs in a certain way in order to achieve better generalization. This paper raised a lot of new interest in the field, and building on it, people have developed a number of other approaches to diminish the issue of overfitting previously encountered with deep NNs. Those techniques are referred to as *regularization*, and we are not going to elaborate on them any further in this thesis. However, the interested reader may have a look at Goodfellow et al. (2016, Chapter 7).

Lastly, it needs to be emphasized that the success of DL is, at least in parts, due to the advancement of computing technologies as well. Besides regularization, another way to remedy overfitting is to increase the amount of data that a model is trained on, in general tremendously. So, the availability of fast computing hardware, especially for general purpose computing on graphical processing units (GPGPU), paved the way for training deep models on massive datasets, and in connection with effective regularization techniques, we are able to apply DL as we know it today.

2.3.2 Recursive Neural Tensor Networks

A particular kind of deep model, that we are going to use later on, is the so-called *recursive neural tensor network* (RNTN; Socher et al., 2013b; Socher et al., 2013a; Socher, 2014). This is an extension of the standard recursive NN (Pollack, 1990), and is used for mapping graphs, or at least data that can be viewed as such, to single vectors. In the following, I am going to explain RNTNs by using the example of binary trees, but the generalization to arbitrary directed acyclic graphs is straightforward.

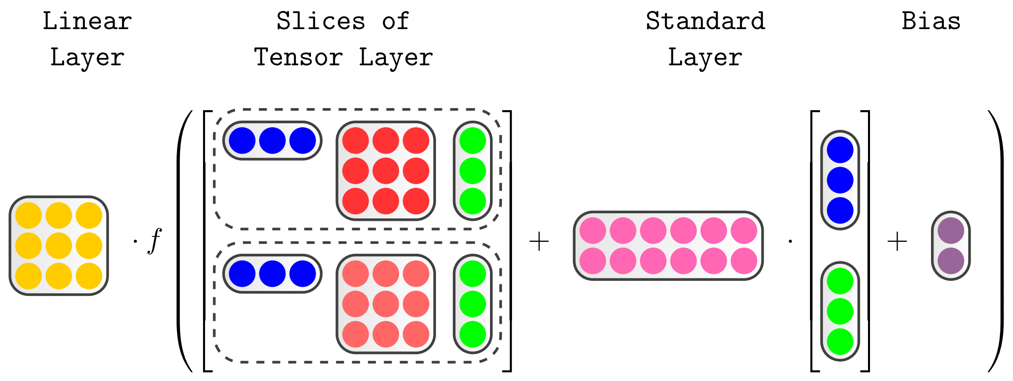
In general, RNTNs are employed for settings like this: suppose that we are facing some ML task for a feature space $\mathcal{X}$ which consists of rooted plane binary trees only, and every leaf of every tree in $\mathcal{X}$ has some vector associated with it. This is the first time in this thesis that we encounter a feature space that is not a subset of some vector space, and the question that we have to answer right here is how to deal with a situation like this. One idea is to generate vectors for entire trees by reducing subtrees to single vectors in a bottom-up fashion. For this purpose, an RNTN makes use of bilinear *tensor layers* which accept two vectors as input and combine them into a single one as output. Notice, however, that, even though they are referred to as layers in the literature, those tensor layers are really more like single neurons restricted to exactly two vector-valued inputs, and the function they compute is the following:

$$g(\mathbf{x}, R, \mathbf{y}) = \mathbf{U}_R f\left(\mathbf{x}^T \mathbf{W}_R^{[1:k]} \mathbf{y} + \mathbf{V}_R \begin{bmatrix} \mathbf{x} \\ \mathbf{y} \end{bmatrix} + \mathbf{b}_R\right) \tag{2.6}$$

for $\mathbf{x}, \mathbf{y} \in \mathbb{R}^n$, $\mathbf{U}_R \in \mathbb{R}^{n \times k}$, $\mathbf{W}_R^{[1:k]} \in \mathbb{R}^{n \times n \times k}$, $\mathbf{V}_R \in \mathbb{R}^{k \times 2n}$, and $\mathbf{b}_R \in \mathbb{R}^k$. (Please note that there exist a few minor variations of this function g. The particular version presented in Equation 2.6 follows Socher et al. 2013a.)

In this equation, $\mathbf{x}$ and $\mathbf{y}$ are the inputs to the tensor layer, and $\mathbf{U}_R$, $\mathbf{W}_R^{[1:k]}$, $\mathbf{V}_R$, and $\mathbf{b}_R$ are adjustable parameters of the model. However, unlike in the case of feed-forward NNs, these are *shared parameters*, which means that all of the layers use the same values and are not parameterized independently. f, as usual, denotes an activation function, most frequently the hyperbolic tangent.

Often times, the inputs to a tensor layer represent objects that are in a certain relation to each other, and we might wish to take this into account for computing their combined vector. For this reason, tensor layers are defined to accept a third input R, which represents this very relation between input $\mathbf{x}$ and $\mathbf{y}$, and, like suggested by



$$\mathbf{U}_R f\left(\mathbf{x}^T \mathbf{W}_R^{[1:2]} \mathbf{y} + \mathbf{V}_R \begin{bmatrix} \mathbf{x} \\ \mathbf{y} \end{bmatrix} + \mathbf{b}_R\right)$$

Figure 2.5: This figure illustrates the computations performed by a single tensor layer of an RNTN for $\mathbf{x}, \mathbf{y} \in \mathbb{R}^3$. The depiction follows Socher et al. (2013a, Figure 2).

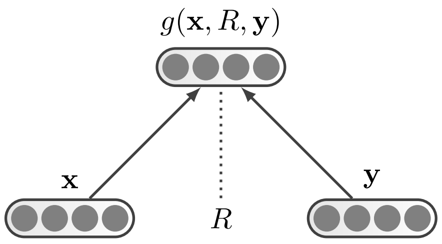


Figure 2.6: A graphical depiction of a single tensor layer.

the subscripts in Equation 2.6, we use a separate set of parameters for every different kind of relation.

The characteristic feature, and at the same time origin of the name, of a tensor layer is the 3-dimensional tensor $\mathbf{W}_R^{[1:k]}$, which is used to account for multiplicative interactions of the inputs. Furthermore, the outermost product $\mathbf{U}_R f(\dots)$ could be considered as an additional linear layer which is squeezed into the function g. Figure 2.5 summarizes all of the computations performed by a tensor layer graphically. Besides that, Figure 2.6 illustrates the way that we usually depict a single tensor layer, and as opposed to the plots of feed-forward NNs, the focus lies on the computed vector representations rather than on the single computation units. Note, however, that we will frequently omit relation R from network diagrams like this in order to emphasize the pure structure.

Now, if we want to compute a vector for an entire tree in $\mathcal{X}$, then we simply resemble its structure in terms of tensor layers as illustrated in Figure 2.7—note that this is certainly possible, since we have vector representations for all of the

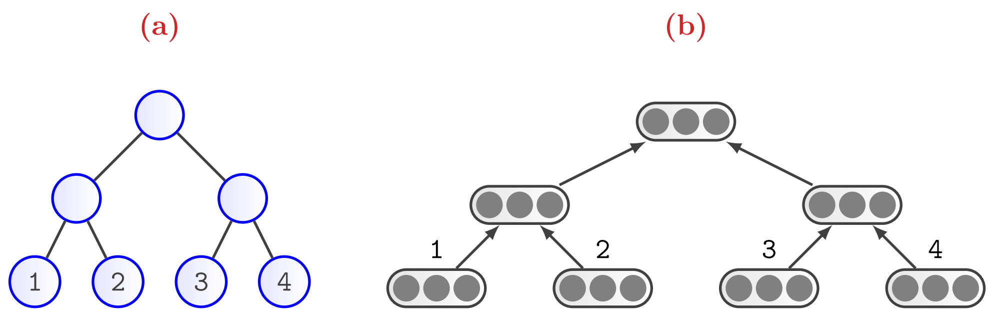


Figure 2.7: Part **(a)** of the figure shows a simple rooted plane binary tree, and **(b)** an according RNTN.

leafs in the tree. However, depending on the particular scenario, we either add one additional step in order to compute the desired prediction based on a tree's vector or consider the computed vector as a prediction already.

At this point, one might come up with the applicable objection that we hardly ever encounter a situation where $\mathcal{X}$ is actually of the particular shape that we just considered. However, RNTNs are usually applied to datasets which contain some kind of structured objects whose constituents possess certain vector representations. Now, in order to facilitate the model, the general strategy is to parse these objects into tree structures first, and provide these as inputs to an RNTN subsequently. Following this pattern, RNTNs have been used successfully for a number of different applications.

There is one last question about RNTNs that needs to be answered, namely how to train these models. In general, recursive NNs are trained with an algorithm called *backpropagation through structure* (Goller and Küchler, 1996), which is a, more or less straightforward, extension of standard backprop. This algorithm, in turn, can be adapted in order to train RNTNs, and the resulting learning technique is called *tensor backprop through structure* (Socher et al., 2013b). Anyhow, I am not going to discuss any details right here, and thus point to the articles just cited as well as Goodfellow et al. (2016, Chapter 10) for further information.

3

CHAPTER

Shortcomings of an Intriguing Vision

In this chapter, I want to shed light on the problem that we aim to solve in this thesis. Therefore, we start by investigating some of the issues attached to SW technologies as we know them today in Section 3.1. Besides that, this section also defines the exact task that we will be concerned with in the remaining parts. Subsequently, Section 3.2 explains why and how people started to employ ML in order to deal with various problems. Section 3.3 then presents the state-of-the-art of machine learning applied in the context of the SW, and Section 3.4 proposes a new approach to tackle the specific problem considered.

3.1 Some Issues of the Semantic Web

As we know from the last chapter, the SW is based on the idea that machine-interpretable data is published as ontological KBs. The formalisms used to describe these ontologies are based on mathematical logic, and thus enable us to employ formal reasoning in order to answer queries about the data. However, it turns out that this logic-based approach comes with a number of problems:

- Although many KBs contain vast magnitudes of data, *incomplete information* is a commonly encountered problem. This raises the question of what to do if we have to make a certain decision, but there is just not enough information to answer our query with formal reasoning. Also, this is a good point to emphasize once again that the SW adheres to the *open-world assumption*, i.e. failure to prove a certain fact true does not allow us to draw the conclusion that the opposite holds.

- Just like every other database maintained by a multitude of people, large SW-KBs often suffer from similar flaws, and one of the most critical is *conflicting information*. This issue actually calls for some non-classical formalisms, still the semantics of the knowledge representation languages for the SW, i.e. the different levels of OWL, are based on description logics. These are mostly fractions of classical first-order logic, and thus allow drawing arbitrary conclusions in the presence of inconsistent KBs—*ex falso quodlibet*.
- For real-world applications, *scalability* is frequently among the major concerns. However, SW data is frequently *distributed over the web*, and thus sometimes more costly to work with. Furthermore, depending on the applied knowledge representation language, *reasoning can be a hard problem*, in a complexity-theoretic sense, and might become infeasible with increasing data stock. Note further that reasoning at the scale of the WWW, which is among the ultimate goals of the SW, is still an open problem.
- Another challenge, frequently encountered in real-world scenarios, is *uncertainty*. In fact, it is almost the normal case that we are not 100% confident about our data, yet this is often disregarded in order to make our lives easier. Just like that, the SW is basically orthogonal to this demand since it is, like mentioned already, rooted in classical logics, and hence based on the assumption of perfect information. Therefore, people have developed numerous extensions of SW technologies which aim to remedy this shortcoming, but none of them has become part of the standardized technology stack yet.
- Last but not least, it needs to be mentioned that the construction of elaborate ontologies, which allow for comprehensive reasoning, is quite a complex task. Accordingly, most real-world ontologies are built with knowledge representation languages of restricted expressivity, and thus allow for limited reasoning only.

Please note that this is not an exhaustive list, yet it illustrates that the original vision of the SW has certain flaws that we need to be aware of. The question that we have to answer at this point is clear: how to deal with (some of) these issues? One possible answer might be by employing *machine learning* rather than formal reasoning.

However, before we move on to dig a little deeper into this idea, let's clarify what we actually want to achieve. In the remaining parts of this document, we will confine ourselves to the problem of *instance checking*, and consider the development of efficient methods to handle it. There are a few reasons why we chose to address this very task. One of them is its practical relevance, as the question whether an individual is part of certain class is an important one in many real-world scenarios.

Furthermore, there are many published papers about different ways to use ML for classifying individuals in a SW knowledge base, which can be used as a reference to assess our own approach.

3.2 Machine Learning to the Rescue

When we say that we want to use ML for instance checking, then what we really mean is the following: given a KB $\mathcal{K}$ and some class $\mathcal{C}$, we aim to induce a function $f_{\mathcal{K}}$ such that

$$f_{\mathcal{K}}(x) = \begin{cases} 1 & \text{if } \mathcal{K} \models (x, \texttt{rdf:type}, \mathcal{C}) \text{ and} \\ -1 & \text{if } \mathcal{K} \models \neg(x, \texttt{rdf:type}, \mathcal{C}). \end{cases}$$

In general, however, we are often times not able to prove any of the entailments on the right-hand side, e.g. due to any of the problems discussed already. Still, $f_{\mathcal{K}}$ needs to provide an answer for any individual x, and is supposed to yield the result that we would obtain if we had perfect information—note that in this case we could establish one of the entailments for sure. Therefore, we want to employ techniques of ML in order to replicate this function as accurately as possible.

Notice that, from a ML point of view, $f_{\mathcal{K}}$ is a discriminant function. However, we will mostly focus on learning discriminative models, i.e. we want to learn the conditional distribution of the class membership. Using basic decision theory, it is, of course, straightforward to mimic $f_{\mathcal{K}}$ based on such a learned distribution.

Now that we discussed the basic idea, it still remains to argue why techniques of ML can actually work in spite of the problems that render formal reasoning impossible. Therefore, let's reconsider the issues just raised in the previous section (in the same order), and examine how this approach might help:

- *Incomplete information.* In the context of ML, this problem is usually referred to as missing values, and there are different ways to handle it. First, there are a number of learning algorithms which are able to handle missing values out-of-the-box, and so we do not need to pay special attention to this issue if we employ any of them. However, a lot of great learning techniques cannot deal with missing values, and so the second option is to make use of some kind of data imputation. Therefore, we replace missing values with reasonable substitutes, and, depending on the chosen strategy, might again make of use of ML in order to fill in these values. Interestingly, even simple imputation techniques, e.g. using the mean value in exchange, turn out to be sufficient in many cases.
- *Conflicting data.* Most techniques of ML are based on ideas from statistics and probability theory, and are not related to any kind of formal semantics. Therefore, they can still provide useful predictions despite possibly conflicting information by exploiting statistical regularities in the data.
- *Scalability.* In general, the application of ML can contribute to the scalability of a system, yet we have to distinguish between so-called *eager* and *lazy* learning

algorithms. Eager learning refers to those techniques, which induce a certain function based on some training set beforehand. This function is then used to compute predictions independently of the original training set. Examples of eager methods are logistic regression, feed-forward NNs, and so forth. Lazy learning methods, in contrast, do not create such a function. As instead, they keep the training data around, and calculate predictions for new data points based on »*close*« samples in the training set. Some examples are k-nearest-neighbour and kernel methods. So, the key difference is that we can discard the training set after training the model for an eager strategy, while we need to keep it available for a lazy one. Accordingly, the computation of predictions with an eager model takes constant time, but depends on the size of the training set if we employ a lazy technique.

- *Uncertainty.* This is a problem that is usually not addressed by off-the-shelf ML algorithms, and in practice it is often just disregarded. Anyhow, it turns out that most techniques can be extended to account for uncertainty in the training data quite straightforwardly. Note, however, that things get more complicated when we have to deal with symbolic rather than numeric data, but depending on the applied model, it might be straightforward to handle this circumstance as well. Another point to consider is that most of the available programming libraries do not care about this particular problem, and so accounting for uncertainty might imply a somewhat higher implementation effort.
- *Complexity of construction.* Let me stress right away that applying some sort of ML on a dataset is not always easy either. However, the most time-consuming part for the ML engineer is usually the preparation of the training data, which is really more of a tedious chore rather than a complex task. So if we are facing high-quality data, then it might be worthwhile to consider training some ML model to answer certain queries, which require the development of a comprehensive and highly expressive ontology otherwise. Another fact, supporting this idea, is that there exist a number of great freely available libraries for numerous programming languages by now, which reduces the programming effort for implementing (most of the available and all of the standard) ML algorithms significantly. Nevertheless, we need to realize that the training procedure might take quite some time, but this strongly depends both on the size of the dataset and the chosen learning model.

3.3 The State-of-the-Art

Although the topic has gained increasing attention over the last few years, the intersection of the SW and ML has been addressed by a manageable amount of papers only. Therefore, it is not quite straightforward to single out a definite state-of-the-art. Nevertheless, people have developed a number of interesting approaches, and we present an overview on the seemingly most promising among these from different areas of ML in this section—of course, we were also guided by what is considered state-of-the-art in related work. For a more comprehensive survey, consider Rettinger et al. (2012). Note, however, that some of the methods presented subsequently were introduced after 2012, and are thus not covered by this article.

3.3.1 Kernel Methods

Kernel methods are an active subfield of ML, and have been applied successfully to a wide range of applications (cf. Shawe-Taylor and Cristianini 2004 for a great treatment of the subject). They are based on the idea that if we choose a suitable feature mapping ϕ, then the training set $\mathcal{T}$ might become linearly separable, and hence we can resort to our state-of-the-art tools for linear models.

One of the reasons why kernel methods are especially appealing is that they allow for a clear separation of the representation of the training data on the one hand and the learning algorithm on the other hand. This is because both the training and new data enter the model via a *kernel function* only:

▶ **Definition 3.1 (Kernel Function, Shawe-Taylor and Cristianini 2004, p. 34)** *A kernel is a function* κ *that for all* $\mathbf{x}, \mathbf{z} \in \mathcal{X}$ *satisfies*

$$\kappa(\mathbf{x}, \mathbf{z}) = \phi(\mathbf{x})^T \, \phi(\mathbf{z}),$$

where ϕ *is a mapping from* $\mathcal{X}$ *to an (inner product) feature space* $\mathcal{F}$. △

In practice, however, kernel functions are hardly ever specified as explicitly as in this definition. As a matter of fact, it is sufficient to ensure that κ is a symmetric, positive semi-definite function, since these coincide with the set of valid kernels. This means that we can actually construct a kernel function without ever touching either ϕ or $\mathcal{F}$, yet we know that both exist and are specified by κ implicitly.

Before we move on, it is instructive to consider the kernel function a little closer. Intuitively, one could view the output of a kernel as a measure of »*closeness*« of two training instances. Note, however, that even though the kernel function needs to yield a numeric value, it might still be defined on non-numeric inputs. This means that the training instances do not necessarily have to be real vectors, and the feature space $\mathcal{X}$ could just as well consist of purely symbolic objects. In this case, κ maps its symbolic arguments to real vectors implicitly, which are used to calculate the numeric outcome eventually.

So, in order to make state-of-the-art kernel methods available for domains other than real vector spaces as well, people have devised numerous kernels for all kinds of symbolic data like text or—and this is especially interesting for us—*graphs*. Another interesting fact to note is that many important non-kernel techniques used for ML have a »*dual*« formulation in terms of some appropriate kernel function. By making use of the just-mentioned kernels, we can make these methods amenable to symbolic data as well.

Now let's stick to this last thought about graph kernels a little longer, since they open up an opportunity to deal with another scenario. Suppose that we are asked to classify instances that are given as vertices of a graph, and we want to employ a kernel method together with some graph kernel. Remember that our data enter the model through the kernel function only, and so it remains to clarify what exactly to feed into κ.

The most common approach is to extract one subgraph for each vertex, and use these as representatives for the vertices themselves. The graphs are usually generated using (some variation of) breadth-first search (BFS) with limited depth, always starting from the vertex for which the subgraph is extracted for. Figure 3.1 illustrates this idea on a simple example.

At this point, it seems like a routine matter to classify individuals in a SW-KB by applying state-of-the-art kernel methods to the corresponding RDF graph. It turns out, however, that there is still some potential for further improvement. Usually, graph kernels follow the idea of measuring the similarity of two graphs by counting certain common substructures. This might make sense for graphs in general, but nodes in an RDF graph are uniquely identified, and so every possible subgraph can occur at most once (Lösch et al., 2012). Therefore, a number of specialized kernels have been developed, which account for some of the peculiarities of a typical RDF graph.

Let's have a look at two kernels which have been applied to SW data successfully next. Both of them were designed for RDF graphs in particular, and we start with the intersection sub-tree (IST) kernel (Lösch et al., 2012). This kernel closely follows the general pattern outlined before:

- Suppose we want to evaluate the IST kernel for two vertices u, v, then we start, as usual, by generating a graph for each of them. Again, we make use of BFS for this purpose, but this time we use the tree-search version rather than the graph-search, which was used in the example in figure 3.1. As a result, the created graphs are trees which might contain repeated vertices, call them t_u and t_v.
- As a next step we compute the intersection of the trees t_u and t_v, and rename all repeated vertices in order to make sure that there is no more than one of u and v. Furthermore, we remove all parts which are not connected to any of the root elements from the created intersection, and call the resulting graph G_{IST}.
- Now we are almost done, since the value of $\kappa(u, v)$ is the number of subtrees of G_{IST}, where each subtree is weighted by a discount factor based on its height. However, Lösch et al. (2012) also considered counting full subtrees only, and their experiments suggest that this version, which they refer to as *full subtree kernel*, works slightly better.

For our own experiments, we employed the full subtree kernel only, and whenever we talk about the IST kernel in the remainder of this thesis, then we refer to the full subtree version.

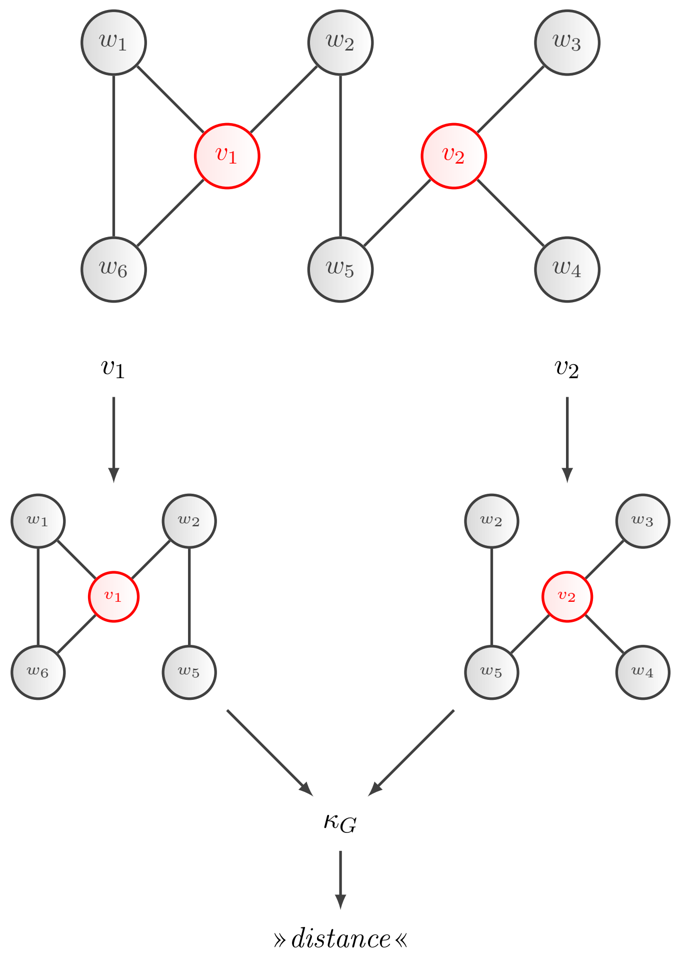


Figure 3.1: This figure illustrates how to define a kernel function κ for vertices in a graph by making use of another kernel function, namely a graph kernel κ_G. The upper part of the figure is a simple example of a graph, and the lower part describes how to calculate $\kappa(v_1, v_2)$ in this setting. Note that BFS with a depth limit of two is used for extracting subgraphs.

Another interesting kernel function is the Weisfeiler-Lehman (WL) kernel for RDF (De Vries, 2013). It is based on a general graph kernel, the WL kernel (Shervashidze et al., 2011), which, in turn, builds on the WL test of graph isomorphism (Weisfeiler and Lehman, 1968). To avoid confusion, let's establish the following convention right away: in the sequel, if we talk about the WL kernel without any further explanation, then we always address the WL kernel for RDF data.

Right here, I am going to outline the basic approach only, since a thorough introduction would lead us a bit too far astray. Anyhow, I want to refer the interested reader to De Vries (2013).

As mentioned before, the kernel is based on the WL test of graph isomorphism. The test proceeds by relabeling the vertices in the two graphs under investigation iteratively. In doing so, every vertex's label in an iteration is updated based on the labels that its neighbors were assigned in the preceding one. This way we keep iterating until we either reach a state in which every vertex has a unique label in its graph, or the sets of labels present in the two graphs differ. In the first case, we can conclude that the graphs are indeed isomorphic, in the second that this is not the case. Note, however, that this procedure does not work under all circumstances (cf. Douglas 2011).

Anyhow, the number of equal labelings in the two graphs is quite a good indicator of how similar they are, and so the basic idea of the WL kernel for general graphs is to count the number of pairs of vertices with equal labels where one vertex belongs to the first and the other to the second graph. These counts are ascertained for a number of iterations, and a weighted sum of them is used as output of the kernel function.

The general WL kernel follows the usual procedure discussed already, i.e. we first generate a graph for each vertex, and use these to compute the value of the kernel function. In De Vries (2013), however, the author reversed these steps in order to create the WL kernel for RDF data, i.e. he performed the iterative relabeling on the entire RDF graph first, and extracted the graphs representing the vertices subsequently. This approach proved to be superior in connection with RDF data in a number of experiments, and can be implemented efficiently as well.

In my own experiments, I employed two state-of-the-art kernel methods together with the kernels for RDF data just introduced, namely the *soft-margin support vector machine* (SM-SVM) and *kernel logistic regression* (KLR).

3.3.2 Transductive Learning

An interesting approach, which came up in recent years only, is based on *transductive inference* (Gammerman et al. 1998; cf. Vapnik 1998, Chapter 8 and Chapelle et al. 2006, Chapter 24 for more comprehensive treatments). Vladimir Vapnik, who coined that term, once described the main principle underlying this kind of inference as follows:

> »*If you possess a restricted amount of information for solving some problem, try to solve the problem directly and never solve a more general problem as an intermediate step. It is possible that the available information is sufficient for a direct solution but is insufficient for solving a more general intermediate problem.*« (Vapnik, 1998, p. 12)

With this statement he actually addressed the problem that classical methods of statistical modelling are not particularly well-suited for analyzing small datasets. Therefore, he advocated, it is sometimes beneficial to consider a problem not in its full generality, but to solve it as straightforward as possible.

However, the proposition does not apply to classical statistics only, and so two approaches that follow this thought were developed for reasoning in the SW as well: *adaptive knowledge propagation* (AKP; Minervini et al., 2014a) and *Gaussian processes knowledge propagation* (GPKP; Minervini et al., 2014b). Both approaches are based on the same ideas, and achieved very similar results in the published experiments. Anyhow, the results reported for AKP were slightly better, which is why I chose to focus on it here.

In the spirit of transductive inference, the authors did not try to predict an individual's membership to a certain class based on its properties, i.e. the literals it is connected with. As instead, they strived to answer this question directly from the knowledge that we have about whether related individuals belong to the considered class or not. AKP achieves this as follows:

- Let $\tilde{\mathbf{W}}_1, \ldots, \tilde{\mathbf{W}}_r$ be the adjacency matrices for all relations in the dataset, and $\mathbf{f}$ a binary vector with one entry for every individual in the KB. The vector $\mathbf{f}$ indicates whether an individual belongs to the considered class, i.e. the corresponding entry is 1, or not, in which case the respective value is 0. Note that some individuals' entries in $\mathbf{f}$ are undefined, which are exactly those that we aim to predict. An interesting fact, to note, is that, unlike most other ML algorithms, AKP predicts all of the missing class memberships at once rather than one at a time.
- Next we define a weight matrix
$$\mathbf{W} = \sum_{i=1}^{r} \mu_i \tilde{\mathbf{W}}_i,$$
where $\boldsymbol{\mu} = (\mu_1, \ldots, \mu_r)^T$ is a parameter of the model and consists of non-negative real values.
- Now in order to classify those individuals whose entries in $\mathbf{f}$ are unspecified, we simply define an error function
$$E(\mathbf{f}) = \frac{1}{2} \sum_i \sum_j \mathbf{W}_{ij} (\mathbf{f}_i - \mathbf{f}_j)^2 + \epsilon \|\mathbf{f}\|_2^2$$
and minimize it over all possible values of the undefined entries:
$$\hat{\mathbf{f}} = \arg\min_{\mathbf{g}} E(\mathbf{g}),$$
where $\mathbf{g}_i = \mathbf{f}_i$ has to hold if $\mathbf{f}_i$ is defined. This way we turned a reasoning problem into one of optimization, which can be solved easily using standard methods.

The intuition underlying this approach is that related individuals tend to belong to similar classes. However, this is, of course, not always the case. Therefore, $\boldsymbol{\mu}$ controls how strong, we believe, this correlation among different pairs of classes is.

Now the last point, that remains to be clarified, is how to learn the parameters $\boldsymbol{\mu}$ and ϵ. However, this turns out to be easier than one might think. If we choose a differentiable error function, e.g. the least-squares error, then the derivatives with respect to both of the parameters have a closed form (Minervini et al., 2014a). Therefore, we can simply employ some kind of gradient-based optimization in order to choose suitable values for $\boldsymbol{\mu}$ and ϵ.

3.3.3 Statistical Relational Learning

The field of statistical relational learning (SRL) is a subdiscipline of statistical learning, and is concerned with learning from relational datasets as opposed to the »*flat*« vector space representations used for most techniques of ML (cf. Getoor and Taskar, 2007, for an introduction to SRL as well as an overview of the field). Not surprisingly, people started to employ methods of SRL to learn from SW-KBs too, and so a number of interesting approaches have been developed. One of the most successful learning methods, which is clearly state-of-the-art today, is called *RESCAL* (Nickel et al., 2011).

RESCAL is based on tensor factorization, which came to be used for SRL in recent years only, and works as follows:

- We begin by defining an adjacency matrix $\mathbf{X}$ as a three-dimensional tensor, i.e. $\mathbf{X} \in \mathbb{R}^{n \times n \times r}$ where n denotes the number of individuals and r the number of relations in the dataset, such that

  $$\mathbf{X}_{ijk} = \begin{cases} 1 & \text{if the relationship } relation_k(entity_i, entity_j) \text{ exists, and} \\ 0 & \text{otherwise.} \end{cases}$$

  So, $\mathbf{X}$ integrates the adjacency matrices of the different relations in a dataset—for AKP we had $\tilde{\mathbf{W}}_1, \dots, \tilde{\mathbf{W}}_r$, e.g.—into a single tensor.

- RESCAL belongs to the category of latent variable models. It aims to factorize $\mathbf{X}$ into a core tensor $\mathbf{R} \in \mathbb{R}^{m \times m \times r}$ and a factor matrix $\mathbf{A} \in \mathbb{R}^{n \times m}$ such that

  $$\mathbf{X} \approx \mathbf{R} \times_1 \mathbf{A} \times_2 \mathbf{A}.$$

  This means that $\mathbf{X}_{ijk} \approx \mathbf{A}_{i\cdot}^T \mathbf{R}_{\cdot\cdot k} \mathbf{A}_{j\cdot}$. Therefore, we can, intuitively, interpret $\mathbf{A}_{i\cdot}$ as the latent representation of individual i, and $\mathbf{R}_{\cdot\cdot k}$ as a matrix which models interactions for relation k.

- Now, in order to actually retrieve $\mathbf{R}$ and $\mathbf{A}$, we simply have to solve a regularized minimization problem for each frontal slice of $\mathbf{X}$ (cf. Nickel et al. 2011).

Up to this point we have not wasted a single word on classification, yet we have all we need already. A certain class membership can be modeled as a relation easily, and so the classification task is turned into one of link prediction. This problem, however, is solved by reconstructing $\mathbf{X}$ from $\mathbf{R}$ and $\mathbf{A}$ without effort.

Another approach, that is sometimes considered state-of-the-art, is called *learning with statistical units node sets*, or just SUNS for short (Tresp et al., 2009). However, RESCAL was shown to be superior to SUNS, which is why I am not going to consider the latter in this work.

3.4 Yet Another Approach

Without being guilty of exaggeration, one can say that DL has been the most important advancement in the field of ML in the last decades, and so it is hardly surprising that methods based on DL represent the state-of-the-art in a variety of fields like computer vision or NLP. However, what is really fascinating about it is that DL seems to be almost domain-agnostic, as it has been used successfully for a plethora of different applications.

With that said, it seems worthwhile to investigate the application of DL techniques in the context of the SW as well, and to the best of my knowledge, it has not been used to tackle the problem of instance checking yet. In the next chapter, we will take a look at a new approach based on techniques from this field and all of its gory details, but for now let's just consider some general thoughts about why this seems to be a promising idea:

- First and foremost—and just like mentioned already—, DL proved to be a great tool for a variety of tasks. In general, however, deep NNs require a great amount of training data in order to generalize well to unseen instances. But this issue seems quite attenuated if we consider that typical SW-KBs nowadays tend to be rather big, and so this approach might be just appropriate for the era of big data.
- As suggested in the introductory chapter already, a number of huge SW-KBs are freely available today. Therefore, we have access to a sufficient amount of data in order to train and test deep models adequately, and thus all we need to challenge our own approach with a number of different scenarios.

However, there are also a few rather challenging circumstances that we should be aware of:

- Standard NNs are tailored to learn from flat data, while SW-KBs are really graphs. Hence, one might take recursive NNs as a point to start from, yet it is not clear how to generalize them from trees to arbitrary directed graphs with cycles, just like a typical RDF graph.
- Another interesting question is what to use as input for such an NN. Individuals in a SW-KB are identified by their URIs, but NNs, of whatever type, operate on real vectors.

- One important part of DL, which can have major influence on a model's performance, is pretraining. Therefore, we might have to adapt existing strategies to the considered problem in order to achieve the best generalization possible.

Many researchers who work in this area make a point employing ML techniques that produce models that are easily interpretable. NNs and especially deep models, however, do not exhibit this particular quality in general, and so it can be rather challenging to reveal what is actually going on in such a model. I hold the opinion, though, that interpretability is not necessarily required for many real-world applications. Often times ML can aid in drawing conclusions obvious for human beings, while formal reasoning might just be infeasible due to some of the problems discussed already. However, if the primary goal is to expose unknown regularities in a KB, then DL might not be the best choice, admittedly.

Now before we close this section, let's get back to the issues discussed before, one last time, and have a look at how I intend to deal with some of them:

- In the context of ontological KBs, the problem of incomplete information is somewhat special. While we can spot missing values easily in traditional ML settings, it is in general not possible to detect a certain lack of information in the SW without human assistance. Therefore, I adopted the philosophy to not expect anything being present in the dataset beforehand, and the next chapter explains how this is done.
- NNs, and thus DL, belong to the category of eager learning techniques, i.e. in general we train a model once and use it to classify new instances in constant time subsequently. However, since SW-KBs are graphs of arbitrary structure we are not able to achieve classification in constant time. Nevertheless, we will see later on that the complexity of classifying individuals following our approach is bounded by the maximal vertex degree in the considered RDF graph and thus aids scalability significantly.
- Regarding the problems of conflicting data and complexity of construction, the suggested approach brings the same advantages as any other one based on ML, and there is not much more to say here in particular. Note, however, that we are not going deal with the issue of uncertainty in this thesis.

4

CHAPTER

Deep Learning for the Semantic Web

In this chapter, we develop a NN model for predicting class memberships of individuals in a SW-KB, and, as suggested at the end of the last chapter already, we will make use of recursive NNs to compute vector representations of the individuals in question based on the RDF graph of the underlying KB. These vector representations, in turn, can be fed into another model in order to calculate the desired prediction.

The remainder of this chapter is organized as follows:

- Section 4.1 introduces some general notions, and examines how NNs are applied to relational data already,
- Section 4.2 presents the concept of vector space models of semantics,
- Section 4.3 provides a first, informal glimpse of the suggested model,
- Section 4.4 illuminates the basic intuition behind the chosen approach,
- in Section 4.5 we define the problem to be solved formally and in a more general setting,
- Section 4.6 develops the full model step by step,
- Section 4.7 discusses different aspects of pretraining, and finally,

- Section 4.8 outlines a few characteristics of the newly introduced model.

4.1 Introduction

If we compare the problem of instance checking with the typical settings that NNs are used in, then we can observe one rather distinctive trait: *SW data is inherently relational.* However, it needs to be pointed out straightforwardly that NNs have been used to learn from relational datasets with great success. In fact, some of their most important fields of application are concerned with relational data. Nevertheless, the structure of the training sets encountered in those areas is usually much simpler than a typical RDF graph, which is why the related tasks are often not considered as relational learning. Another interesting aspect to note is that there is no general type of NN capable of learning from arbitrary relational datasets out-of-the-box. Instead, people have employed different variations of NNs in order to be considerate of the circumstances that specific fields of application imply.

To start with, let's have a quick look at two examples to get an idea of how NNs are applied to learn from certain kinds of relational data.

Example 4.1 The prime example of a successful application of NNs to a relational learning task is *computer vision.* As depicted in Figure 4.1, image data can be interpreted as a relational dataset easily. To this end, we regard every pixel as an individual, and there is only one kind of relation among them, namely a neighborhood relation which indicates that two pixels are side by side in the image—additionally, one might take the particular »*orientation*« of such a neighborhood into account. Furthermore, every pixel is related to three numeric literals, which indicate its color in terms of an RGB value. In practice, however, images are frequently converted to grayscales such that a pixel is related to a single literal only.

The most important type of NN used for computer vision are so-called *convolutional neural networks* (ConvNets; LeCun, 1989). These consider fixed blocks of pixels, and map them to single values independently of the remaining parts of the image. This operation is called *convolution*, and it is employed to reduce the data step-by-step until we reach some vector that is eventually used to calculate a prediction. △

Example 4.2 Another interesting example is the employment of NNs on *NLP.* In this case, the structure of a dataset is even simpler than the image data considered for computer vision, and the only relation that we have to deal with is the neighborhood of two consecutive words in a text. It turns out that there are several ways to employ NNs on different NLP tasks, but one interesting example is the application of RNTNs. Since the words in a text are linearly ordered, we can easily represent (parts of) sentences as trees, which, in turn, enables us to use an RNTN to compute vector representations as illustrated in Figure 4.2. These vectors are then used to calculate the desired prediction or are considered as such already. Note that, for

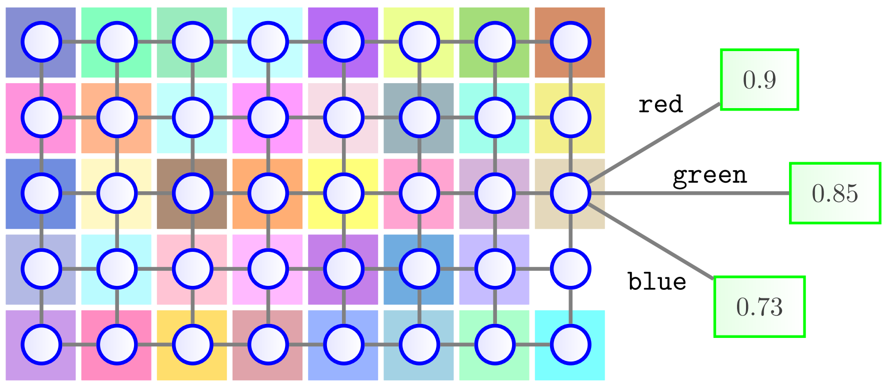


Figure 4.1: An interpretation of an image as a relational dataset.

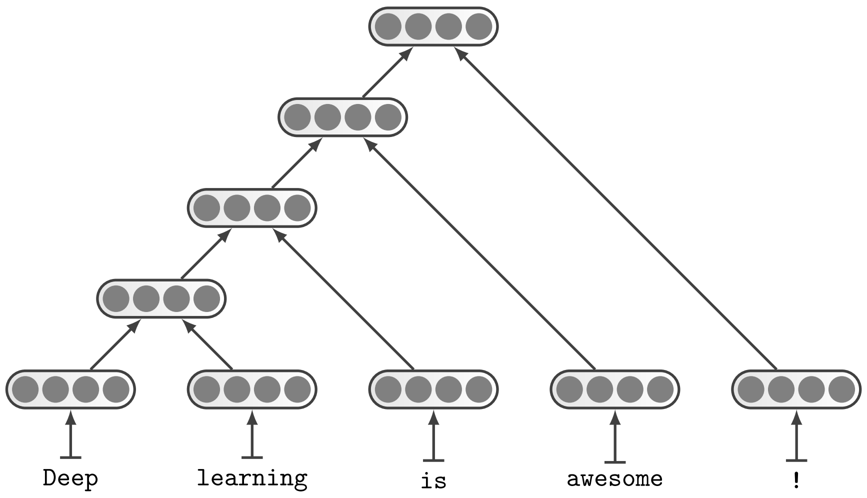


Figure 4.2: This figure illustrates how an RNTN is used to map the sentence »*Deep learning is awesome!*« to a single vector like it is done, e.g., for NLP.

NLP, every word is assigned a certain vector, a so-called word embedding, and thus we can use those as inputs to the RNTN. △

In general, the presence of relations in a dataset necessitates us to change the way we think about our data. When we are faced with a flat dataset, then every tuple contained is basically a point in some, possibly high-dimensional, vector space. In a relational setting, however, things change fundamentally. Relations among individuals constitute big parts of the total information we have, and so we can certainly not just disregard them. However, we could try to map a relational dataset to some suitable flat equivalent in order to make the tools that we have already practicable. But this is not a trivial task. Nevertheless, it is the road that we are going to follow, and it turns out that there are indeed ways to construct mappings that do preserve big parts of the information hidden in the relations of a dataset. Still, it should be clear that we usually have to accept a certain loss of information for all but the easiest learning tasks.

4.2 Vector Space Models of Semantics

Now, how could a set of vectors possibly reflect information previously posed in terms of relations? Well, there are basically two ways to achieve this. First, we can try to map individuals to vectors in such a way that certain areas of the space correspond to some characteristics that an individual might have. And second, we could construct the mapping such that differences between vectors reflect particular relations among them.

To illustrate these ideas, let's consider Figure 4.3. In the upper part of this figure we see a relational dataset—as usual depicted as graph—describing the relations among two famous cartoon characters and their pets, and right below we have a display of the same data as vectors in $\mathbb{R}^3$. It is easy to see that small values of x correspond to human beings, while large values represent animals. Furthermore, within the group of animals we distinguish cats and dogs based on the y-coordinates of the vectors. This demonstrates the use of the first of the ideas mentioned before in order to encode the relation `is`, and is illustrated explicitly in Figure 4.4. The second kind of relation, `has pet`, is encoded using the other approach. Therefore, the differences between the vectors representing human beings and those denoting their pets are, at least to some degree, similar to each other. Note that the different z-values ensure that no one is associated with someone else's pets.

By mapping a relational dataset to a vector space like this we create what is usually referred to as *vector space model of semantics* (Turney and Pantel, 2010). However, at this point we should ask ourselves if this is possible in general, i.e. can we actually represent an *arbitrary* relational dataset as a set of points in some suitable vector space without loosing the bulk of the information provided by links present in the data? Although we can hypothesize only, there are good reasons to believe that the answer is *yes*. It has been shown in a multitude of papers that vector spaces are indeed capable of representing rich semantic structures. A great example is the success of deep learning for NLP, which is heavily based on such vector space models (Socher et al., 2012, 2013a,b). Furthermore, there are a number of published papers that address and positively answer the question of whether NNs can actually learn to do some kind of formal reasoning based on data that is given as vectors (Bowman, 2013; Bowman et al., 2015). This, once again, requires a model to make efficient use of vector space semantics, and so the results support the stated hypothesis as well.

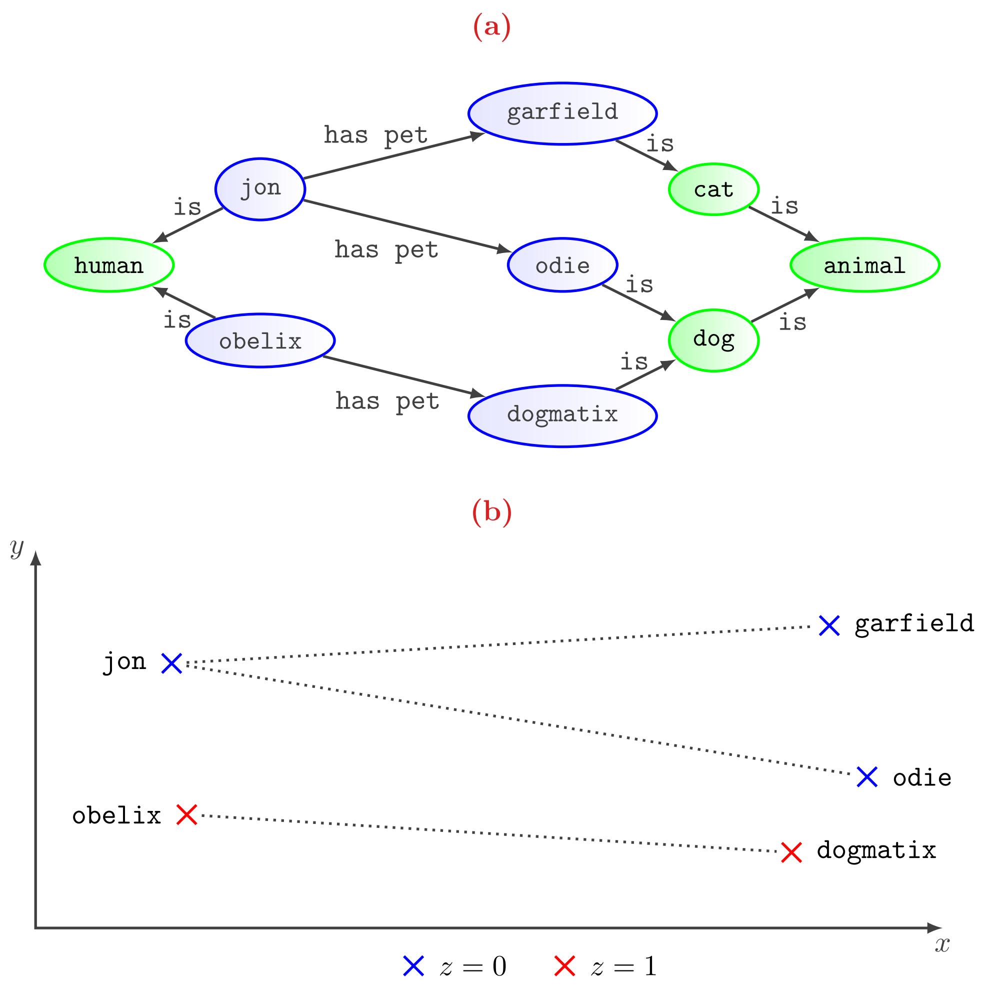


Figure 4.3: This is a toy example that illustrates the use of vector space models of semantics. Part (a) of the figure depicts a simple relational dataset, and part (b) represents the same information as vectors in $\mathbb{R}^3$.

4.3 An Informal Glimpse at a New Approach

Now suppose that we wanted to make use of vector space semantics in order to tackle the task of instance checking, i.e. predicting whether individuals of an ontological KB belong to a certain RDF class. Inspired by the examples discussed at the beginning of this chapter, we could try to take a similar approach, and assemble vector representations for individuals in a dataset incrementally. However, for doing this, we need to clarify two issues, namely how to generate initial vectors for individuals in a SW-KB on the one hand and how to update these step-by-step on the other hand.

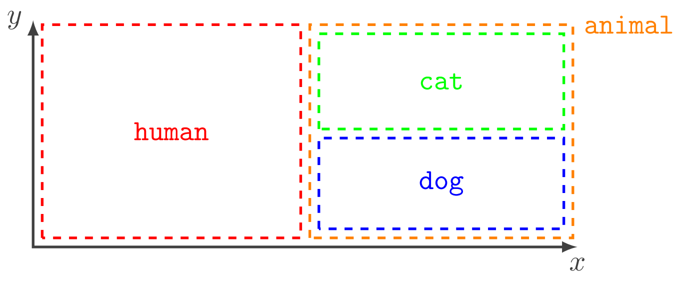


Figure 4.4: This plot describes how vector space semantics are used to categorize individuals from the dataset depicted in Figure 4.3.

Let's elaborate on this idea by means of an example, and thus consider the simple KB depicted in Figure 4.5. In this RDF graph, the resources, literals and relations that illustrate the training data are colored as usual, while the individual that is subject to instance checking is colored red—accordingly, we will also refer to it as *red*.

To make things a little easier, let's assume for the moment that we have vector representations for all of the individuals in the training set already, and so we can focus on computing an according representation for *red*. To that end, one approach could look like this:

- As a first step, we need to generate an initial vector for the individual *red*. For this purpose, we start by creating an incidence vector $v_{red}^{(0)}$ that indicates the RDF classes that *red* belongs to. Let's use $\mathcal{K}$ to refer to the KB in Figure 4.5, and suppose that $c_1, c_2, \dots, c_n$ are all the RDF classes present in $\mathcal{K}$. Then we define $v_{red}^{(0)} \in \{-1, 0, 1\}^n$ and

$$\left(v_{red}^{(0)}\right)_i = \begin{cases} 1 & \text{if } \mathcal{K} \models (red, \texttt{rdf:type}, c_i), \\ -1 & \text{if } \mathcal{K} \models \neg(red, \texttt{rdf:type}, c_i), \text{ and} \\ 0 & \text{otherwise.} \end{cases}$$

  This vector $v_{red}^{(0)}$ serves as initial representation of *red*, and at this point we are, from a conceptual stance, facing the situation depicted in Figure 4.6. In this picture, every resource has been replaced with an according vector representation, and all of the literals were removed from the data. However, this makes perfect sense, since these vectors should summarize everything that we need to know about the single resources, including information about related literals—we do not consider the role of literals in this section, but we will come back to this aspect in short a little later.

- Next, we want to make use of the structured nature of our data, and leverage the relations present in the KB. Therefore, we consider all of the resources that *red* is in relation with one-by-one, and adjust *red*'s vector representation accordingly. This is accomplished by means of an *RNTN*, which takes the

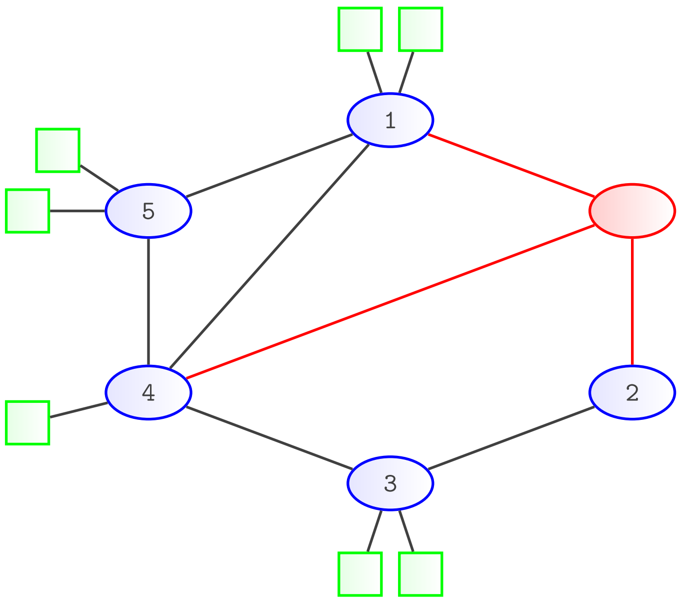


Figure 4.5: This is the RDF graph of a simple SW-KB consisting of six individuals, a single type of relation, and seven literals not further specified. In this graph, the parts which are colored as usual constitute our training set, while the red parts are newly encountered, and the red individual, in particular, is the one to be classified.

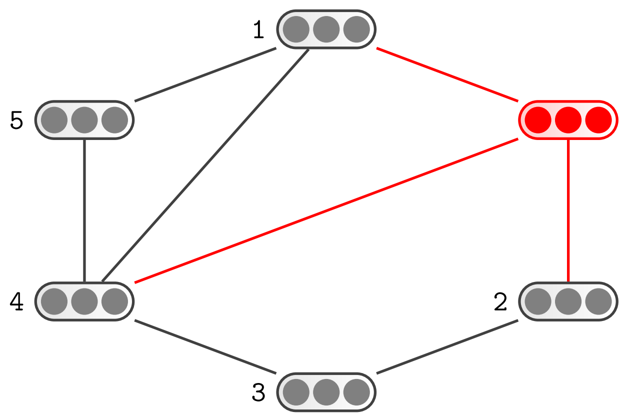


Figure 4.6: After an initial vector representation of the red individual in Figure 4.5 has been computed, we identify all resources with their vectors for the remaining part of the classification task. This also means, that we disregard literals connected to any of the individuals for the further steps in synthesizing *red*'s representation.

vectors of two individuals as input and yields an updated version of its first argument as output. Intuitively, a tensor layer of this network investigates the representations of two resources, and incorporates the information about the relation that they are in into one of them. In the case of individual *red*, which is related to the resources 1, 2, and 4, this process is depicted in Figure 4.7, and the network's output is considered as *red*'s final vector representation that we were looking for. Note that the order in which we consider *red*'s neighbors in the RDF graph does not matter.

After we generated a vector representation for the individual *red* like this, we can use it as input to another ML model in order to predict whether *red* belongs to the RDF class of interest or not. However, this section outlines how we suggest to approach instance checking for arbitrary SW-KBs at a very basic level only. It leaves a number of important questions untouched, and all of the details will be elaborated in-depth in the next few sections.

4.4 Intuition

Before we move on to the formal definition of the learning problem that we aim to master, it is instructive to have a closer look at the intuition underlying the way we assemble vector representations of individuals. Therefore, let's consider the very simple KB depicted in Figure 4.8. This KB contains three individuals, $\{blue, red, green\}$, as well as a single type of relation not further specified.

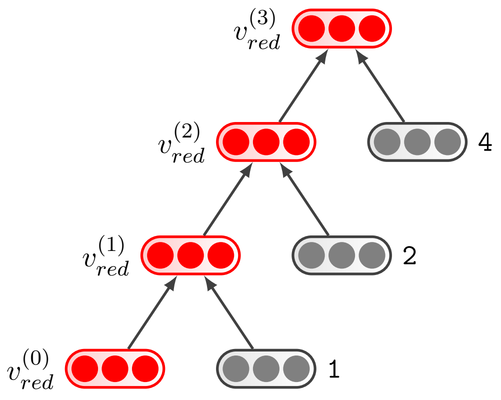


Figure 4.7: Given the setting depicted in Figure 4.6, the next step is to refine *red*'s vector representation based on the relations it is involved in. Therefore, we consider all of the individuals that *red* is related to—i.e. resources `1`, `2`, and `4`—and use an RNTN to adjust its vector stepwise.

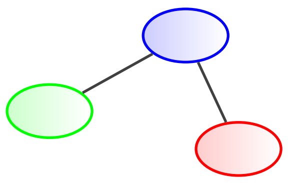


Figure 4.8: This is the RDF graph of a very simple KB consisting of three individuals—*red*, *green*, and *blue*—and a single kind of relation.

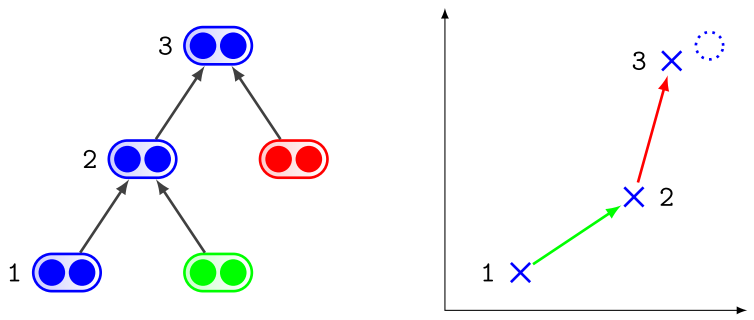


Figure 4.9: This figure demonstrates how the initial vector representation of the individual *blue*—marked as number 1—in the KB given in Figure 4.8 is updated step by step. The left part of the figure shows how an RNTN is used to update *blue*'s vector based on the relations it is involved in. The right part depicts the vector space used for representing resources, and illustrates how the recursive network shifts the vector representation of *blue* through the space.

Now let's suppose that we have a trained model, and need to compute a vector for *blue* while we have representations for *red* and *green* already. Furthermore, let's assume that our model employs $\mathbb{R}^2$ as vector space for the individuals.

The whole process of computing a vector for *blue* is illustrated in Figure 4.9, and, according to what was outlined in the previous section, we start by generating an initial vector. During the training phase, the model learned how to map individuals in the KB to appropriate vectors such that the target space embodies the semantics governed by the underlying ontology as good as possible. Therefore, the initial representation could be regarded as a first »*guess*« of where *blue*'s vector is supposed to be placed in this vector space based on what we know about its class memberships. This vector is marked as `1` in Figure 4.9. Now, since the location of individuals in the vector space is predicated on the semantics specified by the considered ontology, there is a certain point in the space which coincides with the meaning of *blue* as part of the KB. This value is marked as dotted circle on the right side of Figure 4.9, and we can see that it is a bit away from the initial representation we obtained.

However, we are not quite done with computing *blue*'s representation. The next step in our algorithm is to consider *blue*'s neighbors in the RDF graph, and update its vector representation using an RNTN as illustrated on the left side of Figure 4.9. Intuitively, what happens is that we »*tell*« our model that certain relations between *blue* and other individuals exist by supplying their vector representations to the recursive network. The network, in turn, assimilates this information by updating *blue*'s vector appropriately such that the initial representation is shifted through the vector space and towards the optimal value (as depicted on the right side of Figure 4.9).

4.5 Problem Formulation

In this section, we want to state the problem setting more formally, and a good point to start from is what we referred to as feature space before. Obviously, the original »*flat*« definition of this term is no adequate means to describe relational data, like SW-KBs, and so we need to find a more suitable data structure. After short reflection, it seems like the logical choice is to employ a graph-like structure rather than a set of vectors, and it turns out that a directed vertex-labeled graph with multiple types of edges is exactly what we need. Due to the lack of a better name, we will call this data structure *feature graph*, and consider it a relational extension of the traditional feature space. Moreover, it is convenient to include the target space, which does not require any adjustments by itself, into the feature graph as well, and so we arrive at the following definition:

▶ **Definition 4.1 (Feature Graph)** *Let $\mathcal{V}$ be a set of vertices and $\mathcal{E} = \{\mathcal{E}_1, \mathcal{E}_2, \dots\}$ a family of types of edges, i.e. $\mathcal{E}_i \subseteq \mathcal{V} \times \mathcal{V}$ for $\mathcal{E}_i \in \mathcal{E}$. Furthermore, let $\mathcal{X}$ be a feature space, $\mathcal{Y}$ a target space, $\mathfrak{f} : \mathcal{V} \to \mathcal{X}$ a function called feature map, and $\mathfrak{g} : \mathcal{V} \to \mathcal{Y}$ a function called target map. Then the graph*

$$\mathcal{G} = (\mathcal{V}, \mathcal{E}, \mathcal{X}, \mathcal{Y}, \mathfrak{f}, \mathfrak{g})$$

is called feature graph. △

In this definition, the set $\mathcal{V}$ consists of all (distinguishable) objects in the considered data, and $\mathcal{E}$ contains one set for each kind of relation among them. Furthermore, $\mathcal{X}$ is a common flat feature space, which is used to describe properties of the individual objects in $\mathcal{V}$ irrespective of any relations present in the graph. Just like that, $\mathcal{Y}$ is the common target space that we are interested in. The functions $\mathfrak{f}$ and $\mathfrak{g}$ associate all of the objects with corresponding elements in $\mathcal{X}$ and $\mathcal{Y}$, respectively, and could be regarded as two different labelings of the elements in $\mathcal{V}$. Note, however, that these labels are not necessarily unique. Furthermore, it is easy to see that a feature space is really just a feature graph with one distinct vertex for every vector in the space, and without any edges at all.

Now that we have this new notion of a feature graph, the familiar training set calls for an according modification too:

▶ **Definition 4.2 (Training Graph)** *Let $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \mathcal{X}, \mathcal{Y}, \mathfrak{f}, \mathfrak{g})$ be a feature graph, $V \subseteq \mathcal{V}$, and $E = \{E_{i_1}, E_{i_2}, \dots, E_{i_r}\}$ such that $\mathcal{E}_{i_m} \in \mathcal{E}$ and $E_{i_m} \subseteq \mathcal{E}_{i_m} \cap (V \times V)$ for $1 \leq m \leq r$. Furthermore, $f = \mathfrak{f}|_V$ and $g = \mathfrak{g}|_V$. Then*

$$\mathcal{T} = (V, E, f, g)$$

is called training graph. △

So, a training graph $\mathcal{T}$ is basically a subgraph of some feature graph $\mathcal{G}$ with feature and target map restricted to the vertices it contains, and we are going to use $\mathcal{T} \leq \mathcal{G}$

in order to denote this subgraph relationship. In the remainder, we are going to stick to the notation used in these two definitions. However, if there is any risk of confusion, then we will simply add a subscript identifying the according feature or training graph.

Let me stress at this point that the definitions above are rather general, and not related to any kind of relational data in particular. Hence, we need to consider how to describe an arbitrary ontological KB in our newly introduced terminology next. However, to alleviate the subsequent treatment a little, let's introduce the *class indicator function* first:

▶ **Definition 4.3 (Class Indicator)** *Let $\mathcal{K}$ be a SW-KB and $c, c_1, \dots, c_k$ RDF classes in $\mathcal{K}$. Then we define the class indicator function as follows:*

$$\mathbb{1}_c^{\mathcal{K}}(x) = \begin{cases} 1 & \text{if } \mathcal{K} \models (x, \texttt{rdf:type}, c), \\ -1 & \text{if } \mathcal{K} \models \neg(x, \texttt{rdf:type}, c), \text{ and} \\ 0 & \text{otherwise.} \end{cases}$$

This function is easily generalized to treat multiple classes simultaneously:

$$\mathbb{1}_{c_1,\dots,c_k}^{\mathcal{K}}(x) = \begin{bmatrix} \mathbb{1}_{c_1}^{\mathcal{K}}(x) \\ \mathbb{1}_{c_2}^{\mathcal{K}}(x) \\ \vdots \\ \mathbb{1}_{c_k}^{\mathcal{K}}(x) \end{bmatrix}.$$

△

Now let's return to the question of how to describe a KB in terms of feature and training graphs. For this purpose, let's suppose that we are given some SW-KB $\mathcal{K}$, and we want to convert it into an equivalent training graph $\mathcal{T} = (V, E, f, g)$. Considering a KB as a training graph fits our definitions perfectly well, since, intuitively, $\mathcal{K}$ reflects just a certain fraction of the real world, and might thus be considered as part of some idealized, complete feature graph $\mathcal{G}$.

It turns out that there are various ways of formalizing ontological KBs as training graphs, but we will stick to the following, very natural one, which we refer to as the *canonical translation*, denoted as $\mathcal{CT}(\mathcal{K}, c)$. Notice that we always define translations with respect to a certain problem of instance checking, and the parameter c describes the RDF class that we are interested in. However, the canonical translation can be adapted to other reasoning problems easily.

Now we are ready to define the canonical translation $\mathcal{CT}(\mathcal{K}, c) = (V, E, g, f)$ of KB $\mathcal{K}$ for instance checking with respect to class c:

- First of all, it is clear that the vertices in $\mathcal{CT}(\mathcal{K}, c)$ are simply the individuals present in $\mathcal{K}$. However, we exclude all resources that represent RDF classes, i.e. those of type `rdfs:Class`, and the reason for doing this will become obvious soon. For the remaining individuals, we collect their URIs, and add them to the set V.

- Assembling set E is pretty straightforward too, since it has to contain one set of edges for each kind of relationship in the data. So, if $\mathcal{K} \models (s, p, o)$, then there is a set $E_p \in E$, and $(s, o) \in E_p$. Note, however, that we have to disregard any links which include an RDF class either as subject or object, because we excluded those from V.
- Although an RDF graph might exhibit an arbitrary structure, there are a number of characteristics that we can use to describe the individuals contained uniformly, namely the memberships to certain classes. Now suppose that $c_1, c_2, \ldots, c_k$ are all RDF classes which are present in $\mathcal{K}$, and that are different from c. Then we define

  $$\mathcal{X} = \{-1, 0, 1\}^k$$

  and

  $$f : \left\{ \begin{array}{lcl} V & \rightarrow & \mathcal{X} \\ x & \mapsto & \mathbb{1}^{\mathcal{K}}_{c_1,\ldots,c_k}(x) \end{array} \right. .$$

  This means that f maps individuals to incidence vectors, which reflect our knowledge about an individual's membership to any of the classes in the considered KB except c. To that end, the i-th entry of such an incidence vector has a value of 1 if the individual belongs to class c_i, and -1 if we know that this is not the case. However, if do not now whether a resource belongs to a class or not, then the respective entry is 0.
- Since we are interested in predicting the membership to class c, we define the target space and map similarly as

  $$\mathcal{Y} = \{-1, 0, 1\}$$

  and

  $$g : \left\{ \begin{array}{lcl} V & \rightarrow & \mathcal{Y} \\ x & \mapsto & \mathbb{1}^{\mathcal{K}}_{c}(x) \end{array} \right. .$$

At the bottom line, the canonical translation takes all information related to RDF classes and their memberships out of the graph, and represents it through appropriate mappings to a feature and target space, respectively, instead. However, there is one important thing notice to about this: $\mathcal{CT}(\mathcal{K}, c)$ does not include any information about literals any more. This might seem like a big mischief at first sight, but there is just no useful way to account for them in general. If we want to include (some of) the literals present in $\mathcal{K}$ in the training graph anyway, then we can simply consider them as additional dimensions of the feature space $\mathcal{X}$. We need to be aware of the fact, however, that many literals will be missing for all but a few of the individuals, and might thus introduce a certain bias into any ML model trained on the resulting graph. Therefore, the question of which literals to include needs to be answered case-by-case, and I will not elaborate on it any further in this work.

Table 4.1 describes the canonical translation more formally, and in the remainder of this thesis we will always make use of it without mentioning this explicitly. However,

The Canonical Translation. Let $\mathcal{K}$ be an arbitrary SW-KB and $c, c_1, \dots, c_k$ all classes it contains. Suppose further that we want to implement instance checking for class c. Then we define the canonical translation $\mathcal{CT}(\mathcal{K}, c) = (V, E, f, g)$ of $\mathcal{K}$ as follows:

$$
\begin{aligned}
V &= \big(\{x \mid \exists p, o \in \text{resources}(\mathcal{K}) : \mathcal{K} \models (x, p, o)\} \\
&\quad \cup \{x \mid \exists s, p \in \text{resources}(\mathcal{K}) : \mathcal{K} \models (s, p, x)\}\big) \\
&\quad \setminus \{c \mid \mathcal{K} \models (c, \texttt{rdf:type}, \texttt{rdfs:class})\} \\[1em]
E &= \{E_p \mid \exists s, o \in V : \mathcal{K} \models (s, p, o)\} \\
E_p &= \{(s, o) \mid s, o \in V \wedge \mathcal{K} \models (s, p, o)\} \\[1em]
f &: V \to \mathcal{X} \\
&\quad x \mapsto \mathbb{1}^{\mathcal{K}}_{c_1, \dots, c_k}(x) \\[1em]
g &: V \to \mathcal{Y} \\
&\quad x \mapsto \mathbb{1}^{\mathcal{K}}_{c}(x)
\end{aligned}
$$

In this definition, $\mathcal{X} = \{1, 0, -1\}^k$ and $\mathcal{Y} = \{1, 0, -1\}$.

Table 4.1

before we move on to develop a deep model for relational data now, I want to extend Definition 2.5 of ML to the terminology introduced in this section:

▶ **Definition 4.4 (Relational Machine Learning)** *Let $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \mathcal{X}, \mathcal{Y}, \mathfrak{f}, \mathfrak{g})$ be an arbitrary feature graph and $\mathcal{T} = (V, E, f, g)$ a corresponding training graph, i.e. $\mathcal{T} \leq \mathcal{G}$. The task of finding a function $h : \mathcal{V} \to \mathcal{Y}$ that resembles $\mathfrak{g}$ as closely as possible based on $\mathcal{T}$ is called relational ML.* △

Just like before, this definition is appropriate for this thesis, and I do not make any claims of correctness or universal applicability.

4.6 A Deep Model for Relational Data

In this section, we deepen the approach outlined in Section 4.3, and cast it into a formal model. Note, however, that we consider relational ML, as specified in Definition 4.4, in general rather than learning from SW-KBs in particular, since the latter is, as illustrated in the last section, just a special case of the prior. Let's start with introducing a few concepts that we will need later on in order to describe our deep model.

4.6.1 Preparatory Concepts

Probabilistic Breadth-first Search

One technique that we are going to use all the time is a variation of the search strategy breadth-first search (BFS; Cormen et al. 2009, Section 22.4), and we call it *probabilistic breadth-first search* (PBFS)—not to confuse with the eponymous but unrelated algorithm that is used in the context of Wireless Mesh Networks. The reason why we call it »*probabilistic*« is that during the course of the search nodes are expanded with a certain probability only. However, we actually do not employ PBFS for search, but rather as a tool for extracting subgraphs of training graphs. Therefore, we are interested in the search trees generated by the algorithm only, and these are created as follows:

- In general, we proceed like standard BFS, and expand the search tree layer-by-layer. However, in doing so, the single nodes are expanded with a certain probability only. This probability depends on the layer of the search tree, and is defined in terms of a discount factor $\alpha \in (0, 1]$. For $i \in \mathbb{N}$, every node in the i-th layer—the root is considered as layer 0—of the tree is expanded, independently of the other nodes in this layer, with probability α^{i-1}.
- When a node is selected for expansion, then we check whether a copy of it has been expanded already (at some other place in the search tree). If this is the case, then we skip it, and otherwise we expand it, like discussed already, with a certain probability.
- If a node is expanded, then we add its neighbors in the graph as children to the search tree. However, we exclude those that exist in the same branch of the search tree already.

Figure 4.10a shows a simple example of a graph, and 4.10b an according search tree generated by PBFS starting from node `1` with $\alpha = 1$, i.e. every node is expanded with probability 1 if it is not excluded for other reasons. During the search, I always considered a node's neighbors in ascending order with respect to their labels, and the nodes in a layer of the search tree were expanded left-to-right.

So, one could consider PBFS as a mixture of the tree search and graph search versions of the standard BFS, since nodes might be added to a search tree multiple times, but are expanded at most once. However, unlike in the example, we usually use a value of α that is lower than one. This entails that the branches of a search tree are usually truncated at some point, and the longer a branch is already the lower is the probability that it gets extended any further. Figure 4.11 shows a tree that contains all possible branches that a PBFS starting from node `1` in the graph from Figure 4.10a might generate together with the expansion probability for each layer.

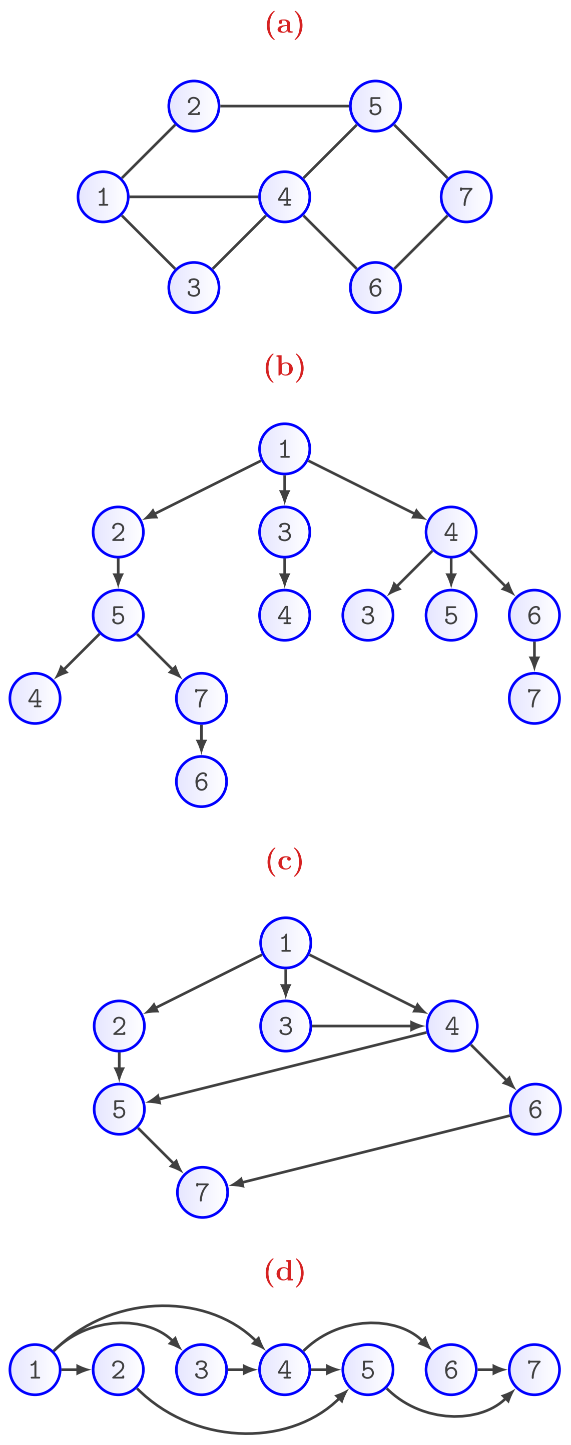


Figure 4.10: Part **(a)** of the figure shows an example graph, and **(b)** an according search tree generated by PBFS starting from node `1` with $\alpha = 1$. **(c)** illustrates how this search tree is adjusted in order to create an RSS, and **(d)** depicts a topological ordering of the same.

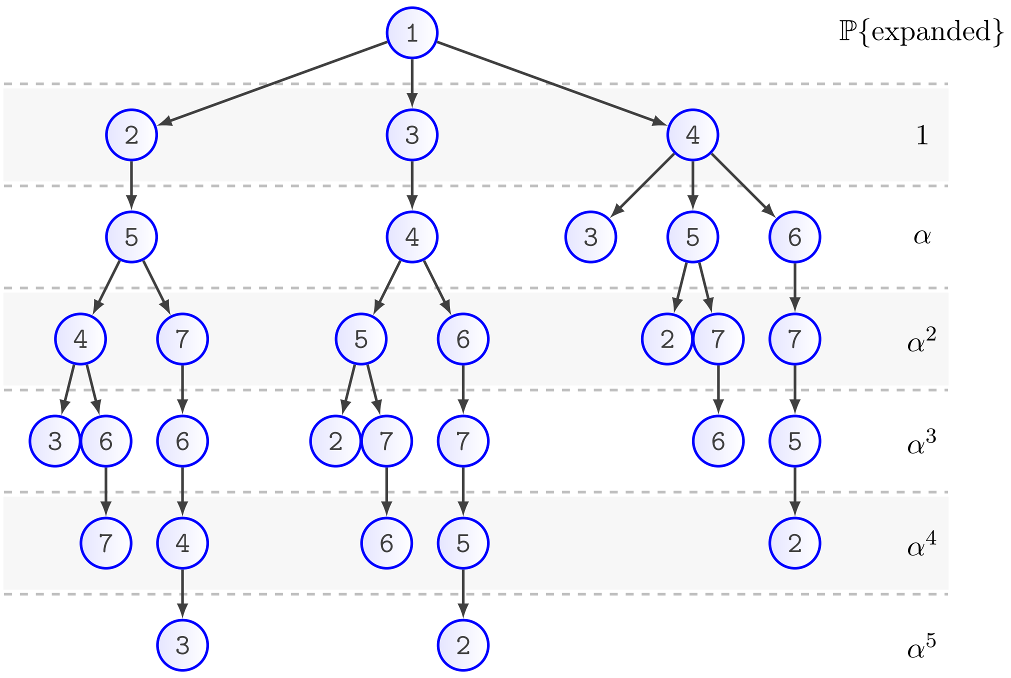


Figure 4.11: This is a tree that contains all possible branches that a search tree generated by PBFS starting from node 1 in the graph in Figure 4.10a might have. Furthermore, each layer of the tree is annotated with the probability that the single nodes it contains get actually expanded, provided that they are not skipped for other reasons. As usual, $0 < \alpha \leq 1$.

Random Simple Subgraphs

Next, we want to define what I refer to as *random simple subgraphs* (RSSs). To that end, let's suppose that we are given an arbitrary, perhaps directed multigraph $G = (V, E)$ and some distinguished vertex $v \in V$. Our goal is to extract some kind of neighborhood of v where we disregard the direction of the edges in G, if G is directed at all, and we are going to call it RSS of G around v, denoted as $\text{RSS}(G, v)$. Our main concern are the vertices in such a neighborhood, and so we will not pay any special attention to multiple edges. However, we also wish to consider an RSS as a blueprint of how to process the vertices it contains. Therefore, we require any such graph to be directed and acyclic. This, in turn, implies that the vertices of an RSS can be ordered topologically, and thus imposes a certain order among them.

$\text{RSS}(G, v)$ is induced as follows:

- We start by replacing all directed edges with undirected ones, and if there are multiple edges between any pair of vertices, then we remove all of them but one. Therefore, the resulting graph is simple and undirected.

- Now we run PBFS starting from vertex v, and retrieve a directed search tree T.
- However, T is certainly not a subgraph of G, and so we have to make a few adjustments. To that end, we examine all of the vertices in T in the order that they were generated in the course of the PBFS:
  - Let u be the currently investigated vertex, then we check whether a copy of u has been generated before.
  - If this is the case, then we remove u from the tree, and connect its parent in T, call it $\text{parent}(u)$, to u's previously generated copy instead. Thereby, $\text{parent}(u)$ is the source of this newly added arc. However, with this new connection we might have introduced a path in T that contains the same vertex twice. Therefore, we check whether there is a directed path from $\text{parent}(u)$ to a copy of itself, and if this is indeed the case, then we drop the latter vertex including all of its descendants.
  - Otherwise, we leave vertex u untouched.

It is easy to see that if we disregard the directions of the edges in G and coalesce multiple edges to single ones, then $\text{RSS}(G, v)$ is indeed a subgraph of G. Figure 4.10c demonstrates how the search tree given in Figure 4.10b is adjusted in order to turn it into a valid RSS.[1] Moreover, Figure 4.10d depicts a certain topological ordering of this very RSS.

4.6.2 Model Architecture

Now we are ready to define our model, and to that end, we pursue the following approach. As a first step, we introduce a NN model that comes with a nice theoretical justification, but seems fairly impractical due to a high computational burden. Hence, as a second step, we introduce a certain shortcut. This way we loose the justification, but arrive at a model that works well in practice.

Throughout this section, we assume that we are given some training graph $\mathcal{T} = (V, E, f, g)$ that is part of an arbitrary feature graph $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \mathcal{X}, \mathcal{Y}, \mathfrak{f}, \mathfrak{g})$, i.e. $\mathcal{T} \leq \mathcal{G}$.

The Full-Fledged Model

To start with, let's consider a few basic thoughts that our model is built upon:

- The whole approach is based on vector space semantics, i.e. we belief that there is a way to represent the individuals in $\mathcal{G}$ in terms of vectors from some

[1] Notice that, since we used $\alpha = 1$ to generate this search tree, the adjusted graph in 4.10c resembles the original one in 4.10a exactly, i.e. apart from the edges' directions it is just another embedding of the same graph. Nevertheless, it demonstrates how a search tree is manipulated nicely.

(part of an) Euclidean space. Our main objective is thus to find a mapping that associates the elements in $\mathcal{V}$ with suitable vectors given the data in $\mathcal{T}$.

- All of the information that we might have about a single individual $x \in \mathcal{V}$ is divided into its feature vector $\mathfrak{f}(x)$ on the one hand and those edges in $\mathcal{E}$ that include x on the other hand. However, in contrast to the latter, $\mathfrak{f}(x)$ does not depend on any other elements in $\mathcal{V}$. Therefore, it seems like a reasonable approach to use it as initial vector representation of x, and accommodate it incrementally based on the relations that x is involved in.
- Incorporating the information hidden in $\mathcal{E}$ into the individuals' vector representations is not straightforward, though, since an update of a single vector might render further adjustments of related individuals necessary. Therefore, it seems natural to make use of a Markov chain for computing these representations. In doing so, one could randomly pick an individual in $\mathcal{V}$, choose any of the relations it is involved in, and update its current representation based on the related individual as well as the type of the chosen edge. This procedure is repeated until the Markov chain converges to its limiting distribution, and eventually we arrive at good vector representations for all of the elements in $\mathcal{V}$.
- However, updating vector representations appropriately based on single edges in $\mathcal{E}$ is certainly not a trivial task either, and so it may be promising to apply ML in order to learn how to achieve this at best. Since we are working on graphs, we are going to make use of RNTNs for this purpose. Thereby, we need two sets of parameters for each kind of relation in $\mathcal{E}$, one to update the source of an edge and another one to update the target.
- As soon as we generated vector representations for the individuals, we can use our traditional ML machinery to compute the desired predictions based on the individuals' vectors. This means that we train some kind of ML algorithm to map the vector representation of an individual $x \in \mathcal{V}$ to an according prediction of $\mathfrak{g}(x)$. However, even though one could basically employ any kind of ML strategy, we stick to logistic regression in this work. Therefore, one could consider our model as a very deep RNTN with an additional feed-forward layer, that contains a single logistic unit only, on top.

Now that we have an idea of how to proceed in general, we want to devise an algorithm that describes these thoughts formally. However, in order to make the subsequent treatment a little easier, let's have a look at the single constituents of our model first:

- Like suggested by the introductory considerations, one could subdivide our approach into two parts: a mapping that associates elements from $\mathcal{V}$ with real vectors and another function that computes predictions for individuals from those vector representations. So, if we consider our model as a function h, then we could describe it as

$$h : \begin{cases} \mathcal{V} & \rightarrow & \mathcal{Y} \\ x & \mapsto & (h_2 \circ h_1)(x) \end{cases}$$

where $h_1 : \mathcal{V} \rightarrow \mathbb{R}^n$ is the representation mapping and $h_2 : \mathbb{R}^n \rightarrow \mathcal{Y}$ a prediction function. Notice that n is a hyperparameter that needs to be specified during the training of the model.

- As stated already, h_2 is a simple 1-layer NN with a single logistic output unit, which corresponds to logistic regression, and thus does not require any further discussion.
- However, in contrast to this, h_1 is really not a trivial operation, and comprises a number of steps in order to compute a vector representation for some individual. To that end, we define h_1 in terms of an RNTN, and if $\mathcal{T}$ contains k different types of relations, then we need a total of $2k$ sets of parameters for this RNTN—two for each kind of relation in $\mathcal{T}$. In this context, we will denote the two different relations—from the RNTN's point of view—that indicate the parameters to use in a tensor layer for a certain kind of relation p as $p \triangleright$ and $\triangleleft\, p$, respectively.

Next, we want to investigate the mapping h_1 in detail, and for this purpose, let's suppose that we need to compute a vector representation for a yet unknown individual $z \in (\mathcal{V} \setminus V)$. However, in order to accomplish this task, we need a little more information about z, and the bare minimum is its feature vector $\mathbf{f}_z = \mathfrak{f}(z)$. Moreover, it is helpful to know (some of) the edges in $\mathcal{E}$ that z is involved in, and we will denote the provided information as a set of triples $F_z \subseteq \{(s, p, o) \mid (s, o) \in \mathcal{E}_p \in \mathcal{E} \wedge z \in \{s, o\}\}$. From here we proceed as follows:

1. We start by incorporating the information about z into our training graph $\mathcal{T}$, and denote the result as $\hat{\mathcal{T}} = \mathcal{T} + (z, \mathbf{f}_z, F_z)$. However, in doing so, we have to beware of a few pitfalls:
   - Some of the edges in F_z might be of a yet unknown type, i.e. a type of relation that is present in $\mathcal{G}$, but not encountered in $\mathcal{T}$ at all. However, the model was trained on $\mathcal{T}$, and thus we have no suitable set of parameters for the tensor layers of our RNTN to account for this type of edge. So, if $(s, p, o) \in F_z$ and $E \not\ni E_p \subseteq \mathcal{E}_p$, then the edge gets simply disregarded.
   - Just like this, edges in F_z may include vertices that are neither z nor part of V. Therefore, those individuals are not contained in the domain of the feature map f of our training graph $\mathcal{T}$, and so we have to disregard these edges as well. This means, if $(s, p, o) \in F_z$ and either $z \neq s \notin V$ or $z \neq p \notin V$, then we, once again, have to ignore the considered edge.

   Taking these aspects into account, it is straightforward to extend $\mathcal{T}$ accordingly.
2. Next, we define initial vector representations for all $x \in V_{\hat{\mathcal{T}}}$, denoted as $v_x^{(0)}$. However, as suggested at the beginning of this section already, we simply use the individuals' feature vectors for this, i.e. $v_x^{(0)} = f_{\hat{\mathcal{T}}}(x)$.

3. Now we arrived at the most important part of the algorithm which, technically speaking, simulates a Markov chain—I will say a few more words about this a little later. For this purpose, we randomly pick a vertex $x \in V_{\hat{\mathcal{T}}}$ together with an edge that it is involved in, e.g. (x, p, y). Subsequently, we use the appropriate layer of our RNTN in order to update the vector representation of x based on this edge. So, if $v_x^{(i)}$ and $v_y^{(j)}$ are the current vectors for x and y, respectively, then we define $v_x^{(i+1)} = g(v_x^{(i)}, p \triangleright, v_y^{(j)})$. Figure 4.12 illustrates an update like that. This simple procedure is repeated for a sufficient number of times, and the latest version of z's vector representation $v_z^{(i)}$ gets returned as the final result.

There are a few more things to say about this algorithm. First and foremost, it is easy to see that one could arrange the single update steps—which were expressed in terms of the function g computed by the layers of our RNTN—as a directed, acyclic graph, and thus h_1 can really be considered as the function computed by a single, very deep RNTN. Furthermore, we adopted the convention that the first of the arguments of a layer of this RNTN—this is the left input in any plot—is the one to be updated. The reason why we need two sets of parameters for the layers in the RNTN for every single relation in $\mathcal{T}$ is that we are facing directed edges, and it makes a difference whether the vector to be updated is the source or the target of such. Therefore, $p \triangleright$ and $\triangleleft\, p$ denote, like mentioned already, the RNTN-relations for updating the source and the target node of an edge of type p, respectively. Algorithm 4.1 summarizes the computation of the mapping function h_1 precisely.

Before we move on, I would like to say a few more words about the Markov chain used to compute vector representations. There exists indeed a nice formulation of a discrete-time Markov chain with a finite number of states that fits our particular scenario, and that converges to a limiting distribution if we make a few natural assumptions. However, from a practical point of view, the details are not of great interest, and so I am not going to elaborate on it in this work.

A Necessary Adaption of the Standard RNTN

The eagle-eyed reader might have noticed a certain flaw in the foregoing deliberations: the purpose that we used the RNTN for does not really fit its nature. In general, recursive NNs are used to compute vectors that represent graphs as a whole, as opposed to representations of individuals that are part of such. As a consequence, the functions computed by the layers in an RNTN do not reflect this asymmetry between one input that is supposed to be updated and another one which serves as information about how to do this only.

However, this deficiency can be remedied easily by using the following adapted version of the standard RNTN tensor layer as specified by Equation 2.6:

$$g(\mathbf{x}, R, \mathbf{y}) = \mathbf{x} + f\Big(\mathbf{x}^T \mathbf{W}_R^{[1:k]} \mathbf{y} + V_R \mathbf{y}\Big). \tag{4.1}$$

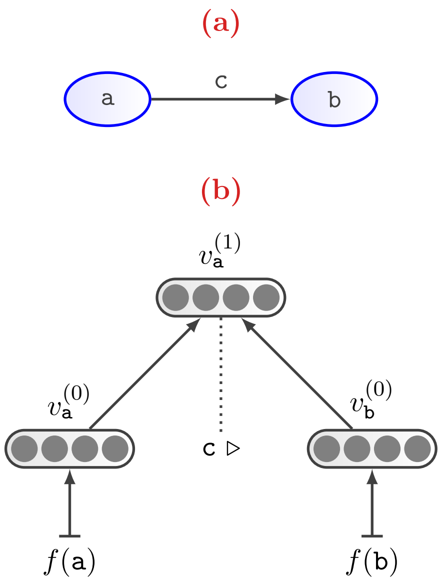


Figure 4.12: Part **(b)** of this figure illustrates how the vector representation of an individual `a` is updated based on the relation shown in part **(a)**. Thereby, we start from both individuals' initial vector representations. The left input to the network is the one to be updated, and `c` ▷ emphasizes that the used RNTN has been trained to update the source node of an edge of type `c`, i.e. the little arrow points from the source to the target.

Notice that the function f in this equation refers to a particular activation function rather than to the feature map of the training graph $\mathcal{T}$. This new version of g differs from the first one as follows:

- The input $\mathbf{x}$ is added to the output of the neuron *outside* of the activation function f. This change is basically dictated by the requirement that we seek to update this very vector.
- In return, $\mathbf{x}$ does not contribute to the argument of f independently of $\mathbf{y}$ anymore. This makes perfect sense, since $\mathbf{x}$ alone should not determine the way it is updated.
- Lastly, we removed the bias $\mathbf{b}_R$ from the activation of the layer. This is predicated on the fact that we do not wish to have some kind of »*default update*«.

Besides the fact that this change is in order with our requirements, it comes with the pleasant side effect of reducing the number of free parameters in the model. This, in turn, is of course advantageous with respect to the training of the same.

```
Input  : $x \in \mathcal{V}$,
         $\mathbf{f}_x = \mathfrak{f}(x)$,
         $F_x \subseteq \{(s,p,o) \mid (s,o) \in \mathcal{E}_p \in \mathcal{E} \wedge z \in \{s,o\}\}$
Output: $h_1(x)$, i.e. a vector representation of $x$
1  $\hat{\mathcal{T}} \leftarrow \mathcal{T} + (x, \mathbf{f}_x, F_x)$ ;        // $\hat{\mathcal{T}} = (V_{\hat{\mathcal{T}}}, E_{\hat{\mathcal{T}}}, f_{\hat{\mathcal{T}}}, g_{\hat{\mathcal{T}}})$
2  foreach $v \in V_{\hat{\mathcal{T}}}$ do
3      vec$[v] \leftarrow f_{\hat{\mathcal{T}}}(v)$ ;
4  end
5  while stopping criterion not met do
6      randomly choose a vertex $v \in V_{\hat{\mathcal{T}}}$ ;
7      randomly choose an edge $(r,s) \in E_p \in E_{\hat{\mathcal{T}}}$ that contains $v$ ;
8      if $v = r$ then
9          vec$[v] \leftarrow g($vec$[v], p \triangleright,$ vec$[s])$ ;
10     else
11         vec$[v] \leftarrow g($vec$[v], \triangleleft\, p,$ vec$[r])$ ;
12     end
13 end
14 return vec$[x]$ ;
```

Algorithm 4.1: This algorithm describes how to compute a vector representation of an individual in the full model.

A Practical Version of the Full Model

Although it is nicely justified, the full model comes with a negative connotation. In general, we do not know how long it takes until the states of a Markov chain actually show up according to its limiting distribution. Therefore, we might have to simulate it for quite some time, and if we are faced with a large training graph $\mathcal{T}$, then this might require a tremendous number of updates.

However, experience has shown that models that are based on Markov chains work well in practice even if we simulate the chain for a few steps only. Hence, we will also make use of a certain shortcut. Instead of randomly choosing vertices and updating them based on some incident edge, we execute two simple steps:

1. We begin by performing a PBFS starting from the vertex of interest, i.e. z. Thereby, we use $\alpha = 1$, but instead impose a maximum depth ℓ of the search tree, like we know it from depth-limited BFS. ℓ is yet another hyperparameter of the model, and is thus determined during the training phase.

```
Input : $x \in \mathcal{V}$,
        $\mathbf{f}_x = \mathfrak{f}(x)$,
        $F_x \subseteq \{(s, p, o) \mid (s, o) \in \mathcal{E}_p \in \mathcal{E} \wedge z \in \{s, o\}\}$
Output: $h_1(x)$, i.e. a vector representation of $x$
1 $\hat{\mathcal{T}} \leftarrow \mathcal{T} + (x, \mathbf{f}_x, F_x)$;
2 $S \leftarrow$ search tree of depth-limited PBFS with $\alpha = 1$ from $x$;
3 $R \leftarrow$ assemble RNTN from $S$;
4 return $R(\mathcal{T})$;
```

Algorithm 4.2: Computes vector representations for individuals in the practical model.

2. Now we take the resulting search tree, and translate it into an RNTN bottom-to-top. In doing so, we arrange the network to update the source of an edge in the search tree based on the target of the same.

This procedure is illustrated in Figure 4.13. Part (a) of the figure shows a simple example of a relational dataset, where `a` is the individual that we are interested in, and (b) depicts the according, and notably unique, PBFS search tree. Figure 4.13c demonstrates the first step in the translation of the search tree into an RNTN, which is based on the edge that is highlighted in part (b). Now, there are two important aspects to note about this:

- In the search tree, `d` is the source of the considered edge, and thus it is the node that is updated by the RNTN, i.e. it is always the left input in Figure 4.13c.
- Furthermore, we see that there are *two* edges between `d` and `e` in this dataset. These are considered one at a time—either in some predefined order or just randomly—, and so the RNTN updates `d`'s representation step-by-step.

As soon as we assembled the RNTN, we run it with the individuals' initial vectors as input, and finally retrieve a representation for z as output. The entire procedure is summarized in Algorithm 4.2.

4.6.3 Training

The training of our model is straightforward, since we can simply use the standard learning algorithm for RNTNs, i.e. tensor backprop through structure (Socher et al., 2013b). Therefore, it remains to clarify how we intend to map individuals in V to an according graph representation, which, in turn, can be converted into an RNTN in the course of the training. An obvious possibility is to copy the approach taken for

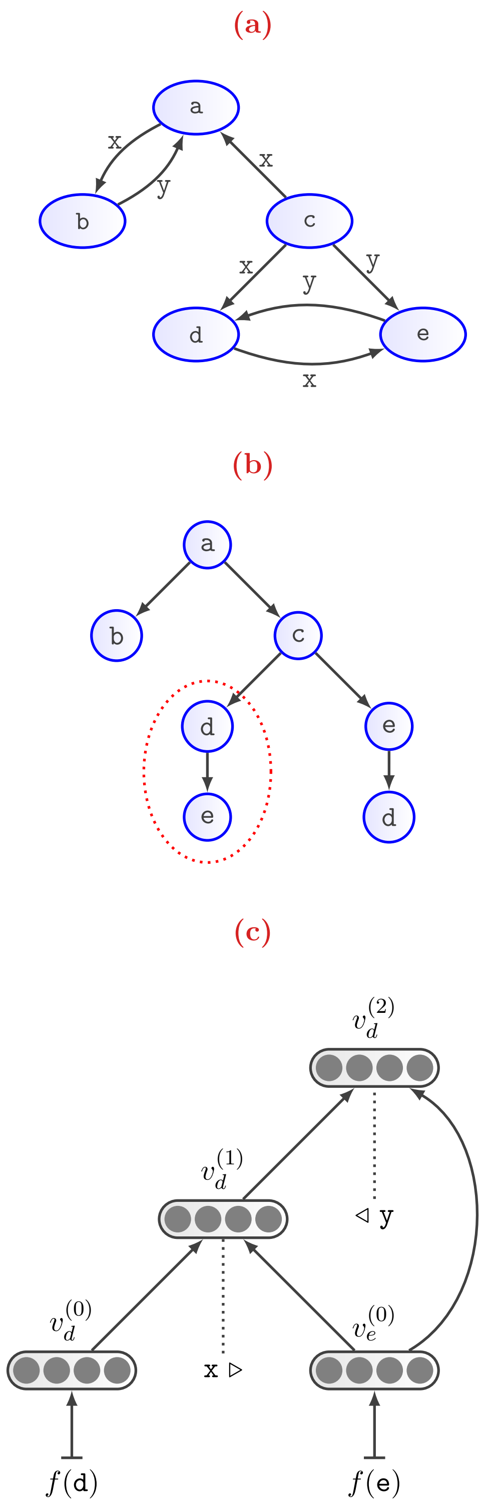


Figure 4.13: Part **(a)** shows a simple relational dataset, **(b)** a PBFS search tree starting from individual a (for $\alpha = 1$), and **(c)** a translation of the highlighted edge in **(b)** into an according RNTN.

Hyperparameters of the Model.

n	...	the dimension of the vector representations
k	...	the number of slices of the tensor used in the layers of the RNTN
ℓ	...	the depth-limit for computing representations
α	...	the discount factor used by PBFS for generating RSSs during the training

Table 4.4

generating representations in the practical model, and employ depth-limited PBFS (with $\alpha = 1$) in order to extract a certain search tree. It turns out, however, that there is a better way.

Every time that an individual $x \in V$ is used during the training phase of the model, we create a *new* graph representation for it by extracting an *RSS* of $\mathcal{T}$ around x, i.e. RSS($\mathcal{T}, x$). The intuition behind doing this is quite simple, and resembles, to some degree, the one underlying a state-of-the-art regularization strategy, the so-called *dropout* (Srivastava et al., 2014). The basic idea of dropout is that, for each training case of an NN, some of the neurons are randomly dropped, i.e. their outputs are basically fixed to 0. This strategy has been shown to have a strong regularizing effect, and can be used to train robust models that generalize well. Accordingly, whenever we create a new RSS for an individual x in the training graph, then we force the model to explore a new way of assembling x's vector representation by randomly dropping parts of the information about it. Thereby, the α used for extracting the RSS determines the amount of information that is getting dropped.

Besides that, the model contains a few hyperparameters that need to be specified before the actual training. These were discussed elsewhere already, and are summarized in Table 4.4.

4.7 Pretraining

Like discussed in Section 2.3.1, the recent hype around DL was caused by the insight that there are ways to pretrain models in order to achieve better generalization, and even though pretraining is frequently not used anymore, there are NN models that can actually benefit from it. A particular field of application that makes use of various pretraining strategies extensively is NLP, and since RNTNs were originally introduced in this context, it seems reasonable to consider pretraining for our model as well.

In this work, we generalized the use of RNTNs to arbitrary feature graphs, and the introduced model can thus be applied to a wide range of different datasets, including text—represented as graph, of course—but also flat data. Therefore, if we ask whether (some kind of) pretraining should be applied to our model, then the answer is *it depends on the particular domain*.

For applications to SW data, however, it seems like there is no such need. In the context of NLP, we start with *arbitrarily defined* vector representations of single words. These do not convey any kind of information in the first place, and so people try to employ pretraining in order to endow them with certain (vector space) semantics. However, if you consider the canonical translation for SW-KBs, then the initial representations of the individuals are predicated on their class memberships, and are thus clearly not arbitrary. Therefore, these vectors do convey a lot of information already, and so it seems questionable whether any kind pretraining is necessary at all. I tend to claim that the answer is *no*—which was also indicated by a few small experiments that I performed—, but a definite proposition requires additional research, and might be subject to future studies.

4.8 Characteristics of the Model

Before we close this chapter, let's discuss a few characteristics of this newly introduced model.

- As suggest at the end of Section 3.4, the model is not capable of computing predictions in constant time, since the computations to perform are dependent on the particular individual in question on the one hand and on the structure of the RDF graph that it belongs to on the other hand. However, we are able to establish an upper bound on the number of computation steps that are necessary for this purpose. If ℓ denotes the depth limit used to compute representations and d is the maximal vertex degree in the considered training graph, then it is easy to see that any prediction may be computed in $\mathcal{O}(d^{\ell})$.
- Furthermore, it is interesting to note how missing values act upon updates of vector representation as specified in Equation 4.1. As we chose to represent missing information about class memberships with a value of 0, any missing value is updated effectively if it is part of a left input to the RNTN, and basically disables the respective adjustment to perform if it belongs to a right input.
- Another nice property of the suggested model is that it allows us to make use of unlabeled vertices of a training graph as well, since these are used to extract PBFS search trees for both training and prediction.
- Furthermore, note that this new model is able to learn from small training graphs efficiently, which is due to the fact that we generate a new RSS whenever

an individual is selected as training instance. However, this advantage comes at the price of the increased computational burden that is implied by the constant generation of RSSs.

- Lastly, the suggested model exhibits one rather unpleasant property, namely that it needs the entire training graph for computing predictions. Anyhow, this issue can safely be ignored for many real-world applications, and becomes a serious problem for really large KBs only.

5

CHAPTER

Experiments

5.1 Experimental Setup

To evaluate the suggested approach, I experimented with two different datasets, and compared the performance of the new model—as suggested in Chapter 3 already—with the following state-of-the-art methods:

- SM-SVM with IST and WL kernel,
- KLR with IST and WL kernel,
- AKP, and
- RESCAL.

I implemented all of these techniques as well as my own approach in Python 3.5, and employed several commonly used frameworks from the SciPy family (Jones et al., 2001–), e.g. SciPy 0.18.0 and NumPy 1.11.1, for this purpose. Additionally, I used *scikit-learn* 0.17.1 (Pedregosa et al., 2011), a well-known ML framework that provides ready-to-use implementations of many different learning strategies together with various helper functions to develop such from scratch, for SM-SVM. KLR was implemented from scratch—based on the fast dual algorithm by Keerthi et al. (2005)— just like AKP, and both were trained by means of SciPy's optimization tools. To implement my own approach, I employed Google's *TensorFlow* 0.10.0rc0 (Abadi et al., 2015), a framework for numerical computations in general, but with a

Ontological KB	Individuals	Links	Classes	Relations
DBpedia (fragment)	16,606	6,627,878	251	133
AIFB Portal	20,756	48,324	57	79

Table 5.1: The characteristics of the datasets used in our experiments.

strong focus on ML and especially DL. For RESCAL, I made use of a nice implementation that is published on GitHub[1], and which is by Maximilian Nickel, who introduced the method as a co-author in Nickel et al. (2011).

In order to compute the canonical translations of the considered SW datasets, I developed a simple converter in Java 1.8. In doing so, I used Apache Jena 2.13.0[2], an open-source framework for handling SW data, together with the SW reasoner Pellet 2.4.0[3].

All experiments were computed on an Intel Core i7 CPU with 4×2.3 GHz and 16 GB of RAM. Furthermore, I did not make use of GPGPU for any of the experiments.

5.2 Description of the Experiments

In the context of the SW, there are a few concrete, and notably real-world, reasoning scenarios that have been used repeatedly for experimenting with ML algorithms. For this work, I decided to follow this very practice, and evaluated my model on the following tasks.

5.2.1 Experiment 1: Political Party Affiliations

The *DBpedia project*[4] (Bizer et al., 2009) maintains a SW-KB that provides access to structured data extracted from Wikipedia. However, this KB contains more than 3 billion triples in total, and is thus hard to work with for any kind of experiment. Therefore, following Nickel et al. (2011), Minervini et al. (2014a) extracted a fraction of DBpedia 3.9 that contains the presidents and vice presidents, respectively, of the

1 `https://github.com/mnick/rescal.py` (last visited on August 29, 2016)

2 `https://jena.apache.org` (last visited on August 29, 2016)

3 `https://github.com/Complexible/pellet` (last visited on August 29, 2016)

4 http://wiki.dbpedia.org/ (last visited on August 29, 2016)

US as well as all individuals that are immediate neighbors of them in the RDF graph of this dataset.[5]

Now, the question to be answered is which political party each of these presidents and vice presidents—76 in total—is affiliated with, the Democratic Party, the Republican Party, the Federalist Party, the Whig Party, or none of them. However, for this experiment, I confined myself to predicting whether any of them belongs the Democratic Party or not. In the original dataset, all of the parties were represented by according individuals, and thus this is actually a task of link prediction. Therefore, I removed all links that express an affiliation with a political party from the data, and introduced a new class with all individuals that are affiliated with the Democratic Party as its members instead. All presidents and vice presidents that have not been affiliated with the Democratic party were explicitly marked to not belong to this newly introduced class, and thus it could be used as the target to predict. This way, I turned the considered problem of link prediction into one of instance checking.

What is particularly interesting about this dataset is that it contains only 16,606 individuals, but 6,627,878 links of 133 different types—i.e. on average approximately 400 links per individual. Furthermore, it contains quite a high number of classes, namely 251 in total.

5.2.2 Experiment 2: Research Group Affiliations

For my second experiment, I used the *AIFB portal*[6] dataset. This data is provided by the Institute of Applied Informatics and Formal Description Methods—in German, this is abbreviated as AIFB—at the Karlsruhe Institute of Technology, and describes the organizational structure as well as the research activities of the institute.

Like Lösch et al. (2012), De Vries (2013), and Minervini et al. (2014a) did before, I considered the problem of predicting affiliations of employees with the research groups of the AIFB. As a matter of fact, I had a look at the group concerned with knowledge management—this is represented by the individual `aifb:Wissensmanagement`[7]—, in particular, and out of 501 in total, 113 employees are members of this group. Notice, however, that people might belong to multiple research groups at the same.

Since this is, once again, a problem of link prediction, I turned it into one of instance checking just like I did it for the first experiment. Furthermore, I did not employ any kind of reasoning for computing the canonical translation of this dataset.

[5] Minervini et al. made this dataset available at `https://code.google.com/archive/p/akp/` (last visited on August 29, 2016).

[6] `http://www.aifb.kit.edu/web/Web_Science_und_Wissensmanagement/Portal` (last visited on August 29, 2016); for scientific experiments, in particular, the websites provides a download of a static dump of the dataset

[7] The prefix `aifb` denotes the URL `http://www.aifb.kit.edu/id/`.

The reason for this is that I simply did not have appropriate hardware, that provides the resources necessary in order to perform reasoning with this dataset—given the fact that the considered dataset is rather small in comparison with other real world tasks, this emphasizes the need for methods based on ML one more time. It turned out, though, that this is not a major problem, since the data that we are interested in is specified explicitly. Anyhow, for the sake of the experiment, I employed default negation for the affiliation of employees with the research group of interest, as there would not have been any negative examples otherwise.

As opposed to DBpedia, the explicitly defined part of the AIFB portal is more like an average SW dataset. It contains 20,756 individuals, 48,324 links of 62 different types, and 57 different classes in total, and is thus by far not as dense as the fraction of DBpedia that was used for the first experiment.

5.3 Implementation Details

Now before we move on to consider the outcomes of the conducted experiments, I would like to discuss a few implementation details:

- Even though the respective datasets contain individuals that represent all kinds of things and persons, both of the considered learning tasks have a focus on particular resources only, namely presidents and vice presidents, respectively, for the first experiment and employees for the second one. Therefore, we have to confine the selection of training instances to these individuals of interest, but may use the whole dataset in constructing the according RSSs, kernels, and so forth.
- Just like this, the different approaches were, of course, only evaluated on these particular instances either.
- While the concept of a discount factor α seems reasonable for an average relational dataset, it turned out to be not quite appropriate for the fraction of DBpedia used for the first experiment. The reason for this is the high branching factor of the according RDF graph, which led to very broad PBFS search trees and, following this, really big RSSs. This, in turn, increased the duration of single training epochs significantly, and thus rendered the suggested approach impractical. However, in order to alleviate this issue, I introduced two hard limits for computing RSSs: first, I defined a maximum of 50 nodes for each level of a PBFS search tree, and second, I constrained the maximal branching factor of any such tree to 5. Interestingly, the same limits proved to work well for the seconds experiment too, and thus I imposed them in general.
- For the same reason, I fixed the parameters of the WL kernel to $d = 1$ and $h = 1$ for the first experiment.
- It turns out that many of the RDF classes encountered in SW datasets have a few members only, and thus it is hardly possible for a ML model to learn how

to deal with them properly. Therefore, for both of the experiments, I included only those classes in the training graph that at least 1% of the individuals belong to.

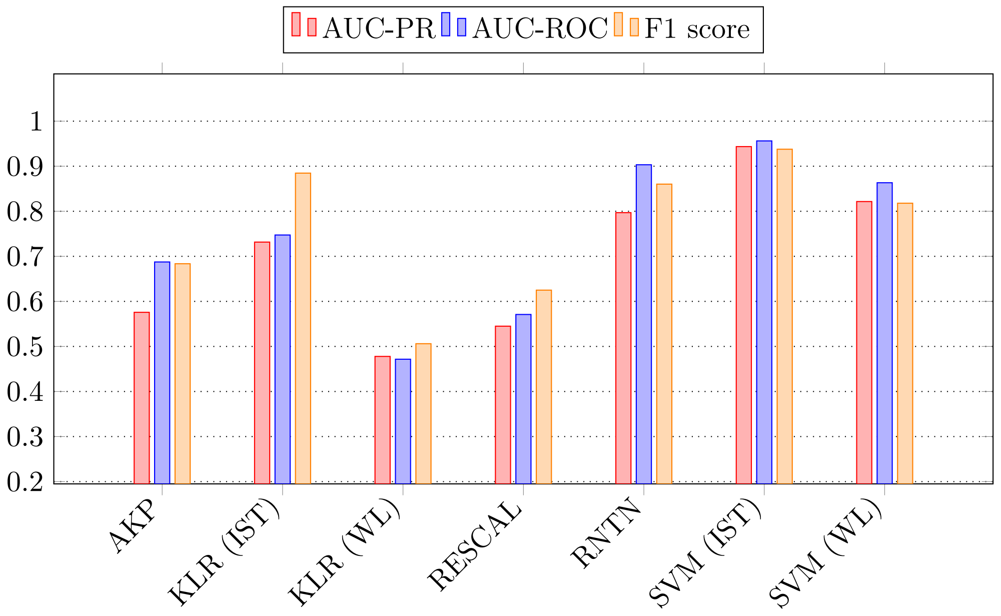
- Similarly, the considered datasets contain many different types of relations, but these do not occur at the same frequency. In fact, there are only a few relations that appear seemingly all the time while most of them are encountered sporadically only. Therefore, the rare kinds of relations do not appear in RSSs generated for training instances too frequently either, and thus it is necessary to remove those from the training data as well. A similar reasoning applies to AKP, which tries to weight different kinds of relations appropriately in order to propagate class labels. For my experiments with these two models, I thus disregarded all kinds of relations that less than 1% of the individuals are involved in at least once.
- Training an RNTN on such small datasets is tricky, since a model like this is highly sensitive to overfitting, and none of the usual guidelines or rules of thumb are applicable. It turned out, however, that this issue could be circumvented by making use of (very) early stopping together with reducing the learning rate more strongly and more quickly than usual. At first sight, this seems to strongly dissent with our intuition of NNs, yet it was found to be the most effective way to train the model in my experiments. Eventually, it turned out that the training—notably, for both experiments—worked best if I stopped after 200 epochs only and reduced the learning rate linearly from $5 \cdot 10^{-3}$ to $5 \cdot 10^{-7}$. Furthermore, the rather small batch size of 10 yielded the best results, and fixing the weight of an L2-regularization term to 10^{-8} prevented the adjustable parameters from diverging. The other hyperparameters were chosen as follows: n and k were set to the dimensions of the respective feature spaces, α was set to 0.5, and ℓ was chosen to be 3.
- AKP was trained by optimizing the leave-one-out error with the truncated Newton Conjugate-Gradient method. However, I did not make use of all unlabeled individuals, but considered those that are connected to at least three labeled instances only. The rationale behind this was to enforce the propagation of labels among training instances without increasing the computational burden too much, and indeed, I achieved better results using this setting.
- For the kernel methods, all of the hyperparameters were determined using 10-fold cross validation.

5.4 Results

For my experiments, I evaluated all of the considered learning algorithms by means of cross validation. However, while I used 10 splits to evaluate the kernel methods as well as RESCAL, I used 5-fold cross validation for AKP and three folds for the RNTNs. The reason for this was simply the limited availability of appropriate computing hardware as well as the varying training costs.

The results are reported in terms of the mean values—over all splits—of the *area under the precision-recall curve* (AUC-PR; Raghavan et al., 1989; Davis and Goadrich, 2006) and the *area under the ROC curve* (AUC-ROC; Davis and Goadrich, 2006). These two performance metrics seem to be just appropriate, since they yield meaningful measures in spite of the presence of unbalanced classes. Furthermore, for the sake of completeness, I reported the commonly used *F1 score*, even though it has to be treated with some caution. The subsequently presented results are discussed with respect to the prior two performance metrics if not stated differently.

5.4.1 Experiment 1: Political Party Affiliations



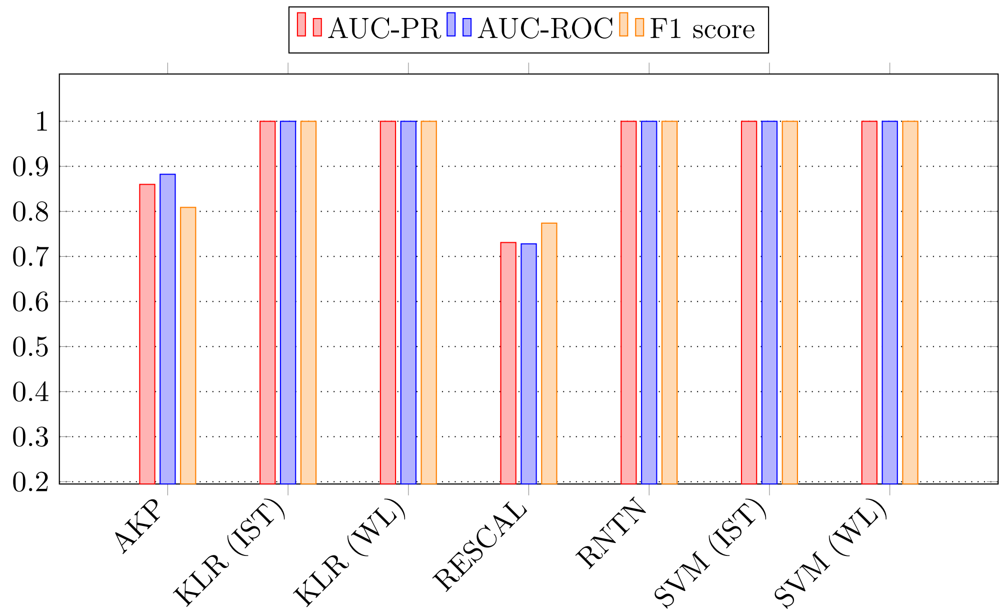
Method	AUC-PR	AUC-ROC	F1 score
AKP	0.576 ± 0.241	0.688 ± 0.182	0.684 ± 0.125
KLR (IST)	0.732 ± 0.208	0.747 ± 0.342	0.885 ± 0.090
KLR (WL)	0.478 ± 0.291	0.472 ± 0.039	0.506 ± 0.158
RESCAL	0.545 ± 0.266	0.571 ± 0.198	0.625 ± 0.238
RNTN	0.798 ± 0.196	0.903 ± 0.068	0.860 ± 0.061
SM-SVM (IST)	0.944 ± 0.078	0.956 ± 0.055	0.938 ± 0.081
SM-SVM (WL)	0.822 ± 0.120	0.863 ± 0.128	0.818 ± 0.123

(all values are denoted as mean ± standard deviation of all splits)

Method	Duration of Training & Evaluation
AKP	00:26:33
KLR (IST)	00:00:01 (+ 03:36:41)
KLR (WL)	00:00:01 (+ 00:21:59)
RESCAL	00:43:12
RNTN	01:46:52
SM-SVM (IST)	00:00:01 (+ 03:36:41)
SM-SVM (WL)	00:00:01 (+ 00:21:59)

(all values are denoted as hh:mm:ss; for kernel methods, the duration of generating the needed Gram matrices is denoted in parentheses)

5.4.2 Experiment 2: Research Group Affiliations



Method	AUC-PR	AUC-ROC	F1 score
AKP	0.860 ± 0.032	0.882 ± 0.051	0.809 ± 0.075
KLR (IST)	1.000 ± 0.000	1.000 ± 0.000	1.000 ± 0.000
KLR (WL)	1.000 ± 0.000	1.000 ± 0.000	1.000 ± 0.000
RESCAL	0.731 ± 0.101	0.728 ± 0.080	0.774 ± 0.091
RNTN	1.000 ± 0.000	1.000 ± 0.000	1.000 ± 0.000
SM-SVM (IST)	1.000 ± 0.000	1.000 ± 0.000	1.000 ± 0.000
SM-SVM (WL)	1.000 ± 0.000	1.000 ± 0.000	1.000 ± 0.000

(all values are denoted as mean ± standard deviation of all splits)

Method	Duration of Training & Evaluation
AKP	34:58:20
KLR (IST)	00:55:48 (+ 03:10:41)
KLR (WL)	00:02:08 (+ 07:51:02)
RESCAL	20:17:55
RNTN	01:52:51
SM-SVM (IST)	00:00:12 (+ 03:10:41)
SM-SVM (WL)	00:00:05 (+ 07:51:02)

(all values are denoted as hh:mm:ss; for kernel methods, the duration of generating the needed Gram matrices is denoted in parentheses)

5.5 Observations and Insights

First and foremost, we notice that the used RNTNs worked well for both of the experiments. However, the results also reveal a few rather surprisings aspects.

According to the retrieved measures, the first of the experiments was clearly the harder task. The SM-SVM with the IST kernel achieved the best score with respect to both metrics followed by our RNTN as well as the SM-SVM with WL kernel. What is kind of surprising about the outcomes of this experiment is that AKP, which has been the best-performing approach in other papers, could not assert itself, and achieved the fifth best score with respect to both measures only. Furthermore, it is interesting to see that the IST kernel outperformed the WL kernel for both of the considered kernel methods, as the latter one tends to work better in general. Also, we notice that KLR worked distinctly worse than SM-SVM, which is interesting as these methods achieve very similar results most of the time, and KLR with the WL kernel even performed worst among all of the treated methods. Lastly, it needs to be mentioned that RESCAL worked somewhat worse than reported in other published articles, but compared to the concretes outcomes of the other methods, it lived up to my expectations.

In contrast to the first one, the second experiment turned out to be rather simple—which I did not know beforehand—given that five of the considered learning techniques did a perfect job. However, it is still interesting to see that the introduced RNTN model cannot be »*tricked*« by trivial or toy problems, but is actually able to adapt to different situations and to learn how to make use of the data presented. Besides that, we can draw similar conclusions as for the first experiment, and it is especially interesting to notice that AKP was, once again, not among the best methods.

In conclusion, it can be said that, even though our model could not surpass all of the other approaches, the conducted experiments produced great results for two reasons:

1. While the suggested approach is really a general-purpose model for relational data, both the IST and the WL kernel were developed for SW data in particular. This means that they were tailored to account for peculiarities of typical RDF graphs, and thus have an advantage over any other approach that does not do so. However, this also means that our own model might still have some potential for improvement if we try to optimize it for SW data as well, but this is left open for future research.
2. Furthermore, it is remarkable that the RNTNs worked that well given that they were trained on two tiny datasets. In general, NNs often times require huge training sets—often comprising hundreds of thousands of training instances—in order to function well, and in my experiments I faced just 76 and 501 labeled individuals, respectively.

6 CHAPTER Conclusion and Future Work

6.1 Summary

For this thesis, I investigated the use of DL to tackle the problem of instance checking, i.e. predicting the membership of individuals in a SW-KB to a certain RDF class. For this purpose, we started with an informal review of the basic notions that the SW is built upon in Chapter 2, followed by an introduction of the basics of ML. Subsequently, we had a more detailed look at NNs, a family of ML models that are used for DL, and which constitute the state-of-the-art of ML in various fields of applications. At the end of this chapter, we introduced RNTNs, a powerful kind of NN that can be used to learn vector representations of graphs.

In Chapter 3, we discussed a number of issues that the original vision of the SW is attached with, and which are frequently encountered in practice. It turned out, however, that many of the addressed problems can be alleviated easily by replacing reasoning strategies rooted in classical logic with approaches based on ML. In fact, there are a number of published papers that are concerned with this very idea, and thus we investigated existing solutions for applying ML in the context of the SW. To that end, we reviewed several approaches based on kernel methods, transductive inference, and SRL.

To the best of our knowledge, there is no previous work on applying methods of DL in the context of the SW. Therefore, Chapter 4 presented a novel approach to instance checking based on RNTNs, which makes use of so-called vector space models of semantics. This means that we tried to create vector representations for individuals in a SW-KB that reflect their semantics as specified by the underlying ontology as well as the relations that they are involved in. To that end, we assembled representations incrementally, starting from simple incidence vectors that reflect all information about an individuals' class memberships in the KB. By means of a modified version of the standard RNTN model, these initial vectors were updated based on the links in the dataset that the individual of interest is involved in. Eventually, the final representation of an individual was used as input to a single-layer feed-forward NN that computed the desired prediction.

Finally, in Chapter 5, we had a look at two experiments that I conducted in order to evaluate the suggested model in comparison with other state-of-the-art methods. In doing so, I applied a number of learning strategies to two real-world reasoning tasks, and measured their performance in terms of AUC-PR and AUC-ROC values. The results of both experiments suggest that the new approach is indeed competitive, and even superior to most of the state-of-the-art techniques.

6.2 Possible Future Work

As suggested on a few occasions already, there are a number of possibilities to continue the research conducted for this thesis. Among these are both questions that address immediate improvements of the introduced model as well as such that are concerned with more comprehensive extensions of the same. Let's start with considering a few prospects of the prior type:

- Like mentioned in Chapter 5 already, the use of a discount factor α for PBFS turned out to be somewhat impractical for certain scenarios. Therefore, it would be interesting to investigate alternative ways to drop parts of PBFS search trees, and identify the situations that these are most suitable for. This is comparable with the research of different activation functions that are used in NNs for particular applications.
- If we are faced with an RDF graph that has a large branching factor—like the fraction of DBpedia used for the first experiment, e.g.—, then the computation of predictions may actually take quite some time, and might even turn out to be too costly for some applications. One expedient would be to make use of PBFS rather than BFS search trees for prediction as well, but this would lead to a probabilistic model. However, an alternative solution is to employ some kind of heuristic to select a subset of links to use for prediction only. This way, the model would be both efficient and deterministic. A heuristic like this could rely, once again, on ML or could simply make use of classical optimization strategies.

- As suggested in the last chapter, it could be interesting to investigate ways to adjust our model in order to improve its performance with respect to SW data in particular.
- Lastly, as mentioned in Section 4.7 already, it might be worthwhile to investigate various pretraining strategies.

Two more elaborate research paths are the following:

- First and foremost, the suggested model allows for a far-reaching generalization to arbitrary relational datasets, i.e. training graphs that contain n-ary rather than just binary relations. For this purpose, one might employ convolutional layers, like we know them from ConvNets, for the construction of networks based on RSSs in order to account for relations that include more than two individuals. The resulting model would be an actual general-purpose NN for relational data, and would thus provide us with a tool to deal with arbitrary datasets out-of-the-box.
- Another interesting idea is to extend the part of the model that computes vector representations to a relational version of the well-known deep autoencoder. This way, we might be able to map relational datasets to flat equivalents efficiently, irrespective of any concrete learning task.

List of Abbreviations

AKP	adaptive knowledge propagation
AUC-PR	area under the precision-recall curve
AUC-ROC	area under the ROC curve
BFS	breadth-first search
ConvNet	convolutional neural network
DL	deep learning
GPGPU	general purpose computing on graphical processing units
GPKP	Gaussian processes knowledge propagation
IST	intersection sub-tree
KB	knowledge base
KLR	kernel logistic regression
ML	machine learning
NLP	natural language processing
NN	neural network
OWL	web ontology language
PBFS	probabilistic breadth-first search
RDF	resource description framework
RDFS	RDF schema
ReLU	rectified linear unit
RNTN	recursive neural tensor network
ROC	receiver operating characteristic
RSS	random simple subgraph
SM-SVM	soft-margin support vector machine
SPARQL	SPARQL protocol and RDF query language
SRL	statistical relational learning
SUNS	statistical units node sets
SW	Semantic Web
URI	uniform resource identifier
URL	uniform resource locater
WL	Weisfeiler-Lehman
WWW	World Wide Web

Bibliography

Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems, 2015. URL `http://tensorflow.org/`. Software available from tensorflow.org.

Franz Baader, Diego Calvanese, Deborah L. McGuinness, Daniele Nardi, and Peter F. Patel-Schneider. *The Description Logic Handbook. Theory, Implementation and Applications.* Cambridge University Press, 2007.

David Beckett, Tim Berners-Lee, Eric Prud'hommeaux, and Gavin Carothers. RDF 1.1 Turtle. Technical report, W3C, February 2014.

Tim Berners-Lee, R. Fielding, U. C. Irvine, and L. Masinter. Uniform Resource Identifiers (URI): Generic Syntax. Technical report, IETF, August 1998. RFC 2396.

Tim Berners-Lee, James Hendler, and Ora Lassila. The Semantic Web. *Scientific American*, May 2001. URL `http://www.scientificamerican.com/article/the-semantic-web/`.

Christopher M. Bishop. *Pattern Recognition and Machine Learning.* Information Science and Statistics. Springer, 2006.

Christian Bizer, Jens Lehmann, Georgi Kobilarov, Sören Auer, Christian Becker, Richard Cyganiak, and Sebastian Hellmann. DBpedia - A crystallization point for the Web of Data. *Web Semantics: Science, Services and Agents on the World Wide Web*, 7(3):154–165, 2009.

Samuel R. Bowman. Can recursive neural tensor networks learn logical reasoning?, 2013.

Samuel R. Bowman, Christopher Potts, and Christopher D. Manning. Recursive Neural Networks Can Learn Logical Semantics, 2015.

Dan Brickley and R. V. Guha. RDF Schema 1.1. Technical report, W3C, February 2014. URL `https://www.w3.org/TR/rdf-schema/`.

Olivier Chapelle, Bernhard Schölkopf, and Alexander Zien. *Semi-Supervised Learning*. Adaptive Computation and Machine Learning. MIT Press, 2006.

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. Information Science and Statistics. MIT Press, 2009.

G. Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals and Systems*, 2(4):303–314, 1989.

Jesse Davis and Mark Goadrich. The Relationship Between Precision-Recall and ROC Curves. In *Proceedings of the 23rd International Conference on Machine Learning*, pages 233–240, 2006.

Gerben K. D. De Vries. A Fast Approximation of the Weisfeiler-Lehman Graph Kernel for RDF Data. In *Machine Learning and Knowledge Discovery in Databases: European Conference, ECML PKDD 2013, Prague, Czech Republic, September 23-27, 2013, Proceedings, Part I*, pages 606–621, 2013.

John Domingue, Dieter Fensel, and James A. Hendler, editors. *Handbook of Semantic Web Technologies*. Springer, 2011.

Brendan L. Douglas. The weisfeiler-lehman method and graph isomorphism testing, 2011.

Alexander Gammerman, Vladimir G. Vovk, and Vladimir N. Vapnik. Learning by transduction. In *Fourteenth conference on Uncertainty in artificial intelligence*, pages 148–155, 1998.

Lise Getoor and Ben Taskar. *Introduction to Statistical Relational Learning*. Adaptive Computation and Machine Learning. MIT Press, 2007.

Christoph Goller and Andreas Küchler. Learning Task-Dependent Distributed Representations by Backpropagation Through Structure. In *IEEE International Conference on Neural Networks*, volume 1, pages 347–352. IEEE, 1996.

Ian Goodfellow, Yoshua Bengio, and Aaron Courville. Deep learning. Book in preparation for MIT Press, 2016. URL `http://www.deeplearningbook.org`.

Geoffrey E. Hinton, Simon Osindero, and Yee-Whye Teh. A fast learning algorithm for deep belief nets. *Neural Computation*, 18(7):1527–1554, 2006.

Pascal Hitzler, Markus Krötzsch, Bijan Parsia, Peter F. Patel-Schneider, and Sebastian Rudolph. OWL 2 Web Ontology Language Primer. Technical report, W3C, December 2012. https://www.w3.org/TR/2012/REC-owl2-primer-20121211/.

Eric Jones, Travis Oliphant, Pearu Peterson, et al. SciPy: Open source scientific tools for Python, 2001–. URL `http://www.scipy.org/`.

S. S. Keerthi, K. B. Duan, S. K. Shevade, and A. N. Poo. A Fast Dual Algorithm for Kernel Logistic Regression. *Machine Learning*, 61(1-3):151–165, 2005.

Yan LeCun. Generalization and Network Design Strategies. Technical report, University of Toronto, 1989.

Uta Lösch, Stephan Bloehdorn, and Achim Rettinger. Graph Kernels for RDF Data. *The Semantic Web: Research and Applications*, 7295:134–148, 2012.

Frank Manola, Eric Miller, and Brian McBride. RDF 1.1 Primer. Technical report, W3C, February 2014. https://www.w3.org/TR/2014/NOTE-rdf11-primer-20140225/.

Pasquale Minervini, Claudia d'Amato, Nicola Fanizzi, and Floriana Esposito. Adaptive Knowledge Propagation in Web Ontologies. In Krzysztof Janowicz, Stefan Schlobach, Patrick Lambrix, and Eero Hyvönen, editors, *Knowledge Engineering and Knowledge Management*, volume 8876, pages 304–319. Springer International Publishing, 2014a.

Pasquale Minervini, Claudia d'Amato, Nicola Fanizzi, and Floriana Esposito. A Gaussian Process Model for Knowledge Propagation in Web Ontologies. *2014 IEEE International Conference on Data Mining*, pages 929–934, 2014b.

Tom M. Mitchell. *Machine Learning*. McGraw-Hill, 1997.

Maximilian Nickel, Volker Tresp, and Hans-Peter Kriegel. A Three-Way Model for Collective Learning on Multi-Relational Data. In *Proceedings of the 28th International Conference on Machine Learning*, pages 809–816, 2011.

F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

Nhathai Phan, Dejing Dou, Hao Wang, David Kil, and Brigitte Piniewski. Ontology-based Deep Learning for Human Behavior Prediction in Health Social Networks. In *Proceedings of the 6th ACM Conference on Bioinformatics, Computational Biology and Health Informatics*, pages 433–442, 2015.

Jordan B. Pollack. Recursive distributed representations. *Artificial Intelligence*, 46 (1):77–105, 1990.

Vijay Raghavan, Peter Bollmann, and Gwang S. Jung. A critical investigation of recall and precision as measures of retrieval system performance. *ACM Transactions on Information Systems*, 7(3):205–229, 1989.

Nathan D. Ratliff, J. Andrew Bagnell, and Martin A. Zinkevich. Subgradient Methods for Maximum Margin Structured Learning. In *ICML workshop on learning in structured output spaces*, volume 46, 2006.

Achim Rettinger, Uta Lösch, Volker Tresp, Claudia d'Amato, and Nicola Fanizzi. Mining the Semantic Web: Statistical Learning for Next Generation Knowledge Bases. *Data Mining and Knowledge Discovery*, 24(3):613–662, 2012.

David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning representations by back-propagating errors. *Nature*, 323(6088):533–536, 1986.

John Shawe-Taylor and Nello Cristianini. *Kernel Methods for Pattern Analysis.* Cambridge University Press, 2004.

Nino Shervashidze, Pascal Schweitzer, Erik J. van Leeuwen, Kurt Mehlhorn, and Karsten M. Borgwardt. Weisfeiler-Lehman graph kernels. *Journal of Machine Learning Research*, 12:2539–2561, 2011.

Richard Socher. *Recursive Deep Learning for Natural Language Processing and Computer Vision.* PhD thesis, Stanford University, 2014.

Richard Socher, Brody Huval, Christopher D. Manning, and Andrew Y. Ng. Semantic Compositionality through Recursive Matrix-Vector Spaces. In *Proceedings of the 2012 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning*, pages 1201–1211, 2012.

Richard Socher, Danqi Chen, Christopher D. Manning, and Andrew Y. Ng. Reasoning With Neural Tensor Networks for Knowledge Base Completion. In Christopher J. C. Burges, Leon Bottou, Max Welling, Zoubin Ghahramani, and Kilian Q. Weinberger, editors, *Advances in Neural Information Processing Systems 26*, pages 926–934. Curran Associates, Inc., 2013a.

Richard Socher, Alex Perelygin, Jean Y. Wu, Jason Chuang, Christopher D. Manning, Andrew Y. Ng, and Christopher Potts. Recursive deep models for semantic compositionality over a sentiment treebank. *Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1631–1642, 2013b.

Nitish Srivastava, Geoffrey E. Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout : A Simple Way to Prevent Neural Networks from Overfitting. *Journal of Machine Learning Research*, 15:1929–1958, 2014.

Volker Tresp, Yi Huang, Markus Bundschus, and Achim Rettinger. Materializing and Querying Learned Knowledge, 2009.

Peter D. Turney and Patrick Pantel. From frequency to meaning: Vector space models of semantics. *Journal of Artificial Intelligence Research*, 37:141–188, 2010.

Vladimir N. Vapnik. *Statistical Learning Theory.* Adaptive and Learning Systems for Signal Processing, Communications, and Control. Wiley-Interscience, 1998.

Boris Weisfeiler and A. A. Lehman. A reduction of a graph to a canonical form and an algebra arising during this reduction. *Nauchno-Technicheskaya Informatsia*, 2 (9):12–16, 1968.

Paul J. Werbos. Backpropagation Through Time: What It Does and How to Do It. *Proceedings of the IEEE*, 78(10):1550–1560, 1990.