Imperial College London
Department of Computing

Neural-Symbolic Learning for Knowledge Base Completion

Shuang Xia

Submitted in part fulfilment of the requirements for the degree of
Doctor of Philosophy in Computing of the Imperial College London and
the Diploma of Imperial College London, September 2022

Statement of Originality

I, Shuang Xia, declare that the work is my own and all else is appropriately referenced in the bibliography.

Copyright Declaration

Abstract

A query answering task computes the prediction scores of ground queries inferred from a Knowledge Base (*KB*). Traditional symbolic-based methods solve this task using 'exact' provers. However, they are not very scalable and difficult to apply to current large *KBs*. Sub-symbolic methods have recently been proposed to address this problem. They require to be trained to learn the semantics of the symbolic representation and use it to make predictions about query answering. Such predictions may rely upon unknown rules over the given *KB*. Not all proposed sub-symbolic systems are capable of inducing rules from the *KB*; and even more challenging is the learning of rules that are human interpretable. Some approaches, e.g., those based on a *Neural Theorem Prover* (*NTP*), are able to address this problem but with limited scalability and expressivity of the rules that they can induce.

We take inspiration from the *NTP* framework and propose three sub-symbolic architectures that solve the query answering task in a scalable manner while supporting the induction of more expressive rules. Two of these architectures, called *Topical NTP* (*TNTP*) and *Topic-Subdomain NTP* (*TSNTP*), address the scalability aspect. Trained representations of predicates and constants are clustered and the soft-unification of the backward chaining proof procedure that they use is controlled by these clusters. The third architecture, called *Negation-as-Failure TSNTP* (*NAF TSNTP*), addresses the expressivity of the induced rules by supporting the learning of rules with negation-as-failure. All these architectures make use of additional hyperparameters that encourage the learning of induced rules during training.

Each architecture is evaluated over benchmark datasets with increased complexity in size of the *KB*, number of predicates and constants present in the *KB*, and level of incompleteness of the *KB* with respect to test sets. The evaluation measures the accuracy of query answering prediction and computational time. The former uses two key metrics, *AUC_PR* and *HITS*, adopted also by existing sub-symbolic systems that solve the same task, whereas the computational time is in terms of CPU training time. The evaluation performance of our systems is compared against that of existing state-of-the-art sub-symbolic systems, showing that our approaches are indeed in most cases more accurate in solving query answering tasks, whilst being more efficient in computational time. The increased accuracy in some tasks is specifically due to the learning of more expressive rules, thus demonstrating the importance of increased expressivity in rule induction.

Acknowledgements

I would like to express my deepest gratitude to my co-supervisors Dr. Krysia Broda and Prof. Alessandra Russo. Thank you very much for your enlightenment on AI, enormous supports and encouragement during the PhD journey. Thanks for offering me the opportunities to keep challenging myself in this academic adventure, while always preparing a safe net in the background. Your enthusiasm, wisdom, dedication and rationality have strongly shaped my mind, both academically and in life.

I would like to thank my parents for their continuous supports on my study and life. I am lucky to still explore the different possibilities of life at this age, and it is not possible without your supports.

I would like to thank my friends and colleagues from the SPIKE group for your companion during this degree.

Table of contents

List of figures

List of tables

Chapter 1

Introduction

Neural-symbolic integration is a fast-growing field (see the survey by [d'Avila Garcez et al., 2019]), which aims to combine the strengths of neural networks and symbolic learning and reasoning whilst overcoming their limitations. Research advances in these two respective areas have led to novel neural-symbolic approaches and architectures (e.g. [d'Avila Garcez et al., 2002, Hammer and Hitzler, 2007, d'Avila Garcez et al., 2009, Yang et al., 2015, Guo et al., 2016, Serafini and d'Avila Garcez, 2016, Cohen et al., 2017, Yang et al., 2017, Rocktäschel and Riedel, 2017, Evans and Grefenstette, 2018, Das et al., 2018, Minervini et al., 2020a, Yang et al., 2020, Qu et al., 2020, Sen et al., 2022]).

Neural networks, and deep learning more generally, have the strength of recognising patterns from large amounts of data in a noise-tolerant way. Examples of successful achievements include the *AlphaFold* [Jumper et al., 2021] that constructs the 3D structure of a protein by recognising and combing its substructures. However, although these systems are good at spotting low-level patterns (such as recognising whether two items are similar), they lack the ability of performing higher-level reasoning. Moreover, they function as a blackbox and lack human interpretability. On the other hand, symbolic reasoning is particularly good at performing higher-level reasoning and learning high-level knowledge expressed as rules, thus making their reasoning human-interpretable. A symbolic system encodes information by symbols, organises this symbolic information in a structured representation (e.g. a knowledge base, introduced in the next paragraph), and uses these structured representations for systematic reasoning processing. Symbolic rule learning ([Muggleton, 1991]) combines learning and reasoning. Unlike neural networks (or sub-symbolic methods) where information are expressed by continuous values, symbolic learning uses discrete information ('True' or 'False') and can learn from small amounts of data. These differences between neural and symbolic reasoning and learning systems

make their integration a way to combine the best features of both.

Engaging with the strengths of the two fields, many neural-symbolic systems have been developed in recent years and their applications cover many fields [Wang and Yang, 2022], such as query answering, visual scene understanding, logical reasoning, robotics and control. For example, [Yang et al., 2015, Rocktäschel and Riedel, 2017, Qu et al., 2020] represent symbolic knowledge bases using embeddings and use embedding representations to capture relationships in knowledge bases for answering queries with good accuracy. Other applications combine image recognition and logical rules or constraints to better classify images and objects in images. Examples are the use of Logic Tensor Networks [Donadello et al., 2017] and [Yang et al., 2020, Manhaeve et al., 2018], which use a logic program to leverage classification of images, by learning a probability distribution over the features extracted from images by supervised learning of a downstream task. In yet other works such as [Tsamoura et al., 2021, Dai and Muggleton, 2021] features of images are expressed as symbolic terms and used for reasoning to improve the robustness of image detection tasks through abductive reasoning. Many of these neuro-symbolic systems are not scalable, but in [Aspis et al., 2022] a scalable approach is presented that first learns an approximate classifier using a downstream supervised task to learn a set of unlabelled clusters and then uses a logic program to fine-tune the classifier and label the clusters again via the downstream labels. [Zhu et al., 2020, Silver et al., 2022] encodes actions using symbolic logic and trains neural networks for predicating the next actions to take given a goal for planning, which both makes the decision making process more interpretable and supports higher-level logic reasoning.

This thesis contributes to the field of neural-symbolic integration. It proposes a neural-symbolic approach to tackle *query answering task*, namely to compute whether a ground query can be inferred from a knowledge base (KB). In symbolic reasoning, information in a KB, such as the ones we consider, is expressed as facts (or triples), of the form $relation(subject, object)$, where the relation is a *predicate* and the subject and object are *constants*. A KB may also include rules, which can be used to answer given queries. For instance, let a KB consist of the facts *is_cured_by(headache, aspirin)*, *has_disease(tom, headache)* and *eats(ben, apple)* and a Horn rule *eats(X, Y)* $\leftarrow$ *has_disease(X, Z), is_cured_by(Z, Y)*, where X, Y and Z are variables that can be instantiated with constants. The query *has_disease(tom, headache)* can be directly inferred from the facts in this KB, whereas the query *eats(tom, aspirin)* can be inferred from KB

only by using the rule together with the facts.[1] Traditional symbolic-based methods for solving such query answering tasks use 'exact' symbolic provers that rely on 'hard-unification'. That is, predicates and constants can only match with themselves. As a result of 'hard-unification', in the above example, if instead of the fact *is_cured_by(headache, aspirin)*, it had included the fact *is_cured_by(migraine, aspirin)*, the *KBs* would have not been able to prove the query *eats(tom, aspirin)*, even though *migraine* and *headache* are semantically similar. In this thesis, we solve the query answering task using soft-unification, which could allow the fact *is_cured_by(migraine, aspirin)* to 'soft-unify' with the body condition *is_cured_by(headache,aspirin)*. In contrast to 'True' or 'False', some neural network approaches use a score between 0 and 1 as a surrogate for whether a query is 'True' or 'False'. Although there are many systems tackling query answering tasks using neural network works (e.g. [Trouillon et al., 2016]), these systems usually act as blackboxes without human interpretability, unlike the human-interpretable rules shown above, so it is hard to understand how they make prediction decisions and whether these decisions are sensible. As a result, these systems are considered as unexplainable in the context of *Explainable AI* [Cyras et al., 2021].

Differentiable methods have recently been proposed to solve query answering tasks over large *KBs*, with the intent of integrating the strengths of neural networks and symbolic reasoning (see [Besold et al., 2017] for a survey). They require to be trained to learn the semantic representation of the symbols and be able to use such semantics to make predictions for query answering. For example, facts (e.g. $q(a, b)$) are represented as triples of high-dimensional vector embeddings (e.g. $[\boldsymbol{\theta}_q, \boldsymbol{\theta}_a, \boldsymbol{\theta}_b]$) where each vector, $\boldsymbol{\theta}_s$, embeds the trainable semantic representation of the symbols s presented in the facts. Trained embeddings intend therefore to capture the semantic meaning of the symbols. Embeddings that are close in vector space represent symbols that are semantically similar. A notion of soft-unification can be defined, for instance, in terms of Euclidean or Cosine distance between embeddings [Mikolov et al., 2013], and be used to perform reasoning and learning in a 'soft' manner. This provides the advantage of identifying connections between symbols that are similar in their semantic meaning. The relevant approaches range from earlier systems that learn neural representations without logical reasoning [Socher et al., 2013, Trouillon et al., 2016], to the more recent approaches that support logical deduction through rules that are given as input [Badreddine et al., 2022, Guo et al., 2016]. However, query answering predictions may sometimes rely upon unknown rules over the *KB*. Rule induction aims to learn rules that capture relations between facts in a knowledge base needed to answer

[1] We present a simple example here. More generally, in logic programming, but not in our systems, body atoms in a rule can be proved by using other rules, not just by using facts, as discussed in Section 2.2.

unknown queries. Approaches such as [Rocktäschel and Riedel, 2017, Minervini et al., 2018, 2020a, Qu et al., 2020] aim to address this problem.

The notion of inducing rules is closely related to inductive logic programming (*ILP*) [Muggleton, 1991]. Systems in the *ILP* domain focus on inducing rules (e.g. *Aleph* [Ashwin, 2007], *ILASP* [Law et al., 2014], *Metagol* [Muggleton et al., 2015]). Such systems take as inputs a logic program as background knowledge, a language bias that governs the structure of the learnable rules, and a set of facts that the learned rules ought to prove to be true or false. In most *ILP* systems, facts are often referred to as positive and negative examples respectively. These systems aim to induce human-interpretable rules as opposed to the so-called 'blackbox models' learnt by neural networks. These rules are induced by iteratively selecting a set of rules from all potential rules generated in the search space defined by the language bias, so that the selected rules prove as many positive facts and as few negative facts as possible. Apart from *ILP*, there are also other systems that aim at inducing rules. However, we prefer to call them 'rule mining systems' instead of 'rule induction systems', because these systems extract rules by selecting rules from a set of potential rules created from the signature of the *KB* according to some given language bias. For example, there is a field 'symbolic reasoning for knowledge base completion', which uses symbolic systems to mine rules. Different from *ILP*, these systems (e.g. *AMIE* [Galárraga et al., 2015], *AnyBURL* [Meilicke et al., 2019]) do not rely on negative examples, as knowledge bases might not contain negative examples. They develop heuristics to iteratively extend a selected set of rules with high scores according to heuristic functions while traversing the search space. There are also systems that use embeddings to mine rules ([Omran et al., 2018, Yang et al., 2015]), which uses embeddings to compute scores for evaluating potential rules. However, they do not learn rules and train embeddings at the same time, which means that rules are not learnt by training neural networks, but are selected from potential rules created using the signature of *KB* and given structures according to some embedding-based heuristic functions. In this thesis we focus on the subset of neurosymbolic reasoning that represents entities in knowledge bases by embeddings and learns rules at the same time as training embeddings, as rules induced through training might be more robust than rules selected by predefined heuristic functions.

Our systems induce first-order definite or normal rules, which are learned by learning vector representations. Our work was originally inspired by the *Neural Theorem Prover* (*NTP*) framework, first presented in [Rocktäschel and Riedel, 2017]. Although *NTP* supports similar or

more flexible rule syntax than other neural-symbolic rule induction systems, it can only induce definite first-order rules without negations, which restricts its expressiveness. We propose three new sub-symbolic architectures that solve query answering tasks in a scalable manner, while supporting the induction of expressive rules. Our architectures are capable of learning vector embedding representations for both predicates and constants in a given *KB*, and inducing rules from given template rules which impose a bias on the structure of learned rules. Our systems induce rules by constructing a computational tree for proving given queries and adopting a *logic programming-style* backward chaining reasoning strategy for performing logical inference. During the backward chaining, soft-unification is used to match symbols. During training, embeddings are updated by minimising the difference between the proof score generated by the computational tree, and the target label of each given query. The underlying principle, used by our architectures, of performing logical deduction through soft-unification and backward chaining over embedding representations, is similar to that used in existing *NTP*-based systems. However, the latter have been shown to be not always scalable and limited in the expressivity and accuracy of the rules that they can induce. Our proposed architectures tackle these limitations. Specifically, our *Topical NTP* (*TNTP*) and *Topic-Subdomain NTP* (*TSNTP*) address the scalability aspect and they induce definite Horn rules efficiently. The third architecture, called *Negation-as-Failure TSNTP* (*NAF TSNTP*), addresses the expressivity of the induced rules by supporting the learning of normal rules with negation-as-failure. All these architectures make use of hyperparameters for encouraging the use of rules during training, leading to the induction of more accurate rules. Induced rules can be decoded to symbolic first-order rules for human-interpretation, so our systems are explainable [Cyras et al., 2021] and could be used to aid commonsense reasoning tasks (see a survey [Davis, 2017]) by inducing rules that capture implicit information explicitly. We describe the contributions made by each of our systems in more detail in what follows.

1.1 Contributions

Topical NTP. Our first and base system is *Topical NTP* (*TNTP*), which builds an efficient rule-induction neural theorem prover. It makes three contributions: 1) it provides an unsupervised topic generation method whereby a topic is a cluster of predicate symbols whose trained embeddings are close in vector space; 2) it makes use of *topical template rules* to bias rule induction by way of language bias; and 3) it makes use of two hyperparameters to favour the use of topical template rules during training, hence facilitating the induction of more accurate rules.

TNTP generates topics in an unsupervised way. It first learns the embedding representations of all predicates and constants that appear in the *KB*. This is done by means of a naive NTP mechanism called *FNTP* (i.e. *fact-only NTP*), where no rule induction is performed. Then, it makes use of unsupervised clustering over the trained predicate embeddings to generate *topics*, i.e. clusters of predicates that are semantically close in high-dimensional space. Using these topics, the facts that are included in a given *KB* are partitioned, by grouping together facts whose predicates belong to the same topic. Leveraging this notion of topics, template rules are generalised to *topical template rules* (*TTRs*) by specifying the topics that predicates in the induced rules should belong to. The use of topical template rules has two main advantages: it 'biases' the induction of a diverse set of rules, thus increasing the search space of the knowledge base, and it improves the computational performance during training by targeting the soft-unification of the conditions of template rules with facts that comply with the topic bias. To amplify the use of topical template rules during the training process, *TNTP* makes use of two hyperparameters. These have the effect of increasing the chance of using template rules during training where reasoning is performed over the computational tree. As a result, topic-related induced predicates receive gradients more frequently, resulting in more accurate induced rules than those induced without use of these hyperparameters. The *TNTP* architecture and related evaluation results are published in [Xia et al., 2020].

Topic-Subdomain NTP. To improve the computational efficiency of *TNTP* even further, in our second system, *Topic-Subdomain NTP (TSNTP)*, the notion of clustering is applied to constants as well as predicates. Trained representations of predicates and constants are clustered to *topics* and *subdomains* by an unsupervised clustering algorithm and the soft-unification of the backward chaining proof procedure is controlled by these clusters. Although the first step of learning embedding representations of constant symbols is similar to that used in *TNTP*, the clustering of constants determines a further partition over the *KB* by grouping facts that have predicates belonging to the same topic and their first argument belonging to the same subdomain. This reduces even more the size of the computational tree without affecting the semantics of a logical derivation, as soft-unification is computed among conditions of template rules and facts that are even more semantically related, i.e. facts that match both topics and subdomains. Only the subdomain of the first argument is used to allow a wide selection of relevant facts when proving each body literals. If the topic of the predicate and two subdomains of both arguments were used to partition the knowledge base, each partition would contain

too few facts. We discuss this in detail in Section 4.1.1. Our evaluation results show that *TSNTP* maintains similar (or even higher) accuracy in solving query answering prediction tasks compared with *TNTP*, while reducing the computation time by 50%.

Negation-as-Failure TSNTP. Whereas the focus of the above two architectures is to provide a scalable and computationally efficient neural-symbolic approach for addressing query answering tasks over large *KBs*, the expressivity of the rules that *TNTP* and *TSNTP* can induce is limited to definite rules. Given the incompleteness of *KBs* and the 'open world assumption' that underpins these *KBs*, induced definite rules may be over-general, causing a high number of false positive predictions and therefore low precision. Increasing the expressivity of the rules that can be induced might be possible to limit this issue. Our third system, *Negation-as-Failure TSNTP* (*NAF TSNTP*), aims to do this and introduces two novel features, namely a new scoring function for matching negative body literals and a method to select the best normal clauses arising out of refinements of definite clauses. To learn normal rules, *NAF TSNTP* first induces a set of definite rules and then uses a normal rule selection mechanism to convert (some of) these rules into potential normal rules according to their impact on the accuracy of the query answering task over the training set. Those normal rules that reduce the number of false positive predictions are then accepted as induced normal rules. Regarding the scoring function for negative literals, notice that soft-unification introduces a symmetry between proving a positive or negative literal. In case of a ground positive atom, say $p(a, b)$, the match is the fact with the best score selected from all facts having a predicate in the same topic as p and subdomain of the first argument the same as the subdomain of a. On the other hand, in case of a ground negative body literal, *not* $p(a, b)$, the score is $1 - score(p(a, b))$ where $score(p(a, b))$ is the best score that matches $p(a, b)$ and all facts having a predicate in the same topic as p and subdomain of the first argument the same as the subdomain of a. (Note that, as previously mentioned, our systems are restricted so rule chaining is not allowed and body literals can only be proved using facts.) These restrictions are justified and compared with the well-known deduction procedure used in logic programming, briefly described in Section 2.2, in Chapter 4 for positive atoms and Chapter 5 for negative atoms, respectively.

Evaluation. Each architecture is evaluated over benchmark datasets, namely *Countries* [Bouchard et al., 2015], *Nations*, *Kinship*, *UMLS* [McCray, 2003] and *FB122* [Guo et al., 2016]. These have increased complexity in terms of size of the *KB*, number of predicates and constants presented in the *KB*, and level of incompleteness of the *KB* with respect to their test sets. Our

evaluation measures the accuracy of query answering predictions and the computational time. The former uses two key metrics, *AUC_PR* and *HITS*, adopted also by existing sub-symbolic systems that solve the same task, whereas the computational time is in terms of CPU training time. We compare the accuracy in solving query answering tasks against that of existing state-of-the-art sub-symbolic systems [Rocktäschel and Riedel, 2017, Minervini et al., 2020a,b, Das et al., 2018, Qu et al., 2020, Trouillon et al., 2016], showing that our approaches are indeed in most cases more accurate whilst being over 30 times faster than *NTP*. Among the three systems, *NAF TSNTP* has the highest accuracy, due to its ability to induce normal rules that reduce the false positive predictions without affecting the true positive predictions, thus demonstrating the importance of increased expressivity in rule induction. The three systems can also decode induced rules to human-interpretable first-order rules, making the systems explainable.

1.2 Thesis Structure

The rest of the thesis is structured as follows. Chapter 2 introduces the notations and background information used throughout the thesis including a brief introduction of logic programming, neural networks, techniques for clustering and evaluation metrics. Chapter 3 presents the *Topical NTP* (*TNTP*), which uses topics to select relevant facts in a computational tree. Chapter 4 presents *TSNTP*, which uses topic and subdomain mechanisms for selecting relevant facts in a computational tree. Chapter 5 extends these two architectures with a new method, *NAF TSNTP*, for revising induced definite rules into induced normal rules. Chapter 6 discusses differences and similarities of our systems with respect to other existing approaches that are related to ours in that they solve query answering prediction tasks and/or perform rule induction. Chapter 7 evaluates our systems by describing the various experiment settings, providing results on accuracy and computational performance and comparing them to existing systems that are closely related and used as our baselines. Chapter 8 summarises the contributions of the thesis and briefly discusses some future research directions.

Chapter 2

Background

In this chapter, we cover various basic notations and concepts used throughout the thesis. We include relevant information about logic programming and artificial neural networks, followed by a brief introduction to methods used in our systems, such as clustering, the *AUC_PR* and *HITS* evaluation metrics.

2.1 Notations and Concepts Used in the Thesis

We firstly define the two inputs to our systems: the notations of knowledge bases (KB) and template rules (TRs). Then, we describe the embedding-based representations of symbols in KB and TRs, our notion of the learning task and the training regime.

We assume that a knowledge base is expressed using a given *Signature* $\Sigma = \mathcal{P} \cup \mathcal{C}$, where $\mathcal{P}$ is the set of binary predicates and $\mathcal{C}$ is the set of constants. A KB is then a set of positive facts, also called atoms or triples, of the form $p(c1, c2)$, where $p \in \mathcal{P}$ and $c1, c2 \in \mathcal{C}$. The constant $c1$ is referred to as the subject of the relation p, and $c2$ as the object of the relation p. We sometimes use $p(\bar{c})$ to denote an arbitrary ground atom where $\bar{c}$ represents the two constant arguments. Only binary atoms are considered in this thesis, because knowledge bases used in our evaluation of our works and other relevant works consist of binary atoms. Other arities could be supported in our work, but are outside the scope of this thesis.

Template rules (TRs) are structures of rules in terms of unknown predicate symbols which are disjoint from $\mathcal{P}$, and variables as arguments that after training will represent the learned rules (these rules are human-interpretable, see Section 3.5 for details). We refer to these unknown predicates as induced predicates. A TR has the form $h(\bar{V})$:- $b_1(\bar{V_1}) \dots b_n(\bar{V_n})$, $n \geq 1$, where h and

b_i are induced predicates, and $\bar{V}$ and $\bar{V_i}$ are pairs of variables. $h(\bar{V})$ is called the head atom and $b_i(\bar{V_i})$ is called a body atom of the rule. Variables in the head atom are distinct and every variable in a *TR* occurs in at least two atoms. Each *TR* is associated with a number m, indicating that there are m copies of the *TR* given as input. Each copy of the *TR* is independent, with unique induced predicates. For example, a *TR* with 3 copies is (3) $\#1(X,Y)$:- $\#2(X,Z), \#3(Z,Y)$. One of its copies could be $\#p1(X,Y)$:- $\#p2(X,Z), \#p3(Z,Y)$ and another copy could be $\#p4(X,Y)$:- $\#p5(X,Z), \#p6(Z,Y)$, where each $\#pn$ is an independent induced predicate to be learned. The set of *induced predicates* in the template rules is denoted by $\#\mathcal{P}$: $\{\#p1, \#p2, \#p3...\}$. The set of copies of *TRs* is denoted as $\mathcal{I}$. An *assignment* of a constant c to a variable V is denoted as V/c, and a *substitution* is a set of assignments of constants to variables, denoted as $d = \{V_i/c_i\}$. If $V_i/c_i \in d$, we say that V_i is in d. A specific assignment to a variable V_i in a given substitution d, is denoted as $d[V_i] = c_i$. All *TRs* are definite first-order rules with binary literals and no constant or function symbols. Normal rules are supported, but they are induced by refining definite rules induced from *TRs* by the addition of negated atoms. There is no restriction on the length of *TRs*, although a longer rule has a larger search space that increases the computational time.

In our differentiable setting, we assume that the symbols of our signature and the unknown predicates are represented as learnable *100*-dimension row vectors of real values, called embedding vectors. We specify the embedding dimension as 100 for comparison with other works, but our systems can employ other values if desired. The union of all embedding vectors is called the *embedding matrix*, denoted by $\boldsymbol{\theta}$, where $\boldsymbol{\theta} \in \mathbb{R}^{Z*100}$ and $Z = |\mathcal{P}| + |\#\mathcal{P}| + |\mathcal{C}|$. To simplify our notation, unless specified, we denote the embedding representation of a symbol x as $\boldsymbol{\theta}_x$ and denote the embeddings of known predicates, induced predicates and constants by $\boldsymbol{\theta}_p$, $\boldsymbol{\theta}_{\#p}$, and $\boldsymbol{\theta}_c$ respectively. A ground fact $p1(c1, c2)$ is represented by the tuple of embedding vectors $\langle \boldsymbol{\theta}_{p1}, \boldsymbol{\theta}_{c1}, \boldsymbol{\theta}_{c2} \rangle$.

Given a *KB*, the training dataset τ consists of facts in *KB*, called the *positive queries*, and corruptions of facts, called the *negative queries* or also *corrupted queries*. For each ground fact $q(\bar{c})$ in *KB*, a *corruption* of $q(\bar{c})$, denoted as $q(\hat{\bar{c}})$, is a ground fact constructed from $q(\bar{c})$ by changing one or more of its constants within the signature of the *KB*, so that $q(\hat{\bar{c}}) \notin KB$. For instance, given a ground fact $p(c_1, c_2)$ in *KB*, 4 corrupted facts could be generated by either changing the first argument $p(\hat{c_1}, c_2)$ (1 corruption) or by changing the second argument, $p(c_1, \hat{c_2})$ (1 corruption) or considering two changes for both arguments, so generating two different

$p(\hat{c_1}, \hat{c_2})$, such that $\hat{c_i} \in \mathcal{C}$ $(i \in \{1, 2\})$, where $\hat{c_i}$ denotes a corruption of constant c_i [Bordes et al., 2011]. Each epoch of training uses a different τ, with the same positive queries but different corruptions. Note that these corruptions are generated assuming a *closed-world assumption*, that is all atoms that are not in the knowledge base are 'false'.

We can now informally define the learning task of our systems. Given a knowledge base, with signature Σ, and a set $\mathcal{I}$ of copies of *TRs*, our learning task consists of learning the embedding matrix $\boldsymbol{\theta}$ of the symbols in $\Sigma \cup \mathcal{I}$, which minimises the errors in answering queries from a training set τ constructed from the knowledge base. These learned embeddings can then be used to answer unseen queries and to generate rules by decoding induced predicates to the nearest predicates (in vector space) in Σ.

2.2 Logic Programming

Logic Programming (see [Gabbay et al., 1994]) is traditionally used to represent *KBs* and make deductions from *KBs*. In what follows, we indicate the subset of the logic programming paradigm that is relevant to our works. In particular, throughout this section we make two assumptions: firstly that logic programs have a signature consisting of predicates and constants only, i.e. no function symbols, and secondly that the proof procedure is limited to matching an initial ground query with a fact or the head of a rule and does not allow chaining of rules. That is, a positive atom in the body of a rule cannot match with the head of another rule and it may only be matched with a fact. Both these assumptions apply to our systems.

A *logic program* consists of a set of facts and rules. An example with four facts and a rule is given below. A fact is a ground atom. A rule is ground if no variable appears as an argument of atoms in the rule. A non-ground rule represents the set of ground rule instances formed by substituting ground terms for its variables in all possible ways. For example, *eat(X, Y) :- has_disease(X, Z), cured_by(Z, Y)* is a non-ground rule and it is read as 'for all X and Y, $eat(X, Y)$ is true if $has\_disease(X, Z)$ and $cured\_by(Z, Y)$ is true'.

```
cured_by(headache, aspirin).
has_disease(tom, headache).
eat(ben, apple).
allergic(tom, apple).
eat(X, Y) :- has_disease(X, Z), cured_by(Z, Y).
```

Fig. 2.1 An example logic program with four facts and one rule, denoted as LP.

We consider two types of rules, definite rules and normal rules. A definite rule is of the form: h :- $b1, ..., bn$, where h and bi are binary atoms with variable arguments only. A normal rule is of the form h :- $b1, ..., bn$, *not* $e1, ...,$ *not* em, where *not* is negation-as-failure [Clark, 1978] and ej $(1 \leq j \leq m)$ is a binary atom whose arguments are variables that appear in h and $b1, ..., bn$. An atom h is proved true if the conjunction of $b1, ..., bn$ can be proved true and none of ej is true. The literal *not* ej is proved true if the atom ej cannot be proved true (called negation-as-failure).

In a *logic program* Π, a ground query can be proved by using a fact or a rule. In case of using a fact, the query must match exactly. This is called hard-unification: the predicates must be the same and each (constant) argument in the query must match with the corresponding (constant) argument in the fact. In the case of using a rule, the query should first unify with the head of the rule by hard-unification - that is the predicates of the query and rule head are identical and the variable arguments in the rule head are unified with the corresponding constant arguments in the query. The resulting substitution is then propagated to the body literals of the rule, which can be proved by using the facts in Π. We call this backward chaining. (Note that this is a restricted form of backward chaining as the body atoms can only match with facts. More generally, in logic programming the body atoms can be unified with the head of other rules.) For example, in our restricted backward chaining, given a query $eat(tom, aspirin)$ to the logic program LP in Figure 2.1, $eat(tom, aspirin)$ is true iff it can be proved by a fact or a rule. $eat(tom, aspirin)$ cannot be proved using any of the four facts directly. The query could be proved using the rule $eat(X, Y)$:- $has\_disease(X, Z), cured\_by(Z, Y)$, by backward chaining. The head atom $eat(X, Y)$ is matched to $eat(tom, aspirin)$, generating the substitution $\{X/tom, Y/aspirin\}$. The instantiated body atoms $has\_disease(tom, Z)$ and $cured\_by(Z, aspirin)$ are then in turn proved by using the second and the first fact in the KB respectively with substitution $\{Z/headache\}$. Using the rule, the logic program can return 'True' for the query $eat(tom, aspirin)$.

A logic program follows the negation-as-failure semantics [Clark, 1978], which assumes facts that cannot be deduced from a *KB* are proved to be 'False'. As a result, a logic program Π returns 'False' for a query if the query matches no facts in Π and there are no rules in Π that can prove the query. For example, the logic program *LP* (in Figure 2.1) returns 'False' for the queries *eat(tom, apple)* and *allergic(ben, aspirin)*. The negation-as-failure semantics also applies to negated atoms in normal rules, where a negated atom *not* $e(X, Y)$, for ground X and Y, is proved true by a logic program Π if $e(X, Y)$ does not hard-unify with any fact in Π.[1] In the logic program *LP*, assume the given rule to be replaced by *eat(X, Y) :- has_disease(X, Z), cured_by(Z, Y), not allergic (X, Y)* and consider the additional fact $allergic(tom, aspirin)$, then the query $eat(tom, aspirin)$ can no longer be proved.

Rule Induction. As shown in the previous example, using rules, an unknown query could be deduced. However, in *KBs* that are extensional, rules are usually unknown. Rule induction, in particular *inductive logic programming* (ILP) [Muggleton, 1991] (as introduced in Chapter 1), is a popular research area that aims at learning rules from given facts in a *KB*.

Recall that as introduced in Chapter 1, *ILP* systems use logic programming for inductive inference and thus rely on hard-unification. They generally allow the knowledge base to consist of both facts and rules and impose a language bias to restrict the form of induced rules. Our task is a little different in several aspects. Firstly, the *KB* does not usually include rules - it is generally a set of facts. Secondly, the *KB* may be noisy in the sense that different predicate and constant symbols are used to represent possible similar predicates or constants respectively, which enables our systems to perform a form of soft-unification, instead of hard-unification. Such soft-unification would allow, for instance, facts such as *parent_of(fred, mary)* and *parent(fred, alice)* to unify with a certain 'score', as they capture similar semantics. Thirdly, although our induced rules are constrained in their structure by *TRs*, which are somewhat similar to the templates of *Metagol* [Muggleton et al., 2015], the predicates in a *TR* are identified by embeddings rather than an exact predicate. This flexibility is also exploited in soft-unification.

[1] This again is the restriction to backward chaining - in general it is necessary to check if $e(X, Y)$ can be proved using facts and rules in Π.

2.3 Artificial Neural Networks

Inspired by biological nervous systems, artificial neural networks (*ANN*) [Rumelhart et al., 1988] have made much progress since early 2000s in learning representations, classifications and pattern recognition. In the following, we introduce the core concepts of *ANN* that are used in our neural architecture.

In an *ANN* classification task, the input is normally a training set, which contains pairs of data and target labels. For example, in a classification task, where, given attributes of a person, such as if the person like sports, etc.., represented by a Boolean value '0' or '1', the task is to classify if a person is a bicycle buyer or not. We can define a training set given by a vector of attributes for a person and a Boolean label '0' or '1'.

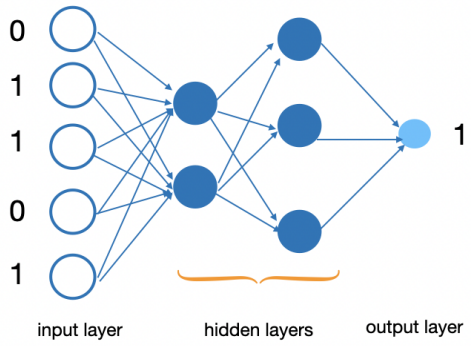

Fig. 2.2 A typical *ANN* architecture with an input layer, two hidden layers and an output layer, consisted of neurons represented as circles.

For example, imagine that the attributes of a person are represented by a vector of integers in $\{0, 1\}$, then we could consider an *ANN* architecture as depicted in Figure 2.2. The inputs, such as $[0, 1, 1, 0, 1]$, are given to the *ANN* through the input layer where each input value is represented by a 'neuron' (a circle in the graph). (Inputs could be other formats, such as embedding vectors or images.) The middle part includes 2 hidden layers, with 2 and 3 neurons respectively. The output layer has 1 neuron as this *ANN* is for binary classification. Figure 2.3 shows the internal details of a neuron. Each edge between element xi of the input vector $\mathbf{x}$ and the neuron has an associated weight wi, yielding a weight vector $\mathbf{w}$. The output of the neuron is computed by a function $\sigma(\mathbf{w^T x} + b)$, where the vector $\mathbf{w}^T$ is multiplied with the input vector $\mathbf{x}$ with an additional learnable bias value b. The σ represents a transfer function (such as a sigmoid function, where $\sigma(x) = \frac{1}{1+e^{-x}}$), which could convert the result of $\mathbf{w^T x} + b$ within a range (i.e. $(0, 1)$ for the sigmoid function). As shown in Figure 2.2, the outputs of a neuron become the inputs of neurons in the next layer. This is how values propagate in a neural networks.

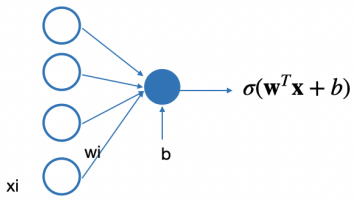

Fig. 2.3 A neuron in an *ANN* architecture with the input vector $\mathbf{x}$, weight vector $\mathbf{w}$, bias value b and the output value $\sigma(\mathbf{w^T x} + b)$.

The objective of the computational task is to train the architecture through gradient descent by updating the trainable parameters (i.e. weight vectors and biases), so that the error between predicted output and ground truth labels is minimised. This is therefore an optimisation problem: find the best parameters that lead to minimal prediction errors (by gradient descent). There are three steps in gradient descent, which are repeated for a number E of epochs over the training data. These are, loosely speaking, forward passing, computing loss and backward passing (backpropagation). We use mini-batch gradient descent, where labelled data are divided into batches and the three steps are applied to each batch in turn.

In forward passing, each batch is passed through the *ANN* and a prediction score for each element in the batch is computed, by computing the outputs of each neuron from left to right. Then, for each element, a loss value is computed by comparing the predicted score with its target label. An example of a basic loss function is given in Equation 2.1 [George Cybenko] and the training goal is to minimise L towards 0 (which is its minimum possible value). Note this is with respect to the full training set τ, although in practice, it is used over a batch.

$$L = \sum_{(q,y)\in\tau} -y\,log(o_q) - (1-y)\,log(1-o_q) \tag{2.1}$$

In Equation 2.1, q is a data point and y is its label, and o_q is the predicted score of q.

During training the value o_q is parameterised into the trainable parameters (e.g. w_i) of the network. So the partial derivative of the loss function with respect to each trainable parameter is computed for each element in a batch using the chain rule of differentiation. The value of each trainable parameter is updated using the average of its partial derivative values over the batch. Once the loss gets closer to zero, the *ANN* is considered to be trained and can be used at inference time on unseen data to make predictions.

2.4 Methods

In this section, we briefly introduce clustering methods over high-dimensional embeddings, as these are used in our approach. We also introduce the metrics that we use to evaluate the accuracy of our trained models.

2.4.1 Clustering Methods

Given a set of embeddings in vector space, they can be partitioned (or clustered), so that embeddings in the same partition are 'close' in the vector space. We introduce two methods for clustering embeddings, namely *K-means* [Macqueen, 1967] and *Agglomerative* [Gower and Ross, 1969]. Both methods are unsupervised approaches that could generate good clustering results.

The *K-Means* algorithm takes as inputs a number K of clusters and a set D of data in vector representation. The aim is to find the K centroid values for the clusters and assign each element in D to the nearest cluster, such that the overall distance between each element and the centroid of its assigned cluster is minimised. It is an iterative method which, after initialisation, performs two steps repeatedly at each iteration. Initialisation consists of randomly choosing K values as centroids of the K clusters. On each iteration, elements in D are assigned to the cluster whose centroid is closest in vector space, according to the least squared Euclidean distance. The centroids of all K clusters are then recomputed. These two steps are repeated until the cluster assignment does not change, or a given number of iterations is reached.

The *Agglomerative* clustering algorithm is a 'bottom-up' hierarchical approach. For a given K, initially, there are $|D|$ clusters, each element in D forming its own cluster. At each iteration, pairs of clusters are merged such that each cluster is merged with its closest cluster, according to the least squared Euclidean distance between the two centroids. This is continued until there are K clusters.

There are some differences between the two methods. Firstly, clustering results generated by *K-means* are not deterministic, because it uses an iterative refinement with random initialisation, whereas clustering results generated by *Agglomerative* algorithm are deterministic. Also, *K-means* needs a pre-specified cluster number before clustering, whereas *Agglomerative* can generate clustering results for all possible cluster numbers.

2.4.2 Evaluation Method

In this thesis, we use two evaluation metrics, namely *AUC_PR* and *HITS*. Before describing each metric, we present the basic concepts used in the metrics, in particular how to determine true/false positive and negative predictions and compute the *precision* and *recall* values under soft-unification.

Firstly, since we use soft-unification, a predication score *ps* could be any float between 0 and 1. So to classify whether *ps* is true or not, a *true atom threshold*, *ta_thres*, is introduced. For instance, a threshold of 0.6 means that if a prediction score $ps \geq 0.6$, the predicted output is considered as true; if $ps < 0.6$, the predicted output is considered as false.

Using this threshold, the precision and recall values can be computed based on the number of true/false positive and negative predictions in a knowledge base. Precision (Equation 2.2) measures the proportion of the predicted 'true' facts that are indeed true and recall (Equation 2.3) measures the proportion of positive facts that are correctly predicted (i.e. true positive rate).

$$precision = \frac{TP}{TP + FP} \tag{2.2}$$

$$recall = \frac{TP}{TP + FN} \tag{2.3}$$

where true positives TP (true negative TN) is the number of positive (negative) test queries in the knowledge base that are correctly classified as positive (negative) queries according to *ta_thres*. Similarly, false positives FP (false negatives FN) are the number of negative (positive) queries that are wrongly classified as positive (negative) queries.

According to Equation 2.2, precision is very sensitive to the number of false positive predictions, because this number is part of the denominator of the equation. If there are more false positive predictions than true positive predictions, the precision value would be low. Similarly, according to Equation 2.3, recall is very sensitive to the number of false negative predictions. This sensitivity makes them suitable to measure the performance of predictions, especially in a knowledge base with unbalanced number of positive and negative queries. For example, if a knowledge base has few positive queries and many negative queries, to achieve high precision and recall scores, it needs to keep both the number of false positive and false negative predictions low.

Note that since we use closed-world assumption, some queries that we assumed to be false (because they are not in the knowledge base) might, in fact, be true. In contrast, positive facts in a knowledge base are always true without noise. This may affect the calculation of precision and recall, which is based on counting true/false positive and negative predictions. However, since we do not have access to the complete knowledge base, using the closed-world assumption is an efficient way to generate negative queries. In exchange, we have to tolerate wrong predictions due to the incomplete knowledge base and therefore noisy labels to start with.

AUC_PR Using the definitions of precision and recall given above, *AUC_PR* is computed by constructing the precision-recall curve of a knowledge base and then measuring the area under the curve. The curve represents the precision values and recall values using different true atom thresholds *ta_thres*.

Given a specific *ta_thres*, its precision and recall values are computed (using Equation 2.2 and 2.3). They represent a dot on the PR curve, with recall as the x-axis and precision as the y-axis. Connecting precision and recall values over all thresholds gives us a PR curve. The area under this curve is computed as the final *AUC_PR* value, using the algorithm provided in [Davis and Goadrich, 2006]. The area under the curve can be also perceived as the average value of precision over all recall values. The larger the *AUC_PR* value, the better the performance. To reach a high *AUC_PR* value, a system needs high precision values over all recall values, which further means that it needs to maximise the number of true positive and true negative predictions.

HITS In the *HITS* metric, it is the relative difference between the proof score of a query and its corruptions that is important, not the absolute value of the proof score. To evaluate the difference, for each (binary) test query q, two lists of corrupted queries are created, namely $\hat{q_1}$ and $\hat{q_2}$, by corrupting either the first or the second argument of q, respectively, using all constants in the signature of the *KB*. Thus, $\hat{q_1}$ contains corrupted queries that have the same predicate and second argument as q, but different first arguments. Similarly, $\hat{q_2}$ contains corrupted queries that have the same predicate and first argument as q and different second arguments.

With $\hat{q_1}$ and $\hat{q_2}$, the score of q can be easily compared to that of its corrupted queries. To compute the *HITS* scores, two lists, $S_{\hat{q_1}}$ and $S_{\hat{q_2}}$, are created. $S_{\hat{q_1}}$ ($S_{\hat{q_2}}$) stores the scores of q, as well as those of all its corrupted queries in $\hat{q_1}$ ($\hat{q_2}$). These scores are sorted in a descending order. Note that multiple corrupted queries in $S_{\hat{q_i}}$ may have the same proof score as q. In this

case, the rank of q is one of the ranks of queries with the same score, decided by the order of their id values, as used in [Minervini et al., 2020a][2]. A different method can be used to select the rank of q. As in [Rocktäschel and Riedel, 2017] and our paper [Xia et al., 2020], the *HITS* measurement selects the lowest rank when multiple queries generate the same proof score. This inflates the *HITS* measurements, because if multiple corrupted queries have the same proof scores as the test query, the rank of the test query counts as the minimum rank of these queries. This measurement is particularly inaccurate when these scores are bad and is less fair than the one used here.

The *HITS* metric is further quantified by two indicators, namely *MRR* (Mean Reciprocal Rank) and *HITS@m*. *MRR* counts the average reciprocal rank (in the range $(0, 1]$) of the score of each test query q among scores of the corrupted lists, $S_{\hat{q_1}}$ and $S_{\hat{q_2}}$. *HITS@m* measures the percentage of test queries that are within the rank m when compared to their corruptions. The *MRR* and *HITS@m* of each query q and its corrupted query list $S_{\hat{q_i}}$ ($i \in \{1, 2\}$) are calculated according to Equation 2.4 and Equation 2.5.

$$MRR(q, S_{\hat{q_i}}) = \frac{1}{rank(S_{\hat{q_i}}, q)} \tag{2.4}$$

$$HITS@m(q, S_{\hat{q_i}}) = \left\{ \begin{array}{ll} 1 & \text{if } rank(S_{\hat{q_i}}, q) \leq m \\ 0 & \text{otherwise} \end{array} \right\} \tag{2.5}$$

The *MRR* of each test query q is the average of $MRR(q, S_{\hat{q_1}})$ and $MRR(q, S_{\hat{q_2}})$ and the MRR of the test set is the average *MRR* value over all test queries. The *HITS@m* value of the test set is calculated in the same way.

[2]There could be other ways to determine the ranks of queries with same proof scores, as used by other systems in Chapter 6. For a fair comparison, we use the same method as [Minervini et al., 2020a].

Chapter 3

Topical NTP

In this chapter we summarise the existing *Neural Theorem Prover* (*NTP*) approach, highlighting its major limitations. We then present our *Topical NTP* (*TNTP*) system, which addresses these limitations, and discuss in detail the characteristics of *TNTP* that enables increased scalability and accuracy of query answering predictions. We conclude the chapter with a method for extracting induced rules from a trained *TNTP* model in a form that is understandable to humans.

3.1 Overview of *NTP*

We introduce here the key concepts of the *NTP* framework [Rocktäschel and Riedel, 2017]. The task of *NTP* is to induce rules and train a set of embeddings, one for each constant and (known or induced) predicate. It does so by creating a computational tree, built from facts and copies of template rules, to answer queries. *NTP* takes as inputs a knowledge base consisting of a set of facts, and a set of template rules. The outputs of *NTP* are the set of trained embeddings and the induced rules which is learnt by learning embeddings of induced predicates in template rules. The only learnable parameter in *NTP* is the embedding matrix, which includes embeddings of known predicates and constants from the *KB* and induced predicates from template rules, and these embeddings are trained using gradient descent. In what follows, we describe the construction of the computational tree and the training process.

3.1.1 Knowledge Bases and Template Rules

The inputs of *NTP* consist of a symbolic knowledge base (*KB*) and a set of template rules. The *KB* has a set of positive binary facts. Examples are given below.

```
location_of(acquired_abnormality,experimental_model_of_disease).
manifestation_of(anatomical_abnormality,physiologic_function).
isa(alga,entity).
affects(mental_dysfunction,experimental_model_of_disease)
```

Template rules (*TRs*) have the form given below.

```
#1(X, Y) :- #2(X, Y).
#1(X, Y) :- #2(Y, X).
#1(X, Y) :- #2(X, Z), #3(Z, Y).
```

Predicates with the prefix # are induced predicates, each with an independent embedding. Each template rule i has an associated number mi, indicating the number of copies of i that are allowed to be used in the construction of the computational tree, e.g., (5) $\#1(X, Y)$:- $\#2(X, Y)$ indicates that there are 5 copies of the template rule of the form $\#1(X, Y)$:- $\#2(X, Y)$. Each copy has its own unique induced predicates. We indicate with $\mathcal{I}$ the set of copies of all template rules.

NTP learns the embedding representations of predicates and constants. We denote with $\mathcal{P}$ and $\mathcal{C}$ the set of known predicates and constants (respectively) used in the given *KB*, and with $\#\mathcal{P}$ the set of induced predicates in $\mathcal{I}$. The embedding matrix, $\boldsymbol{\theta}$, consists of an embedding for each symbol in $\mathcal{P} \cup \#\mathcal{P} \cup \mathcal{C}$, and has size $[|\mathcal{P}| + |\#\mathcal{P}| + |\mathcal{C}|, 100]$, where 100 is the size of each embedding. The embedding matrix is initialised randomly and it is the only trainable parameter in *NTP*.

3.1.2 Computational Tree Construction

The *NTP* approach uses a backward chaining reasoning mechanism to construct the computational tree used to classify the truth value of a ground query (a labelled ground atom from the training set), similar to that used in proving queries in logic programming [Gabbay et al., 1994] and described in Section 2.2. Unlike logic programming, which uses *symbol unification* (sometimes also called hard-unification), the backward chaining of *NTP* uses *subsymbolic soft-unification* (see Definition 3.1.1). This subsymbolic soft-unification employs the Radial Basis Function (RBF) formula $rbf(\boldsymbol{\theta}_1, \boldsymbol{\theta}_2) = exp(\frac{-||\boldsymbol{\theta}_1 - \boldsymbol{\theta}_2||_2}{2\mu^2})$ (μ is a hyperparameter that is set to $\frac{1}{\sqrt{2}}$ in experiments for the convenience of computation) for computing the similarity/distance between embedding vectors $\boldsymbol{\theta}_1$, $\boldsymbol{\theta}_2$.

Definition 3.1.1. Given an embedding matrix $\boldsymbol{\theta}$, a substitution set d, and two symbols i and j, which can be predicates, constants or variables, the unification between i and j, denoted as $uni\_s(i, j, d)$, generates a tuple (d_u, s) defined as follows, where d_u is the updated substitution set and s is the unification score.

$$(d_u, s) = \left\{ \begin{array}{ll} (d',\ 1), \text{where } d' = d \cup \{i/j\}) & \text{if is\_var}(i) \text{ and } i \notin d \\ (d,\ rbf(\boldsymbol{\theta}_c, \boldsymbol{\theta}_j)) & \text{if is\_var}(i) \text{ and } i/c \in d \\ (d,\ rbf(\boldsymbol{\theta}_i, \boldsymbol{\theta}_j)) & \text{otherwise} \end{array} \right\} \tag{3.1}$$

The unification of two atoms encoded as tuples of embedding representations of the predicate symbols and their respective (constant) arguments, is achieved by applying uni_s pairwise on the predicates and arguments. The unification score of two atoms $a1$ and, $a2$ with respect to a substitution set d is computed by the function $uni\_two\_atoms(a1, a2, d)$. This returns the minimum score among each pairwise predicate-predicate and corresponding argument-argument unification scores unified left to right (by Definition 3.1.1), and with any updated substitution set propagated through the argument unifications to give a final updated substitution set of the two atoms' unification. The unification score of two atoms is the minimum score among each pairwise predicate-predicate and argument-argument unification score, so that it can only have a high unification score if all of its predicates and constants are matched well.

Given a query q, the proof score of a head atom ha starts with an empty substitution set $\{\}$ and the proof is computed by $uni\_two\_atoms(q, ha, \{\})$ as defined above, which generates a proof score of the head atom with an outgoing substitution set. The outgoing substitution set is computed by propagating the substituting set from left to right when proving each uni_s computation on each pair of predicates or constants, as defined in Definition 3.1.1. The proof score of a body atom ba with an incoming substitution set d_b, denoted as $body\_atom\_proof\_score(ba, d_b)$, is the maximal proof score among the unification scores of $uni\_two\_atoms(ba, fi, d_b)$ for each fact fi in the KB. The maximal proof score is selected, because as long as the body atom can be proved by one fact, the atom can be considered as 'true'. The outgoing substitution set of the atom is passed on to prove the following atoms as their incoming substitution set.

Figure 3.1 illustrates an example NTP computational tree using soft-unification.

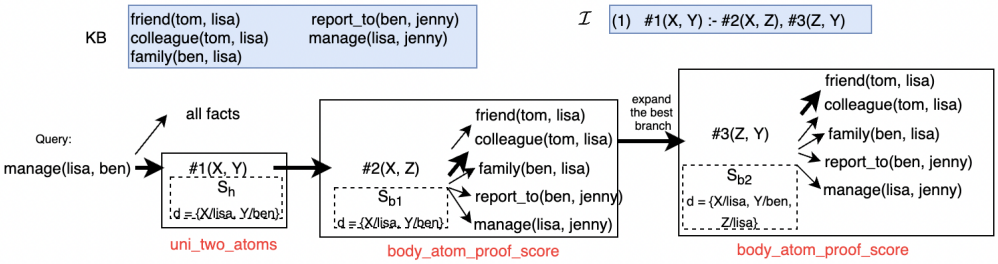

Fig. 3.1 An example of *NTP* computational tree given a knowledge base KB and a copy of a template rule in $\mathcal{I}$. In a computational tree, a query is unified with all facts in the KB (indicated by arrow from query to all facts) and all copies of template rules in $\mathcal{I}$ (here just one template rule copy and so one arrow). In the case of a fact branch, the query unifies with the fact using *uni_two_atoms*. In the case of a rule branch, the query firstly unifies with the head atom using *uni_two_atoms*, generating a substitution set for its variables, S_h. Then, it proves the first body atom using the substitution set S_{b1} ($S_{b1} = S_h$) according to *body_atom_proof_score*, matching with all facts in the KB, as represented by arrows. The fat arrow indicates the best match among all facts. For the simplicity of illustration, only the best-matched branch is extended to prove the following atoms, passing on its outgoing substitution set (S_{b2} in the figure). (Other branches can be extended in the same way, each with a different substitution set and to be proved independently with following body literals.) Using S_{b2}, the second body atom is proved in the same way as the first body atom, unifying with all facts. The fat arrows show the highest score branch that is used to generate the proof score for this query using this rule. Other rules and facts are proved using the same method, each generating a proof score. The highest proof score among these scores is the proof score of the query, which is used to compute the loss and update gradients as part of the training process as described in Section 3.1.3.

Figure 3.1 shows a computational tree for a KB composed of five facts and a template rule with one copy. The tree is used for evaluating queries in the same way in training and testing. On receiving a query, all possible soft-unification of the query with the facts in KB and all copies of template rules in $\mathcal{I}$ are considered. The arrow to all facts indicates a set of independent branches, each performing *uni_two_atoms* between the query and a fact (fi) in the KB using an empty substitution set, to give a proof score for unifying the query with the fact. In the case of a template rule (just one rule here), the query first unifies with the head atom, which applies *uni_two_atoms* between the query *manage(lisa,ben)* and the atom *#1(X,Y)*. This results in the substitution set of the head atom, $S_h : \{X/lisa, Y/ben\}$ (each substitution for the variable X and Y generates a unification score 1 according to *uni_s* as defined in Definition 3.1.1), and a score s_p, generated by unifying predicate embeddings $\boldsymbol{\theta}_{manage}$ and $\boldsymbol{\theta}_{\#1}$ using *uni_s*. The proof score of the head atom is the minimum unification score among $[s_p, 1, 1]$, which is s_p. Then, the updated substitution set is passed on to body atoms' proofs, as the substitution set of the first body atom, denoted as S_{b1} (equivalent with S_h). Note that once a variable binds with a constant in a substitution set, the binding between the variable and the constant is fixed over all following branches that extend from this branch. The first body atom is proved by $body\_atom\_proof\_score(\#2(X, Z), S_{b1})$, which unifies $\#2(X, Z)$

with all facts in the *KB*, pairwisely, using *uni_two_atoms*. The k best unification scores ($k = 1$ here for simplicity) for the atom $\#2(X, Z)$ ($\{X/lisa\}$ in S_{b1}) give the fact in the *KB* that outputs the highest score (i.e. $colleague(tom, lisa)$ in Figure 3.1 as shown by the bold arrow). This generates an updated substitution set $S_{b2} = \{X/lisa, Y/ben, Z/lisa\}$, which is carried on to prove the body atom $\#3(Z, Y)$ that is instantiated to $\#3(lisa, ben)$ in Figure 3.1, using $body\_atom\_proof\_score(\#3(Z, Y), S_{b2})$. If $k = n$, n substitution sets would be created, each would be carried on to prove the second body atoms independently. Expanding the k best branches after each body atom unification, instead of expanding over all branches, is an optimisation strategy of *NTP*, called k_{max}. In *NTP* evaluation [Rocktäschel and Riedel, 2017], $k = 10$ is used, meaning that 10 branches are expanded independently.

The proof score of a copy of a template rule, denoted as $rule\_proof\_score$, is the maximum proof branch score over all proof branches of the copy of the template rule, where a proof branch score is the minimum of the scores of the unifications of head atom and body atoms. A branch is a 'proof branch' if there is a path from the root of the computational tree to the leaf of the tree, where each literal of the template rule is proved in this path. For example, in Figure 3.1, the branch with bold arrows is a proof branch with the maximum proof branch score, given by the minimum proof score among the atom unification (*uni_two_atoms*) between $\#1(X, Y)$ and $manage(lisa, ben)$, between $\#2(lisa, Z)$ and $colleague(tom, lisa)$, and between $\#3(lisa, ben)$ and $friend(tom, lisa)$ respectively. The proof branch score takes the minimum of the head atom and body atom unification scores, so that a branch can only have a high score if all of its literals are proved with a high score. The $rule\_proof\_score$ takes the maximum proof branch score over all of its proof branches, because as long as one proof branch can prove the query, the query is considered as 'true'.

Finally, the proof score of a query $q(\bar{c})$, denoted as $ntp^{\kappa}_{\boldsymbol{\theta}}(q, \bar{c})$, is computed as the maximum proof branch score obtained by unifying $q(\bar{c})$ with each fact in $\mathcal{F}$ and with each copy of the given template rules. Note that, unification of $q(\bar{c})$ with each fact introduces a single branch in the computational tree, and unification with a template rule introduces several branches in the computational tree depending on the hyperparameter k_{max}. Of all these branches only the one with maximum proof branch score receives gradients during backpropagation [Rocktäschel and Riedel, 2017].

3.1.3 Training

As described in Section 2.3, the neural network training involves three steps: forward passing, computing loss and backward passing, which are performed repeatedly for a number of epochs using a given training dataset τ. Differently from a typical neural network consisting of neurons with trainable parameters (as shown in Figure 2.3), such as weight and bias vectors, as described above *NTP* uses a computational tree as its architecture and its only trainable parameter is the embedding matrix, which includes the embeddings of known predicates, constants and induced predicates. Despite the different architectures, the paradigm of backpropagation is essentially the same, where trainable parameters are used to compute the proof score in the forward passing, then after computing the loss, the trainable parameters are updated by gradients. The forward passing step is as explained in Figure 3.1, which given a query, computes its best proof score using the computational tree. Then, the loss value of the query using the *NTP* computational tree is computed using Equation 3.4. In the backward passing step, the loss value is passed backward in the computational tree.

At the beginning of the training process, the embedding matrix $\boldsymbol{\theta}$ is randomly initialised. The training goal is to minimise the difference between $ntp^{\kappa}_{\boldsymbol{\theta}}(q, \bar{c})$ and its target label y (1 or 0 for true or false) for each $q(\bar{c}) \in \tau$, by optimising $\boldsymbol{\theta}$ through mini-batch gradient descent. The loss function $L_{ntp^{\kappa}_{\boldsymbol{\theta}}}$ is defined as the negative log-likelihood of $ntp^{\kappa}_{\boldsymbol{\theta}}$:

$$L_{ntp^{\kappa}_{\boldsymbol{\theta}}} = \sum_{([q,\bar{c}],y)\in\tau} -y\, log(ntp^{\kappa}_{\boldsymbol{\theta}}(q,\bar{c})) - (1-y)\, log(1 - ntp^{\kappa}_{\boldsymbol{\theta}}(q,\bar{c})) \tag{3.2}$$

To prevent a query $q(\bar{c})$ from unifying with itself, the query is temporarily masked in $\mathcal{F}$ during training.

During training, a neural link predictor, *ComplEx* [Trouillon et al., 2016], is used as a regulariser term to speedup the learning of embedding representations, so that *NTP* could focus on rule induction. Note that *ComplEx* is only used during training and does not affect testing. A loss function $L_{complex^{\kappa}_{\boldsymbol{\theta}}}$ is defined as the negative log-likelihood of $complex_{\boldsymbol{\theta}}(q, \bar{c})$:

$$L_{complex^{\kappa}_{\boldsymbol{\theta}}} = \sum_{([q,\bar{c}],y)\in\tau} -y\, log(complex_{\boldsymbol{\theta}}(q,\bar{c})) - (1-y)\, log(1 - complex_{\boldsymbol{\theta}}(q,\bar{c})) \tag{3.3}$$

Hence, the joint training loss, denoted as $L_{ntp\lambda_{\boldsymbol{\theta}}^{\kappa}}$, is given by:

$$L_{ntp\lambda_{\boldsymbol{\theta}}^{\kappa}} = L_{ntp_{\boldsymbol{\theta}}^{\kappa}} + L_{complex_{\boldsymbol{\theta}}^{\kappa}} \tag{3.4}$$

Recall that in forward passing, due to the *min* or the *max* operators in the computational tree, the proof score of each query is the unification score between a selected pair of embeddings involved when proving the query. As a result, in the backward passing, only these two embeddings that determine the overall proof score of a query are updated. Their loss values are computed by taking the partial derivative of the loss function with respect to themselves, using the chain rule of differentiation. For instance, for a query $q(a, b)$, if the proof score is generated by the unification score between a and a constant l (from a best-matching fact), then the embedding $\mathbf{a}$ would receive gradient update, as computed by the equation below (similarly for $\mathbf{l}$ by taking the partial derivative of the *NTP* loss function with respect to $\mathbf{l}$).

$$\frac{\partial L_{ntp_{\boldsymbol{\theta}}^{\kappa}}}{\partial \mathbf{a}} = \frac{\partial L_{ntp_{\boldsymbol{\theta}}^{\kappa}}}{\partial ntp_{\boldsymbol{\theta}}^{\kappa}} \times \frac{\partial ntp_{\boldsymbol{\theta}}^{\kappa}}{\partial rbf(\mathbf{a}, \mathbf{l})} \times \frac{\partial rbf(\mathbf{a}, \mathbf{l})}{\partial \mathbf{a}} \tag{3.5}$$

The embedding $\mathbf{a}$ and $\mathbf{l}$ would also receive another embedding update at the same time by taking the partial derivative of $L_{complex_{\boldsymbol{\theta}}}$ (see Section 6.1.1 for details) with respect to themselves, as shown below.

$$\begin{aligned}\frac{\partial L_{complex_{\boldsymbol{\theta}}^{\kappa}}}{\partial \mathbf{a}} = & \frac{\partial < Re(a), Re(q), Re(b) >}{\partial a} + \frac{\partial < Re(a), Im(q), Im(b) >}{\partial a} \\ & + \frac{\partial < Im(a), Re(q), Im(b) >}{\partial a} + \frac{\partial < Im(a), Im(q), Re(b) >}{\partial a}\end{aligned} \tag{3.6}$$

where *ComplEx* splits each embedding e to two parts: the upper part is denoted as $Re(e)$ and the lower part is denoted as $Im(e)$. $< a, b, c >$ sums values in $a \times b \times c$.

The gradient received by $\mathbf{a}$ for the given query is the partial derivative of the full loss from Equation 3.4, which is the sum of the two partial derivatives as given by Equation 3.5 and 3.6.

Algorithm 3.1 illustrates the high-level algorithm of *NTP* training using the *Tensorflow* framework.

```
1 def training_task(facts, tr, train_data):
2     theta = tf.Variable([SYMBOL_NUM, EMB_DIM])
3     ntp = build_computational_tree(facts, tr, theta)
4     loss = calculate_loss(ntp)
5     train_step = tf.optimiser.minimise(loss)
```

```
6     for i in range(0, TRAIN_EPOCH):
7         for query_batch in train_data:
8             theta = tf.run(train_step, feed_dict=query_batch)
```

Algorithm 3.1 The high-level training process of *NTP* using the Tensorflow framework. The algorithm starts with defining the embedding matrix *theta* as a variable to be learned. Then, the following three lines define the forward passing, loss computation and backward passing functions. The for-loop conducts the actual training using the *train_data*.

The algorithm starts with input the facts (the knowledge base), template rules (tr) and the training dataset. In line 2, it creates a 'tf.Variable' of the dimension size [*SYMBOL_NUM, EMB_DIM*]. 'tf.Variable' indicates that this matrix is a trainable parameter and it will get updated in backward passing. Line 3 defines the forward passing step by constructing the computational tree which computes the proof score of each query using the method described for Figure 3.1. Line 4 is the loss function computation step. Line 5 registers the loss function with an optimiser (an Adam optimiser is used here) and gradients are computed automatically by the optimiser. In lines 3-5, the three steps of gradient descent are defined as a general architecture with queries as placeholders. In lines 6-8, the actual training happens, which applies the three training steps using each query batch in *train_data* for *TRAIN_EPOCH*. In line 8, 'tf.run' conducts the three steps using the given query batch and updates the relevant embeddings.

Training *NTP* with a full computational tree is computationally intensive and cannot be applied to large *KBs*. Even though in $ntp^{\kappa}_{\boldsymbol{\theta}}(q, \bar{c})$, only one proof branch of the tree gets gradients in backpropagation, many thousands of branches are computed in the forward pass. For example, given a rule with 2 body literals and a knowledge base with 10000 facts, 10000×10000 branches are considered for each query using this rule. Although *NTP* introduces a k_{max} gradient approximation approach to reduce redundant computations – when a query unifies with a rule only the k_{max} branches of each body atom unification with the highest scores get propagated further and the other branches are discarded – *NTP* still needs to unify each body atom with all possible facts in forward propagation to pick the k_{max} to expand, generating $k \times 10000$ branches, so it still suffers from scalability issues.

3.2 Introducing *TNTP*

In Section 3.1, we have presented the *NTP* approach, which uses gradient descent to learn embeddings and induce rules that can be used to complete a given *KB*. However, *NTP* has scalability issues related to the fact that during training, the proof of body literals of given

template rules is done with respect to all facts included in the *KB*.

In this chapter we address this limitation by introducing topical template rules (denoted as *TTRs*), which allows the proof of body literals to focus on these facts from the *KB* that are 'topic relevant'. One way to achieve this is to cluster facts in the *KB* into *topics*, based on their semantics similarity in high-dimension space, and define template rules in terms of the identified topics, so to constrain the unification process during training. The outcome is a diverse set of induced rules where the induced predicates of *TTRs* are learned.

Our approach, called *Topical NTP* (*TNTP*), involves two phases of training: the topic generation phase and the induction training phase. The first phase identifies clusters over a large *KB*, referred to as *topics*. The second phase uses these clusters (topics) to control the soft-unification of predicates during the learning process, with the effect of reducing the size of the computational tree needed to induce first-order rules. To do so, *TNTP* uses *topical template rules* which assign a topic to each induced predicate in body atoms. This also enables *TNTP* to induce a diverse set of rules. Our experiments demonstrate that *TNTP* is over 20 times faster than *NTP* on benchmark datasets (the speedup depends on the number of topics), while achieving higher accuracy of query answering predictions. The details of the evaluation can be found in Chapter 7.

A second problem of *NTP* is that the number of branches resulting from matching a query with copies of template rules used in the computational tree is low compared to the number of branches resulting from matching a query with facts from the given *KB*. To solve this problem, *TNTP* makes use of two hyperparameters to amplify the proof scores of copies of *TTRs* used in the computational tree. Consequently, rules are used more often during training, thus improving their final accuracy of query answering predictions.

The rest of this chapter is organised as follows. We present the methodology of *TNTP*, including the high-level architecture overview, the 2-step training, the topic generation, *TTRs* and their use in computational trees, and the high-level algorithm. Then, we discuss the addition of the amplification hyperparameters. We conclude with presenting the decoding mechanism of learned topical induced rules into rules that are human interpretable. The material in this chapter was presented in our *GCAI* paper [Xia et al., 2020].

3.3 The *TNTP* Approach

Unlike *NTP*, *TNTP* involves two training phases (see Figure 3.2): the topic generation phase and the induction training phase. The topic generation phase learns embedding representations of the predicates in the *KB* and computes clusters over these embeddings. These clusters can be seen as topics, which group predicates with similar embedding representation in high-dimension space. These topics are used in the induction training phase to select the subset of topic-related facts to be used in each body atom unification.

A high-level overview of the *TNTP* architecture is given in Figure 3.2.

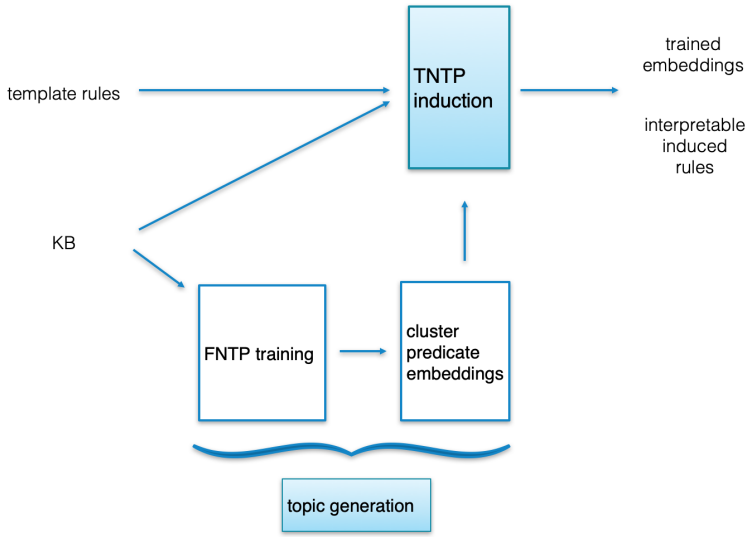

Fig. 3.2 The training pipeline of *TNTP*. The inputs of *TNTP* are the knowledge base (*KB*) and template rules. There are two main steps: topic generation and *TNTP* induction. Topic generation creates topics using trained embeddings of *FNTP*. These topics are used for *TNTP* induction, which generates trained embeddings and interpretable induced rules.

TNTP takes as inputs a *KB* and a set of template rules. The first step of the topic generation is to train the embeddings of the facts in the *KB*. This is done using a fact-only *NTP*, denoted as *FNTP*. The learned embeddings capture semantics of the predicate symbols. An example of the 3-dimension reduction of the learned embeddings for some predicates in the *Nation* dataset is given in Figure 3.3, where the distance between points indicates how semantic similar they are. Then, a clustering algorithm (explained in Section 3.3.1) is applied over the trained embeddings, which groups semantically related high-dimension predicates into topics, i.e. predicates whose embeddings are close in space.

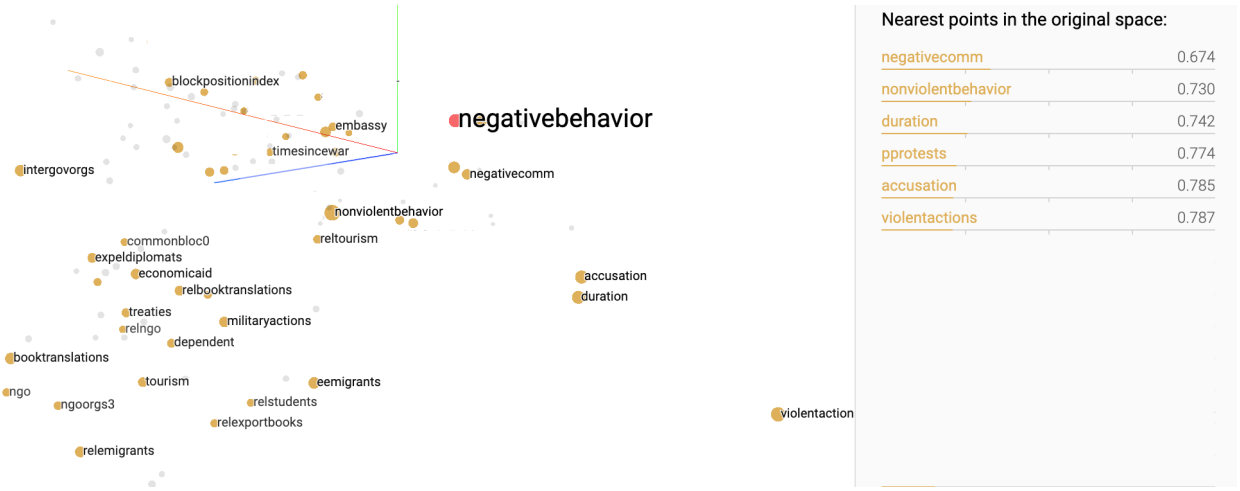

Fig. 3.3 An example vector space of predicate embeddings in the *Nations* dataset after the *FNTP* training. The panel on the right-hand side lists the nearest neighbours of *negativebehavior* and the scores are the euclidean distances between two points. The closer the distance between two points, the more similar they are.

As an illustration, Figure 3.3 shows the neighbours of the predicate *negativebehavior* in 3-dimension space and their respective euclidean distances on the right hand side panel. As shown by the list, a majority of the neighbours of *negativebehavior* are 'unfriendly relations', which fits human commonsense. Note that although most predicates in a cluster have similar semantic meanings, some outliers could still exist, such as the 'nonviolentbehavior' in the neighbours of 'negativebehavior'. When clusters are used as a filter to select potential predicates, these outlier predicates might no longer be selected for matching with topic-based induced predicates, or be wrongly selected for matching, so proofs involving queries using these outlier predicates could be affected. However, most predicates in the topic could still match with their most relevant predicates, while reducing the redundant computations in a computational tree. Our experiments in Chapter 7 demonstrate that although using topics might affect the proof of some outlier predicates, the accuracy of *TNTP* query answering predictions is better than that of *NTP*. As a result, outliers in topic generation are acceptable, as long as most of predicates in a topic are semantically similar.

The *TNTP* induction phase generates copies of *TTRs* and constructs the topical computational tree using these *TTRs*. Instead of unifying with all facts as in *NTP*, in *TNTP*, a body literal unifies only with facts in the topic specified by the *TTR*. This cuts down the number of unifications required and leads to a more compact computational tree. The topical computational tree is then trained using gradient descent, in the same way as described above for *NTP*, using the loss function given in Equation 3.2-3.4.

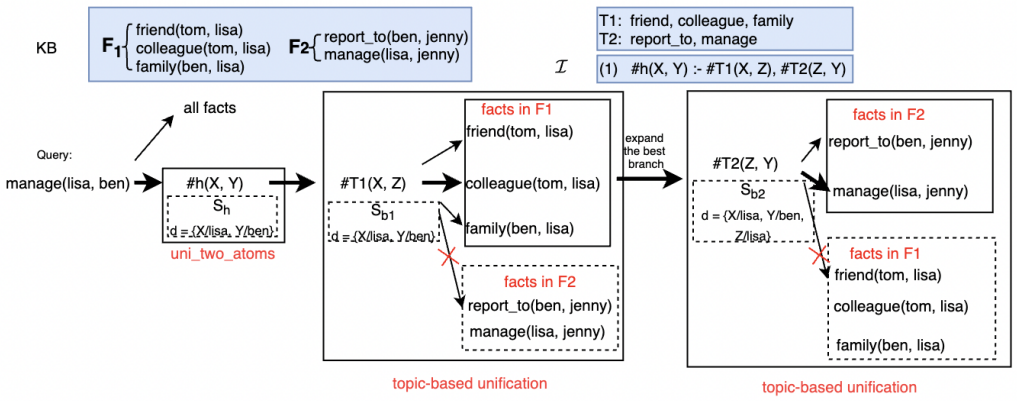

Fig. 3.4 An example of *TNTP* computational tree using a topical template rule. *TNTP* receives the knowledge base *KB*, a copy of a template rule in $\mathcal{I}$ and topics ($T1$ and $T2$) as inputs. The *KB* is partitioned according to topics (F_1 and F_2). Given a query, it matches with all facts in *KB* and all rules in $\mathcal{I}$. As in Figure 3.1 the fat arrows depict, for illustration, the highest scoring branch. When proving a copy of *TTR*, the query firstly matches with the head atom, proved using *uni_two_atoms* with an updated substitution set. Then, *topic-based unification* is applied for each body literal, where each body literal only matches with facts in the same topic, ignoring irrelevant branches as shown by the branches with red crosses.

Figure 3.4 presents an illustration of efficiency improvement using *TNTP* over *NTP*. Consider, for instance, the same *KB* and the single copy of the template rule $\#h(X, Y)$:- $\#1(X, Z), \#2(Z, Y)$ from Figure 3.1. Let us assume that the topic generation phase gives two clusters $T1$ and $T2$ (see Figure 3.4). The copy of the template rule is converted to a copy of a *TTR* by assigning topics to predicates in body literals, forming $\#h(X, Y)$:- $\#T1(X, Z), \#T2(Z, Y)$ (see Section 3.3.2 for details of *TTR* conversion). The topical computational tree for the query $manage(lisa, ben)$ is then given in Figure 3.4. The unification of predicates is limited to just the facts from the *KB* that belong to the topic indicated by the copy of the *TTRs*. For instance, the unification of the body predicate $\#T1(X, Z)$ considers only the three facts in $F1$, instead of all five facts in the *KB*.

3.3.1 Topic Generation

We describe here the two steps of the topic generation phase. The first step consists of training the embeddings for predicates and constants using *FNTP*, which generates an embedding matrix $\boldsymbol{\theta}$ of the size $[n, 100]$, where n is the total number of symbols (predicates and constants in the given *KB* and induced predicates) and 100 is the size of embedding vectors. This is shown in lines 2-5 in Algorithm 3.2. The second step applies a clustering algorithm over the learned predicate embeddings, as shown in lines 6-8 of Algorithm 3.2.

```
1 def TOPIC_GEN(facts, predicate_ids, train_data, t_cluster):
```

```
2     theta = init_embed(facts)
3     fntp = build_fact_only_tree(facts, theta)
4     for i in range(0, PRETRAIN_EPOCH):
5         theta = train(fntp, theta, train_data)
6     pca_theta = pca(theta, D_DIMENSION)
7     pca_pred_theta = extract_predicate(pca_theta, predicate_ids)
8     pred_cluster_dic = compute_clusters(pca_pred_theta, t_cluster)
9     return pred_cluster_dic
```

Algorithm 3.2 A high-level algorithm of the topic generation phase. Strings in capital letters are hyperparameters of the algorithm. The algorithm firstly trains a *FNTP* to get a trained embedding matrix *theta*. Then, it uses *PCA* algorithm to reduce the dimension of *theta*, which are then used to generate topics by a clustering algorithm.

At the beginning of topic generation, embeddings are initialised randomly (line 2). To learn the embedding matrix $\boldsymbol{\theta}$ (theta in Algorithm 3.2) for clustering, we use *NTP* with a facts-only computational tree, denoted *fntp* (line 3). Figure 3.5 illustrates an example *FNTP* computational tree.

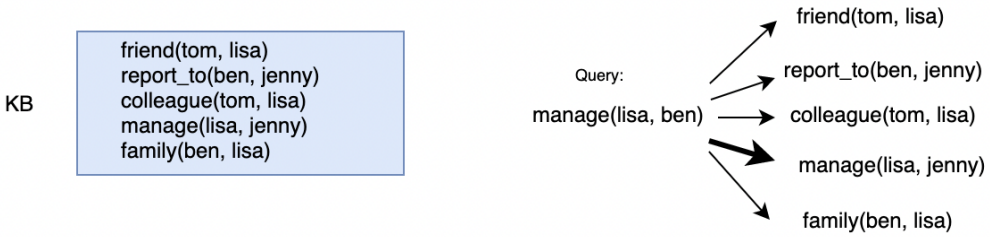

Fig. 3.5 An example *FNTP* computational tree. No template rules are given to *FNTP* and each query is proved by the fact in the knowledge base that generates the highest proof score. During the *FNTP* training, the embeddings are trained to capture semantic information, where semantically similar embeddings are close in vector space.

FNTP is trained through gradient descent according to the loss function given in Equation 3.4. Each query q unifies with all facts, except itself, and outputs the highest unification score, which is used to update $\boldsymbol{\theta}_q$. The purpose of this training is to force embeddings of semantically similar predicates and constants to be closer in vector space. For example, if the fact $manage(lisa, jenny)$ generates the highest proof score for the query $manage(lisa, ben)$, it would be because the vector representations of *jenny* and *ben* are close in vector space. The training is performed using a given number of epochs (*PRETRAIN_EPOCH*). At each epoch, a different query set is used. The positive queries are the same but the corrupted facts (4 per query sampled randomly) are different in each query set, as explained in Section 2.1. It is known that high-dimension embeddings inflate Euclidean distances involved in K-means training [Hinneburg

and Keim, 1999]. Thus, before applying clustering, we use Principal Component Analysis (*PCA*) [Jolliffe, 1986], to reduce the matrix $\boldsymbol{\theta}$ to a d-dimension $\boldsymbol{\theta}_d$. (We used $d = 20$, because the topics generated by this value lead to the best accuracy of query answering prediction in experiments.) Then, the predicate embeddings are extracted from $\boldsymbol{\theta}_d$ using *predicate_ids*, denoted as $\boldsymbol{\theta}_{d_P}$ (line 7 of Algorithm 3.2).

We explored two clustering algorithms to generate the clusters: *K-means* [Macqueen, 1967] and *Agglomerative* [Gower and Ross, 1969] (see Section 2.4.1). We select one of the two methods to cluster predicate embeddings $\boldsymbol{\theta}_{d_P}$ to T clusters (line 8). Both algorithms are unsupervised clustering algorithms which partition a set of N-dimension embeddings into T sets. In our topic generation, we use both clustering algorithms and pick the number T of topics that gives the best accuracy of the *TNTP* with respect to a validation set.

As an example, we show part of a topic distribution of the *Nations* dataset. *Nations* dataset contains 56 predicates and they have been divided into 5 topics using the *K-means* algorithm. Some of the predicates in two of the topics ($T1$ and $T2$) are given below. It is easy to see that the topics generated by clustering embeddings fit human commonsense. For example, $T1$ contains 'unfriendly' relationships between countries; $T2$ contains 'friendly' relationships between countries.

T1: time since war, negative behavior, accusation, protests, military actions, ...

T2: military alliance, exports, ngo, students, commercial aid, tourism, ...

To choose the best T number of clusters, we use the Silhouette score [Rousseeuw, 1987] to evaluate the quality of the clustering results. The Silhouette score measures the quality of clusters: that is the compactness of points within the cluster and the disparity between points in this cluster against other clusters. The higher the Silhouette score is, the better is the quality of the clusters. In practice, however, we found that the highest Silhouette score may not necessarily generate the best accuracy of query answering predictions after *TNTP* training. If predicates are spread out to more clusters, fewer facts are used in each body unification, causing some relevant facts to be missed, hence leading possibly to a lower accuracy of query answering predictions. So, although Silhouette score is a good way to reflect the quality of topic generation, in practice the choice of the correct number of clusters is determined by its effect on the accuracy of query answering predictions and induction performance of the *TNTP* model. In general, a lower number of topics leads to better rule induction quality and a higher number of topics leads to faster training with less induced rules. We demonstrated this in our evaluation in Section 7.4.3.

By applying either the *K-means* or *Agglomerative* cluster algorithm (whichever is most appropriate) to the predicate embedding $\boldsymbol{\theta}_{d_P}$, we generate clusters $\{\mathcal{T}_1, \mathcal{T}_2, ..., \mathcal{T}_T\}$. The topic of a predicate p is given by the cluster its embedding belongs to:

$$topic(p) = i \text{ iff } \boldsymbol{\theta}_p \in \mathcal{T}_i \tag{3.7}$$

Note that each predicate belongs to exactly one topic.

After generating topics, KB is partitioned according to the topics of the atomic predicates, forming subsets $\mathcal{F}_i$, as defined in Def. 3.3.1. It is this partition of KB that is used during topical unification, as illustrated in Figure 3.4.

Definition 3.3.1. Given a knowledge base KB with a signature $< \mathcal{P}, \mathcal{C} >$ and a set $\{\mathcal{T}_1, \mathcal{T}_2, ..., \mathcal{T}_T\}$ of T topics, the set of facts $\mathcal{F}$ in KB is partitioned into T (sub)sets of facts, where for each set $\mathcal{F}_i$, $1 \leq i \leq T$, a fact $p(a, b) \in \mathcal{F}_i$ iff $\boldsymbol{\theta}_p \in \mathcal{T}_i$.

3.3.2 Topic-based Unification

We describe now how *TNTP* supports topic-based rule induction. It includes two main parts: generation of *TTRs* and topic-based unification.

Generation of *TTRs*

TNTP receives as input a set of template rules. *TTRs* are constructed from the input template rules using the generated topics. Specifically, each copy of a template rule is assigned a topic for each body atom in the rule. For instance, a copy of the template rule, $\#1(X, Y)$:- $\#2(X, Y), \#3(X, Y)$, might be converted to $\#H(X, Y)$:- $\#T5(X, Y), \#T2(X, Y)$. The head predicate is represented as $\#H$ to indicate that it is supposed to unify with any given predicate. The body predicates $\#T5$ and $\#T2$ indicate that during training, the two induced body predicates are expected to be similar to predicates in topic 5 and topic 2 respectively. Some copies of *TTRs* might have the same topic combinations, but each copy has independent embeddings and will lead to different induced rules. We indicate with $\mathcal{I}$ the allowed set of copies of all *TTRs*.

To cover as much of the search space of learnable rules as possible, we consider a range of *TTRs* over the possible topics. The selection of topics can either be based on on a distribution

over T topics giving equal probability to each topic, or on a distribution in which the probability of each topic is weighted according to the size (number of facts) in the topic. We call the first method *fair selection* and the latter *weighted selection*, formally defined below.

Definition 3.3.2. Let KB be a knowledge base partitioned into T topics. The probability of a predicate p to be chosen from a topic i according to a *fair selection* is $1/T$.

Definition 3.3.3. Let KB be a knowledge base including $|F|$ facts partitioned into T topics $\{\mathcal{T}_1, \mathcal{T}_2, ..., \mathcal{T}_T\}$. The probability of selecting a predicate p from $\mathcal{T}_i$ according to a weighted selection is given by $|\mathcal{T}_i|/|F|$.

As discussed in Section 3.3.1, since the clustering mechanisms will unlikely generate clusters with similar sizes, to have a better coverage of the inductive search space during training, the weighted selection is preferred to fair selection. In our experiments, when the entire search space can be covered by half of the number of copies of a template rule (10 in this case), 50% copies of the template rule are converted to *TTRs* by *fair selection* and 50% of it are converted to *TTRs* by *weighted selection*. Otherwise, all copies of the template rule are created by weighted topic assignment. This mixture selection approach enables most topics to be used in *TTRs* while giving topics with more facts higher weights.

For example, given 5 topics and a template rule $\#1(X, Y) : -\#2(X, Y)$ with 20 copies to be generated as *TTRs*, 10 copies of it would be created by fair topic assignment of the form:

(2) #H(X, Y) :– #T1(X, Y).
(2) #H(X, Y) :– #T2(X, Y).
(2) #H(X, Y) :– #T3(X, Y).
(2) #H(X, Y) :– #T4(X, Y).
(2) #H(X, Y) :– #T5(X, Y).

(2) indicates there are two copies of its TTR. The topics of the other 10 copies of *TTRs* are selected by weighted topic assignment.

For a template rule, such as $\#1(X, Y) : -\#2(X, Z), \#3(Z, Y)$ with 20 copies to be generated as *TTRs*, the selection of topics will not cover the entire search space (which would require 25 copies, since there are 5 topics and 2 body atoms). As a result, topics of all copies are generated by weighted topic assignment. For example, in the case that $|F_1|$ and $|F_2|$ are each double the size of F_3, F_4 and F_5, the chance of $T1$ or $T2$ appearing in each body atom of 20 copies would be 2/7, whereas the chance of $T3$, $T4$ or $T5$ would be 1/7.

By using *TTRs* instead of non-topical template rules, both the computational efficiency and the diversity of rules induced by *TNTP* are improved compared with *NTP*. For instance, when applying *NTP* to the *UMLS* dataset, 12 out of 20 induced rules using the same *TR* are exactly the same, whereas *TNTP* induces 20 different rules. The computational efficiency improvement is obvious, because in each body unification, only a subset of facts is used instead of all facts in *KB*. This is demonstrated in our evaluation (see Section 7.3.2). The improved diversity of induced rules is due to the topic-based selection strategy that guarantees a wide topic combination in the *TTRs*.

Topic-based Unification

TNTP uses a different body atom unification mechanism than the one used in *NTP*. We have seen that in the *NTP* architecture when a body atom $q(\bar{c})$ is queried, all facts in *KB* are considered in the construction of the computational tree. In our *TNTP*, the *TTRs* specify the topic of the body atoms to be considered in the construction of the computational tree and therefore in the unification. If the query $q(\bar{c})$ is of topic i, the unification of $q(\bar{c})$ is computed with respect to all facts F_i in topic i. We call this unification, *topic-based unification*.

Definition 3.3.4. Given a knowledge base *KB* that is partitioned into T sets of $\mathcal{F}_i$ $(1 \leq i \leq T)$ according to the topics $(\mathcal{T}_1, ..., \mathcal{T}_T)$, the *topic-based unification* of a body atom $\#T_n(\bar{c})$ with topic n $(1 \leq n \leq T)$ and arguments $\bar{c}$, is the unification between $\#T_n(\bar{c})$ and all facts in $\mathcal{F}_n$.

It is important to point out that in *TNTP* topics are not updated during training, as they are used as a bias to filter unimportant information. Since topics are generated according to well-trained embeddings of *FNTP*, the clustering quality is preserved. Generally, topics may be a little different after training, because there are changes on boundaries of clusters in vector space. These changes on the boundary do not matter because predicates in the boundary of a topic are usually not close to any predicates (otherwise they would be near the centroid of a topic). As a result, the effect of changing topics for these predicates is minor. Our experiments in Section 7.3.1 demonstrate that *TNTP* is able to achieve good accuracy of query answering predictions using topic-based unification.

3.3.3 High-level Algorithm of *TNTP*

Algorithm 3.3 captures at a high-level the *TNTP* induction training mechanism.

```
1 def TNTP_induction(facts, rules, train_data, topics):
```

```
2    theta = init_emb(facts, rules)
3    ttrs = build_ttr(rules, topics)
4    tntp = build_topical_computational_tree(facts, ttrs, theta)
5    for i in range(0, TRAIN_EPOCH):
6        theta = train(tntp, theta, train_data)
7    induced_rules = decode_rules(theta, ttrs)
8    return theta, induced_rules
```

Algorithm 3.3 The *TNTP* induction pipeline that uses *TTRs* to construct computational tree and trains embeddings for rule induction.

Algorithm 3.3 receives as input a set of positive facts (KB), template rules, training data and topics. At the beginning, the embedding matrix $\boldsymbol{\theta}$ (theta in Algorithm 3.3) is initialised randomly. Note that we do not reuse the embeddings learnt by *FNTP*, because mixing well-trained embeddings with untrained induced predicate embeddings does not help rule induction. If *TSNTP* starts with the trained embeddings of *FNTP* (including known predicates and constants) and randomly initialised induced predicate embeddings, facts would be selected for most of the time when proving queries, because their trained embeddings could match well, hence the *TTRs* with random embeddings would rarely get any gradients. In line 3, template rules are assigned topics according to Section 3.3.2, forming $TTRs$. Next, the topical computational tree is built using these $TTRs$. Topic-based unification is applied for each body atom. In lines 5-6, *TNTP* is trained through gradient descent. Finally, induced rules are generated by decoding the trained embeddings with respect to the KB (see Section 3.5).

We list here the software frameworks used in all our three systems. Our systems are developed using *Python 3.6* version and *Tensorflow 1.15* version [Abadi et al., 2016]. We use Tensorflow, because initially we used the open source code of *NTP*[1] to run some initial exploratory experiments on *NTP*. However, we have rebuilt the code completely for the development of our systems. For generating clusters, we inspected two typical clustering algorithms from *sklearn.cluster* packages, the *Kmeans* and *Agglomerative* clustering algorithms. *Kmeans* identifies clusters using a top-down approach and *Agglomerative* identifies clusters using a bottom-up approach. Before applying clustering algorithms, to avoid the curse of dimension as explained in Section 3.3.1, we also use the *sklearn.decomposition* package to apply *PCA* to trained embeddings. In our experiments, we use both clustering algorithms to generate clusters using *PCA*-compressed trained embeddings with 20-dimension. Our experiments demonstrate that conducting dimension reduction indeed leads to better accuracy of query answering predictions, but the improvement is

[1] https://github.com/uclnlp/ntp

not significant. This is most likely because *TNTP* does not require perfect clusters – empirically we found that as long as most relevant predicates are in the same topic, the rule induction works well. Clusters are selected based on the accuracy of query answering predictions using a *TNTP* with fixed hyperparameters. We fine-tune other hyperparameters after we select the clusters. In Chapter 7.2, we explain the experiment settings and the procedure of tuning these hyperparameters in detail.

3.4 Hyperparameters for Amplifying Rule Learning

Recall that the proof score of a rule is the minimum proof score among proof scores of each head and body literals. As a result, a short rule is more likely to yield a higher proof score than a long rule. Thus in general matching with a fact is more likely to give a higher proof score than matching with a rule. Similarly, the proof score of two atoms is the minimum unification score among its predicate-predicate and argument-argument unification. Induced predicates can only get updated if the induced rule generates the highest proof score, whereas constants can get updated in both fact and rule branches. As a result, constants are updated more frequently than predicates, especially than induced predicates. Therefore, during training, the rule induction gets dominated by query-fact soft-unification, and consequently, the induced predicates in *TTRs* may not be trained well. This was also empirically verified.

To improve rule induction, we therefore introduced in *TNTP* two hyperparameters α and β. The α hyperparameter scales induced predicate unification scores (see Definition 3.4.1) while the β hyperparameter scales rule unification scores (see Definition 3.4.2). These amplification hyperparameters boost the unification scores obtained via *TTRs* and increase the chances for induced rules to receive gradients. In this way, the embeddings of induced predicates get trained more frequently and better rules are learned. The effects of these hyperparameters are demonstrated in the evaluation, in Section 7.4.1.

By way of exemplification, consider a query $q(a, b)$ to be proved using a copy of a *TTR*: $\#H(X, Y)$:- $\#T1(X, Y), \#T2(X, Y)$, which after substitution becomes the copy $\#H(a, b)$:- $\#T1(a, b), \#T2(a, b)$. During each backpropagation, a and b have three chances to be updated within the same branch of the computational tree, while each of $\#H$, $\#T1$ and $\#T2$ has one chance. Even worse, these $\#P$ can only be updated if the rule generates the highest proof score over other rules. In contrast, q, a and b can also be updated when unifying with facts instead of

rules. As a result, the chance for $\#P$ to get updated is very low, compared with constant and known predicates in $\mathcal{P}$. The learned embeddings of constants capture therefore better semantic information in vector space, giving a better unification score, whereas the learned embeddings of predicates are more noisy, giving in general a lower unification score between two predicates. The amplification hyperparameters α and β help to balance this out.

Definition 3.4.1. Given an embedding matrix $\boldsymbol{\theta}$, a substitution set d, an induced predicate amplification hyperparameter α ($\alpha > 0$), the unification between arguments i and j (i, j can be predicates, constants or variables), denoted as $uni\_s(i, j, d)$, generates a tuple (d_u, s) defined as follows, where d_u is the updated substitution set and s is the unification score:

$$(d_u, s) = \left\{ \begin{array}{ll} (d', 1) \quad \text{where } d' = d \cup \{i/j\} & \text{if is\_var}(i) \text{ and } i \notin d \\ (d,\ rbf(\boldsymbol{\theta}_c, \boldsymbol{\theta}_j)) & \text{if is\_var}(i) \text{ and } i/c \in d \\ (d,\ \alpha \times rbf(\boldsymbol{\theta}_i, \boldsymbol{\theta}_j)) & \text{if is\_ind\_pred}(i) \\ (d,\ rbf(\boldsymbol{\theta}_i, \boldsymbol{\theta}_j)) & \text{otherwise} \end{array} \right\} \tag{3.8}$$

Recall that the proof score of $q(\bar{c})$ using a rule $\#H(X, Y)$:- $\#T1(X, Z)$, $\#T2(Z, Y)$ is the minimum proof score among the three literals $\#H(X, Y)$, $\#T1(X, Z)$ and $\#T2(Z, Y)$. In contrast, the proof score of $q(\bar{c})$ using a fact $p(c, d)$ is just the proof score of unifying these two atoms. Clearly, the longer the rule, the more atoms need to be proved. With more atoms, if one of the atoms has a low unification score, the proof score of the rule is low. As a result, a fact tends to get a higher unification score than a rule. In order not to penalise rules, a rule amplification hyperparameter β is applied to amplify rules' unification scores by β.

Definition 3.4.2. Given a knowledge base KB with facts $\mathcal{F}$, a set $\mathcal{I}$ of copies of $TTRs$, an embedding matrix $\boldsymbol{\theta}$, a query $q(\bar{c})$ and a rule amplification hyperparameter β ($\beta > 0$), we denote

$$SF(q(\bar{c})) = \{tanh(rule\_proof\_score(q(\bar{c}), r, \mathcal{F}) | r \in \mathcal{F})\} \tag{3.9}$$

and

$$SI(q(\bar{c})) = \{tanh(\beta \times rule\_proof\_score(q(\bar{c}), r, \mathcal{F}) | r \in \mathcal{I})\} \tag{3.10}$$

The $TNTP$ proof score for the query $q(\bar{c})$ is given by:

$$tntp^{\kappa}_{\boldsymbol{\theta}}(q, \bar{c}) = max\{SF(q(\bar{c}) \cup SI(q(\bar{c})))\} \tag{3.11}$$

In the above definition, the $rule\_proof\_score$ function is the same as the one given in Section 3.1.2. The use of the amplification hyperparameters α and β yield $TNTP$ proof scores in

different ranges, depending on whether they were derived from matching with a fact or a rule (respectively $(0..\alpha]$ or $(0..\alpha \times \beta]$). As these are to be compared they are both normalised by *tanh* to the same range of $(0, 1)$. This $tntp^{\kappa}_{\boldsymbol{\theta}}$ is then used to compute the loss function as defined in Equation 3.2 to 3.4, replacing $ntp^{\kappa}_{\boldsymbol{\theta}}$. The role of α and β in the proof is illustrated by Figure 3.6.

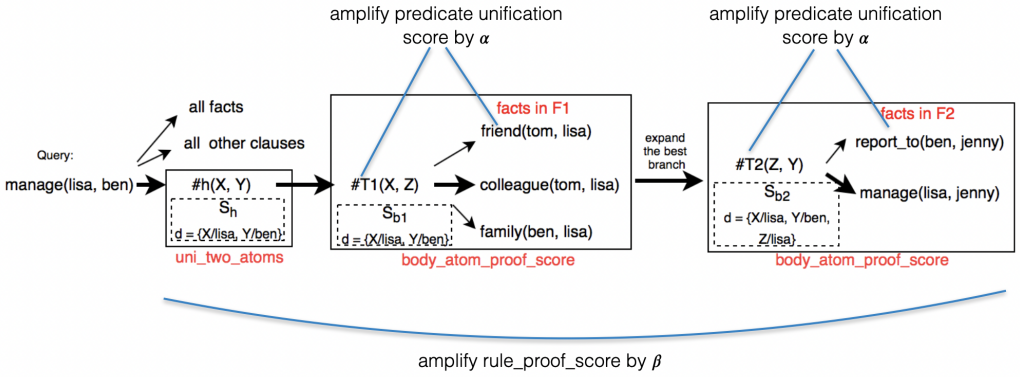

Fig. 3.6 The role of α and β in the proof. α amplifies the predicate unification score and β amplified the unification score of each rule, as defined in Definition 3.4.1 and Definition 3.4.2.

3.5 Interpretability

Once the *TNTP* model has been trained, the learned induced rules are still in terms of trained induced predicate embeddings. In this section we explain how these induced rules can be converted to symbolic rules for human interpretation. Human-interpretability is important, because it enables an understanding of how the system makes decisions. In contrast, neural networks often work as a blackbox, making it hard for humans to understand its decision making process. This problem limits the use of neural networks in some accuracy-sensitive industries and has been widely discussed in the field of *explainable AI* [Cyras et al., 2021, Adadi and Berrada, 2018], which aims to improve the trust and transparency of *AI-based* systems. Although there are works that try to explain neural networks using logic representations (such as [Ferreira et al., 2022]), these systems usually work as post-processing after neural networks are learned. In contrast, rules in our systems are learned during training and they are optimised for many epochs. In what follows, we present our 'decoding' mechanism that converts embedding-based induced rules to interpretable first-order rules.

The learned embedding matrix $\boldsymbol{\theta}$ captures the semantic relationships among $\mathcal{P}$, $\#\mathcal{P}$ and $\mathcal{C}$, so decoding of such matrix into interpretable rules is performed with respect to the vocabulary of *KB*. To do this, all copies of *TTRs* in $\mathcal{I}$ are first ranked according to the frequency that

an induced rule is used by positive (i.e. uncorrupted) queries in the last iteration of training. Among all induced rules, some of them are rarely picked as the best performing branch, so they receive very few gradient updates. These rules are not worth decoding. Therefore, to decode induced rules into the discrete space of the language in KB, we define a frequency threshold and only induced rules with frequency higher than this threshold are decoded. The frequency threshold can be any value between 0 and 1. For example, if its value = 0.2, it means that a rule is only decoded if it was used by the system over 20% of times when proving positive queries in the last epoch of training. Using the frequency threshold, we get a set of 'frequently used' copies of $TTRs$ to decode. We also define a predicate decoding threshold (between 0 and 1) for predicate decoding in those induced rules that exceeded the frequency threshold. Induced predicate, such as $\#T2$, is decoded into a known predicate, say p, only if its unification score is above the predicate decoding threshold. Note that the frequency threshold and predicate decoding threshold are used to generate a selected set of induced rules. A broader set of induced rules can be generated by lowering these threshold values.

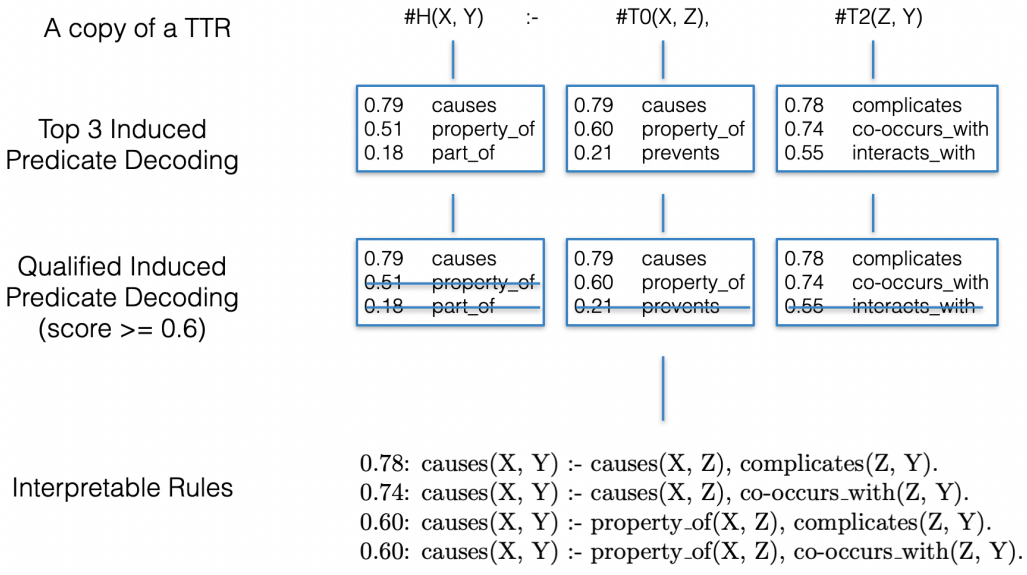

Fig. 3.7 An example of induced rule decoding. Given a copy of a TTR that exceeds the frequency threshold, the top 3 nearest neighbours of each induced predicate in the copy of TTR are identified. Inside each blue box, the score is the unification score between the induced predicate and the matched known predicate. The higher the score, the better the match. Then, the qualified decoding of each induced predicate is selected based on whether it passes the predicate decoding threshold. In the end, a set of interpretable rules is constructed from the induced rule by taking all combinations of the decoded predicates that pass the predicate decoding threshold. The score of an induced rule is the minimum value of the decoding scores of the decoded predicates. For example, the score 0.78 of the first decoded rule is the minimum of the scores of the three predicates, namely 0.79, 0.79 and 0.78 respectively.

Figure 3.7 illustrates the predicate decoding process of a copy of a TTR that exceeds the frequency threshold. Each induced predicate in such a rule is decoded to one of its top 3 nearest

neighbours in the given topic (in all topics for head predicates). Here, three nearest neighbours are presented for human interpretation, but more or fewer can be specified by a user as well. The unification score between the induced predicate and the potential decoded predicate is shown on the left hand side of each potential decoded predicate (three boxes on the top). Then, the potential decoded predicates with unification scores below a predicate decoding threshold (0.6 used in the example) are deleted (three boxes in the middle). Various rules are then formed using different combinations of the remaining predicates. A decoded rule is only constructed from decoded predicates with decoding scores above the decoding threshold. The decoding score of a rule is calculated as the minimum score of its predicate decoding scores. If the decoding score of a constructed rule is close to or equal to 1, it indicates that all decoded predicate scores are also close to or equal to 1, which in turn indicates the decoded rule is likely to be a good match of the embedding-based induced rule. The final decoded rules (at the bottom) are the rules with scores above a predicate decoding threshold (0.6 used in the table), which indicates that all of its induced predicates have a score equal to or bigger than it. Table 3.1 gives further examples of learned rules using *TNTP* from the *UMLS* dataset. Recall that when applying *NTP* to the *UMLS* dataset, 12 out of 20 induced rules using the same *TR* are exactly the same, whereas *TNTP* induces 20 different rules.

Topical Template Rule	Top 3 Induced Predicate Decoding	Decoded Rules
#H(X, Y) :- #T2(X, Z), #T2(Z, Y).	#H [0.81, 0.21, 0.17] [affects, evaluation_of, indicates] #T2 [0.70, 0.37, 0.27] [affects, precedes, process_of] #T2 [0.82, 0.82, 0.23] [interacts_with, precedes, degree_of]	0.70: affects(X, Y) :- affects(X, Z), interacts_with(Z, Y). 0.70: affects(X, Y) :- affects(X, Z), precedes(Z, Y).
#H(X,Y) :- #T1(X,Y).	#H [0.87, 0.85, 0.20] [location_of, carries_out, adjacent_to] #T1 [0.84, 0.81, 0.28] [carries_out, location_of, interconnects]	0.84: location_of(X, Y) :- carries_out(X, Y). 0.84: carries_out(X, Y) :- carries_out(X, Y). 0.81: location_of(X, Y) :- location_of(X, Y). 0.81: carries_out(X, Y) :- location_of(X, Y).
#H(X,Y) :- #T0(X, Z), #T2(Z, Y).	#H [0.79, 0.51, 0.18] [causes, property_of, part_of] #T0 [0.79, 0.60, 0.21] [causes, property_of, prevents] #T2 [0.78, 0.74, 0.55] [complicates, co-occurs_with, interacts_with]	0.78: causes(X, Y) :- causes(X, Z), complicates(Z, Y). 0.74: causes(X, Y) :- causes(X, Z), co-occurs_with(Z, Y). 0.60: causes(X, Y) :- property_of(X, Z), complicates(Z, Y). 0.60: causes(X, Y) :- property_of(X, Z), co-occurs_with(Z, Y).

Table 3.1 The decoding of some rules induced in *UMLS* dataset using the decoding mechanism described above (with the predicate decoding threshold = 0.6).

At inference time the computational tree and the interpretable decoded rules are used to identify how a query has been proved. For instance, consider the query *interacts_with(antibiotic, biologically_active_substance)*. This is run through the trained computational tree to extract the best proof branch, which itself corresponds to a trained copy of a *TTR*. The decoded rule that gives the highest score with respect to such copy of the *TTR* is selected and returned, so making the inference interpretable. However, note that the returned rule is only an approximation

of the facts from the *KB* that are used to prove a given query, as the unification in *TNTP* is soft. So for the query mentioned above, the induced rule would be: *interacts_with(X, Y) :- interacts_with(Y, X).* The body atom *interacts_with(Y, X)* is proved by the fact *interacts_with(biologically_active_substance, receptor)* from the *KB* in the *UMLS* dataset, where the constant 'receptor' is close in high dimensional space to the constant 'antibiotic' that appears in the query.

Summary. *TNTP* introduces a two-phase training to use topics to narrow down the number of facts involved in each body atom unification. It firstly uses a fact-only *NTP* to train embeddings and extract clusters from embeddings. Then, it builds a set of *TTRs*, which contains template rules with different topic combinations, aiming to cover the search space as much as possible. With the simplification brought by topics, in each body atom unification, only a subset of indexed facts in *KB* is used, improving the computational efficiency. In addition, it introduces two hyperparameters to improve the chances of induced rules to be used in the proof, so leading to better induced rules. As shown in Chapter 7, with these changes, *TNTP* achieves a better accuracy of query answering predictions while being more computational efficient.

Chapter 4

Topic-Subdomain NTP

The *TNTP* system demonstrates that grouping predicates into topics helps to filter facts during induction learning. In this chapter, we apply a similar clustering method to group constants into *subdomains*. We extend *TNTP* to the *Topic-Subdomain NTP* (*TSNTP*). An overview of the *TSNTP* architecture is given in Figure 4.1.

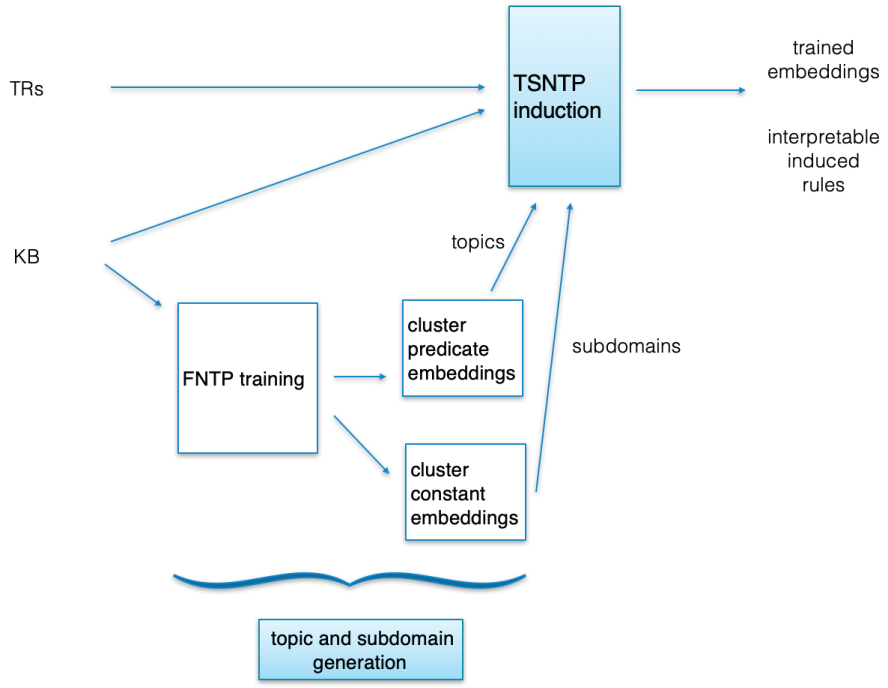

Fig. 4.1 An overview of *TSNTP* architecture. The *TSNTP* architecture differs from the *TNTP* architecture shown in Figure 3.2 in two respects: (i) the initial stage generates subdomains as well as topics, and (ii) in *TSNTP* induction each body literal matches with facts taking into account both topics and subdomains.

This chapter starts with a high-level method overview. Then, we highlight the challenges of implementing *TSNTP* and how we solved them.

4.1 Method Overview

Figure 4.2 shows a high-level computational tree of how the *TSNTP* induction uses facts partitioned by both topics on predicates and subdomains on the first arguments.

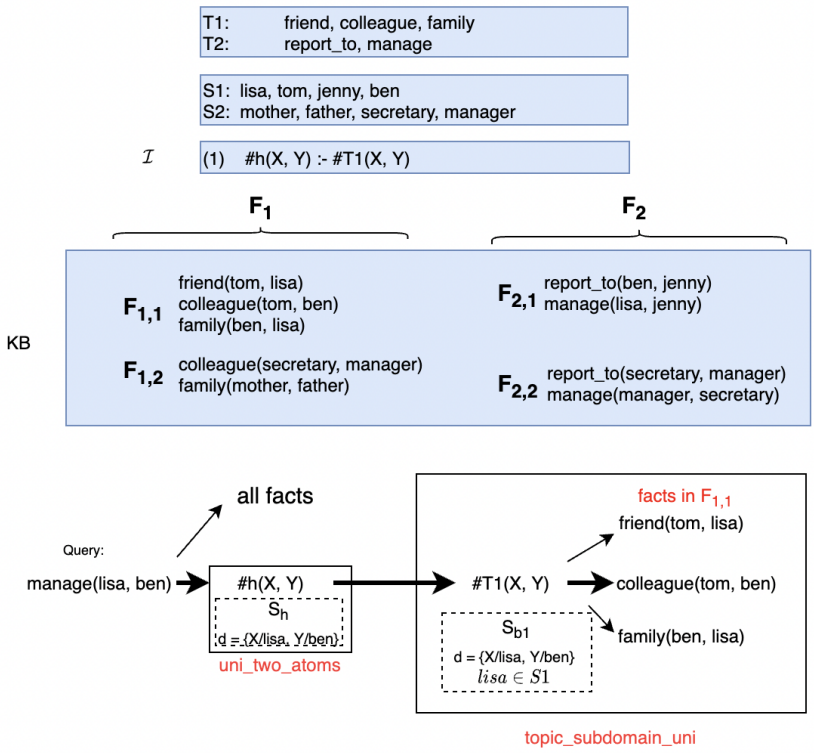

Fig. 4.2 An example *TSNTP* computational tree. The *TSNTP* receives the topics ($T1$ and $T2$) and subdomains ($S1$ and $S2$) generated by *FNTP*, and a set of *TTRs* ($\mathcal{I}$) (one copy of a *TTR* is used here). The *KB* is partitioned by the topic of the predicate and the subdomain of the first argument for each fact. As a result, in each body atom unification, *topic_subdomain_uni* is used, which selects facts in the matched topic and subdomain, improving the computation efficiency.

As shown in Figure 4.2, predicates are clustered into two topics (T1 and T2) and constants are clustered into two subdomains (S1 and S2). The topics and subdomains clustering is performed over the trained embeddings for predicates and constants generated using *FNTP*. All the (ground) facts are partitioned according to topic i of their predicates and subdomain j of their first argument. We denote with $\mathcal{F}_{i,j}$ all facts with predicate in topic i and first argument in subdomain j, with $\mathcal{F}_i$ all facts with predicates in topic i and with $\mathcal{F}_{Sj}$ all facts with first argument in subdomain j. In this way, when computing a body unification of a current body condition $\#Ti(X, Y)$, only facts whose predicate belongs to topic i and first argument belong to the subdomain of the current substitution of X will be used. Such restricted unification step is computed by the function *topic_subdomain_uni*, which essentially performs unification only between facts that have the given topic and first argument subdomain.

In *TSNTP*, facts are indexed by both topics for predicates and subdomains for the first argument. As a result, only a fraction of facts in a given topic is required for each body atom unification, making *TSNTP* on average more scalable than *TNTP*. Facts are selected based on the subdomain of one argument to enable a wider selection of facts. Otherwise, if facts are indexed by one topic and two subdomains, the selected facts might be too restricted for a normal size knowledge base and as a consequence, the soft-unification might not be learned well.

We introduce now the high-level architecture of *TSNTP*, including subdomain generation, computational tree construction and knowledge base partition.

4.1.1 Subdomain Generation

In the *TSNTP* architecture the first step is similar to *TNTP*, namely the fact-only training (*FNTP*). Subdomain generation happens after *FNTP* training and its results are used as the basis of the following *TSNTP* induction. Similar to topic generation, subdomains are computed by applying either the *K-means* or *Agglomerative* clustering algorithm (whichever is most appropriate) to the learned constant embeddings (with a reduced dimension d generated by *PCA*). This generates S clusters $\{\mathcal{S}_1, \mathcal{S}_2, ..., \mathcal{S}_S\}$. The subdomain of a constant c, denoted as $sub(c)$, is given by the cluster its embedding belongs to:

$$sub(c) = j \quad \text{iff} \quad \boldsymbol{\theta}_c \in \mathcal{S}_j \tag{4.1}$$

As result of the clustering each constant belongs to exactly one subdomain.

The subdomains are generated using Algorithm 4.1. After a given number of epochs, the *FNTP* step generates learned embedding $\boldsymbol{\theta}$ (line 5 in the algorithm), from which the embeddings of constants are extracted (line 7 in the algorithm). The subdomain clustering happens only once in *TSNTP*. The number of subdomains S can be specified directly or be selected by optimum Silhouette scores within a specified subdomain number range.

```
1 def TOPIC_SUBDOMAIN_GEN(facts, pred_ids, const_ids, train_data):
2     theta = init_embed(facts)
3     fntp = build_fact_only_tree(facts, theta)
4     for i in range(0, PRETRAIN_EPOCH):
5         theta = train(fntp, theta, train_data)
6     pca_theta = pca(theta, D_DIMENSION)
```

```
7   pred_theta, const_theta = divide_theta(pca_theta, pred_ids, const_ids)
8   topics = compute_clusters(pred_theta, T_TOPICS)
9   subdomains = compute_clusters(const_theta, S_SUBDOMAINS)
10  return topics, subdomains
```

Algorithm 4.1 A high-level algorithm of the topic and subdomain generation, using *FNTP*. Strings in capital letters are hyperparameters of the algorithm. This algorithm is similar to the topic generation algorithm 3.2, with the addition of line 7 and line 9 for computing clusters for constants.

The only difference between this algorithm and the topic generation algorithm 3.2 is the addition of lines 7 and 9, which, respectively, extracts the learned embeddings of the predicates and of the constants using their ids, and clusters the constants' embeddings into subdomains. Note that topics and subdomains are clustered separately because we only unify predicates with predicates and constants with constants. As a result, the embeddings of predicates and constants are independent. This decision is also justified by the observation that in all datasets, the trained embeddings of *FNTP* show a clear natural separation between the embeddings of predicates and constants in vector space, as shown in Figure 4.3 for the case of the *UMLS* dataset.

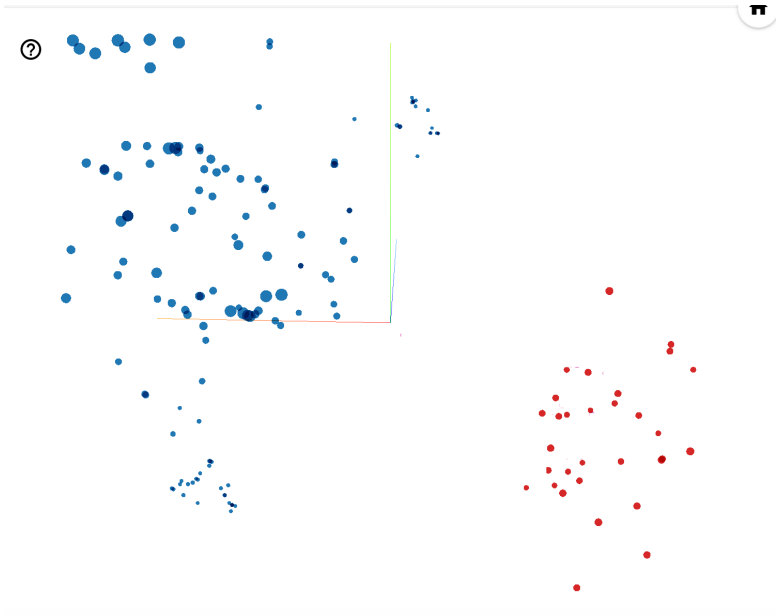

Fig. 4.3 The vector space of trained embeddings for the *UMLS* dataset learned by *FNTP*. The red dots are predicate embeddings, the blue dots are constant embeddings. This figure illustrates that the embeddings of constants and predicates are well separated in vector space.

The *UMLS* dataset contains 135 constants. The subdomain clustering has partitioned the constants into 10 subdomains. The list below presents an excerpt from the 10 subdomains. Subdomains reflect semantic similarity. For example, $S1$ contains constants about groups and $S2$ contains biological species. Such good clustering is due to the fact that *FNTP* training focuses on proving queries by unifying them just with facts going through many iterations

of symbolic unification. Symbols with similar meaning are brought closer and closer through backpropagation.

S1: population_group, age_group, professional_group, patient_or_disabled_group, ...

S2: amphibian, bacterium, invertebrate, human, vertebrate, animal, reptile , virus , fish , plant , mammal, bird, ...

S3: acquired_abnormality, experimental_model_of_disease, anatomical_abnormality, mental_or_behavioral_dysfunction, ...

Definition 4.1.1. Consider a knowledge base KB with signature $< \mathcal{P}, \mathcal{C} >$ composed of a set $\mathcal{P}$ of predicate embeddings, clustered in T topics, and a set $\mathcal{C}$ of constant embeddings, clustered in S subdomains. The set of facts $\mathcal{F}$ in KB is partitioned into sets $\mathcal{F}_{i,j}$ ($1 \leq i \leq T$ and $1 \leq j \leq S$), such that fact $p(a, b) \in \mathcal{F}_{i,j}$ if and only if the embedding of p is in topic cluster i (i.e. $topic(p) = i$), and the embedding of a is in subdomain j (i.e. $sub(a) = j$).

Note that only the subdomain of the first argument in each fact in $\mathcal{F}$ is used to partition $\mathcal{F}$. Otherwise, if facts are indexed by one topic and two subdomains, the selected facts might be too restricted for a normal size knowledge base and as a consequence, the soft-unification might not be learned well. A second reason is related to the run-time efficiency of constructing and traversing the computational tree, to avoid keeping track of the run-time substitution of variables in a *TTR* that only appear in body literals. (*TSNTP* uses the same *TTRs* as *TNTP*. See Section 3.3.2 for details.) Consider, for instance, the *TTR*: $\#H(X, Y)$:- $\#T1(X, Z), \#T3(Z, Y)$. Even if the partitions were created using the subdomains of both arguments when constructing the computational tree, the system could not determine which partition of facts to use when applying the *topic_subdomain_uni* function over $\#T1(X, Z)$. This is because Z will not be bound with a constant when proving the body atom. This problem will be discussed in more detail in Section 4.2.

4.1.2 Computational Tree Simplification

Given the topic-subdomain partition of the facts $\mathcal{F}$ of a knowledge base, the body atom unification process in *TSNTP* can be further simplified. In this section, we firstly present the aspects of *TSNTP* that are similar to *TNTP*, and then introduce the simplification made in *TSNTP*.

The high-level proof mechanism for *TSNTP* is the same as that in *TNTP*, as shown in Figure 4.2. In order to prove a query of the form $q(a, b)$, each fact branch and rule branch generates a

branch proof score and the highest proof score is selected as the proof score for $q(a, b)$. The proof score of a rule is the minimum score among its head atom unification score and all of its body atom unification scores.

TSNTP optimises body atom unification in the construction of the computational tree, by selecting a subset of facts that match both the topic and the subdomain of the first argument. To facilitate this unification process, all facts in a given knowledge base are partitioned into sets $\mathcal{F}_{i,j}$ where i is the topic clustering index of the predicate and j is the subdomain clustering index of the first argument of the facts in $\mathcal{F}_{i,j}$. As a result, given a body atom $\#Ti(X, Y)$ where X and Y are variables and X is bound to a constant c of subdomain j, such atom will only unify with facts in $\mathcal{F}_{i,j}$. If X were not bound to a constant, such fact would unify with all facts in topic i (i.e. all facts in $\mathcal{F}_i$).

We present two cases to demonstrate this optimisation in the body unification process. In the first case, consider a query $q(a, b)$ and a copy of a *TTR* of the form: $\#H(X, Y)$:- $\#Tn(X, Y)$, where X is bound to a with subdomain $sub(a) = j$. The facts used for the unification of $\#Tn(X, Y)$ are then only from the set $\mathcal{F}_{n,j}$, instead of all facts in topic n as done in *TNTP*. If facts in the knowledge base are fairly distributed by the topic and the subdomain of the first argument, then this reduces the computation of each body atom by approximately $1/S$ (where S is the number of subdomains). In the second case, consider a copy of a *TTR* of the form $\#H(X, Y)$:- $\#Tn(Z, X), \#Tm(Z, Y)$. Then Z would be unbound when the body atom $\#Tn(Z, X)$ is being proved. In such a case, this body atom will unify with all facts in topic n (falling back into the *TNTP* unification process for this particular body atom proof)[1].

4.2 Challenges in Implementing TSNTP

Although the methodology builds upon *TNTP*, the implementation of *TSNTP* had to be changed in order to include the extra subdomain processing. This section highlights challenges that we have overcome in the *TSNTP* implementation. These challenges include subdomain information in template rules, imbalanced distribution of facts over subdomains and dynamic binding of subdomains in batch processing.

Subdomain in *TTRs*. In the previous chapter, we have seen that the introduction of topics has led to a new notion of template rules, that is *TTRs*, where topics of learnable predicates are

[1]This situation will be discussed in detail in Section 4.3

explicitly indicated in the template rules. It is natural to ask ourselves whether subdomains would also need to be explicitly included in the definition of rule templates. Consider for example the following copies of *TTRs*:

```
#H(X, Y) :− #T5(X, Y).
#H(X, Y) :− #T3(Y, X).
#H(X, Y) :− #T2(X, Z), #T1(Z, Y).
```

If subdomain information were specified in each *TTR*, the number of copies of each *TTR* would increase dramatically. Assuming for instance $\mathcal{S}_1$, $\mathcal{S}_2$,...,$\mathcal{S}_S$ subdomains and 3 copies of the first *TTR* as given above, we would need to replace each copy of $\#H(X,Y)$:- $\#T5(X,Y)$, with multiple copies, one for each subdomain. For instance, specifying S possible subdomains for argument X in the *TTR* would require $3 \times S$ copies of the *TTR* (instead of 3 copies) of the form[2]:

```
#H(X, Y) :− #T5(#S1X, Y).
#H(X, Y) :− #T5(#S2X, Y).
 ...
#H(X, Y) :− #T5(#SSX, Y).
```

To maintain the simplicity of template rules, *TSNTP* uses the same format of *TTRs* as in *TNTP*, without subdomain information. Instead, subdomain information is determined dynamically as described below.

Imbalanced Fact Distribution. The subdomain generation can cluster constants with similar semantic meaning together effectively, but it cannot guarantee that facts are evenly distributed among the subdomains.

Recall that during the subdomain generation, S subdomains are created, and each subdomain contains various number of constants. Since clusters are generated by running clustering algorithm on embeddings, there is no control on the distribution of facts, except the number of subdomains S. Sometimes, this imbalance is caused by the knowledge base itself. Although the imbalanced distribution of facts might be ameliorated by selecting the number of subdomains carefully, the size of each set $\mathcal{F}_{i,j}$ of the partition depends on both the distribution of topics and the distribution of subdomains. For instance, some $\mathcal{F}_{i,j}$ groups may be very small, because there are few facts in topic i (regardless of subdomain j), few facts in subdomain j, or a combination

[2]Note that subdomains would not need to be specified next to the arguments of head predicates since these arguments are always immediately unified with the constants that appear in the given queries.

of both.

Despite the potential imbalanced distribution of facts, *TSNTP* is still a more scalable version than *TNTP*, because it uses only a fraction of topical facts in each body atom proof.

Dynamic Binding of Subdomains in Batch Processing. Unlike topics, which are given by the *TTRs* before the construction of computational trees, subdomains of variables need to be determined dynamically after variables have been grounded, using a function $sub(c)$ which returns the subdomain for a given constant c. These subdomains, which essentially cluster constants with similar semantic meaning, can be used to group the facts in a given *KB* and filter the unification process in the construction of a computational tree, even further than what *TNTP* does (based just on topics). However, constructing computational trees in the same way as in *TNTP* and imposing subdomain constraints in the unification process would cause computational inefficiency, caused by operations over sparse matrices, even though the number of facts to be unified would be reduced. We illustrate this point in what follows.

Although checking the subdomain of a constant is not complicated, proving a group of atoms with different subdomains as arguments, at the same time, is complicated. This situation happens when batch processing is involved. Batch processing is normally used to speedup training, meaning that a batch of queries (usually 50 queries per batch) are proved together in a computational tree. Batch processing applies to each step of a computational tree construction including, but not limited to the steps uni_s, $unify\_two\_atoms$ and $topic\_based\_unification$ that we have presented in the previous chapter. Let's revisit the computational tree construction of *TNTP*. Consider, for instance, a batch of 50 queries, as indicated in Figure 4.4.

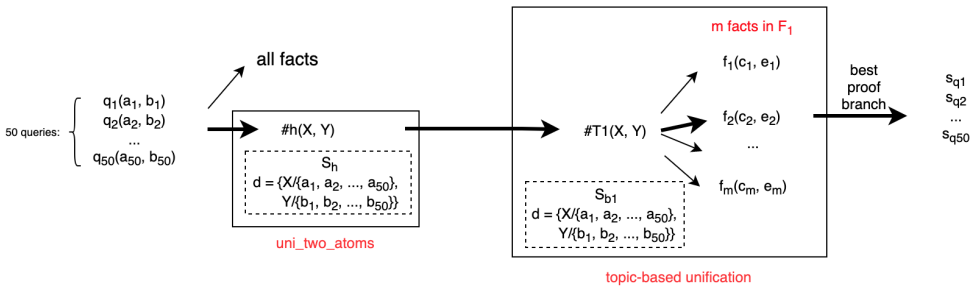

Fig. 4.4 An example computational tree in the batch processing mode in *TNTP*. A query batch with 50 queries are given as inputs of the computational tree. They are computed at the same time at each step of the computation tree. In each atom unification, 50 independent substitution sets are generated and 50 atom proof scores are computed using each substitution set respectively. In the end, 50 proof scores are returned, one for each query.

The unification of the 50 queries with the head atom $\#h(X, Y)$ generates a vector of 50 proof scores, each computed as uni_s (symbol-symbol unification) between the embedding of a query predicate and that of $\#h$, and 50 substitutions for the variables X and Y respectively. The 50 proof scores are denoted as S_h and the substitutions as d in Figure 4.4. Then, the body atom is proved using d. Considering the body atom $\#T1(X, Y)$ with a specific substitution for X and Y, the *TNTP* approach uses the topic-based unification, which unifies $\#T1(X, Y)$ only with facts whose predicates belong to topic 1 ($\mathcal{F}_1$). Let's assume that $|\mathcal{F}_1| = m$. The unification of $\#T1(X, Y)$ with each of the m facts is then performed pairwisely (predicate with predicate, first argument with first argument, and second argument with second argument) and a body atom unification score S_{q_i} is generated corresponding to each initial query q_i in the batch. This is performed for each of the 50 queries in the batch, so the unification for $\#T1(X, Y)$ generates a matrix of size $50 \times m$, where each row i captures the unification of $\#T1(X, Y)$ with the m facts in $\mathcal{F}_1$, corresponding to proving query q_i in the batch. What is important to notice here is that because the filter of facts in KB is only based on the topic of the predicate, then for every query in the batch, the number of facts considered for the unification of $\#T1(X, Y)$ is fixed and given by $|\mathcal{F}_1|$. This is not the case when we control the unification in terms of subdomains, as this will depend on the number of facts in KB that have arguments belonging to the same subdomain of the arguments given in the query.

Consider one of the queries $q_i(a_i, b_i)$ in a given batch. In *TSNTP*, the unification of the body atom $\#T1(X, Y)$ chooses the facts from KB that have predicates belonging to the topic 1 and among these facts, chooses only those that have first argument belonging to the same subdomain cluster as that of the constant a_i used in the current substitution of X. As different queries in the batch may use different constants as first argument, the number of facts in KB chosen based on the subdomain is not the same across different queries in the same batch, causing the unification process for the atom $\#T1(X, Y)$ to generate 'sparse matrices'. An example of the uneven selection of constant unifications for the first argument of queries in a given batch is depicted in Figure 4.5. Constants c_1,..., c_m are the arguments of the facts that appear in F_1 when unifying $\#T1(X, Y)$ and constants a_1,..., a_{50} are the 50 constants that appear as first argument in the 50 queries of a given batch. So, for substitution X/a_1 the unification of $\#T1(X, Y)$ would consider only facts in $\mathcal{F}_{1,1} \subseteq \mathcal{F}_1$ which have predicates belonging to topic 1 and first arguments belonging to subdomain 1, where $S1 = sub(a_1)$.

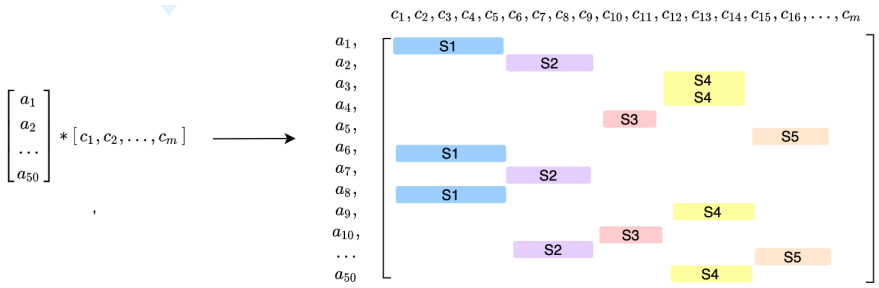

Fig. 4.5 An example unification between 50 groundings of a variable 'X' ($[a_1, a_2, ..., a_{50}]$) and constants in the first argument of selected facts ($[c_1, c_2, ..., c_m]$). The highlighted area in the matrix indicates the items that are considered for unification based on subdomains. The non-highlighted area in the matrix are unifications that need to be ignored, due to unmatched-subdomains. The $*$ represents soft-unification between two embeddings, which is computed using the RBF kernel as explained in Section 3.1.2. The highlighted area applies to each operator (i.e. multiply, subtraction) in the RBF kernel to simplify the computation, where for each operator only the highlighted parts are used for computation. This matrix can be represented by a sparse matrix.

To avoid computations with sparse matrices (which are inefficient in *Tensorflow*[3]), *TSNTP* orders the entire KB by topics of the predicates, and within each of these groups, facts are ordered by the subdomain of the first argument. So, if for a given KB we have T topics and S subdomains, the ordered KB will be $KB^* = \{\mathcal{F}_{1,1}, ..., \mathcal{F}_{1,S}, \mathcal{F}_{2,1}, ..., \mathcal{F}_{2,S}, ..., \mathcal{F}_{T,1}, ..., \mathcal{F}_{T,S}\}$. The ordering essentially induces a partition over the initial KB. Similarly, query batches are formed so that queries in the same batch have their two arguments belonging to the same respective subdomains. As a result, given a batch with 50 queries grouped in this way, the substitution of X would be with constants that belong to the same subdomain, say $\mathcal{S}_j$. Therefore the 50 body atom unifications of $\#Tn(X, Y)$, with respect to the different 50 groundings of X, will all refer to facts from the group $\mathcal{F}_{n,j} \subseteq KB^*$. The matrix unification process is in this case simplified dramatically, as facts will be selected only from the matched subdomain $\mathcal{F}_{n,j}$ for all queries in the same batch. For example, the unification for the first argument of the 50 queries given in a batch shown above in Figure 4.5, can be computed in *TSNTP* using a subdomain specific matrix as illustrated in Figure 4.6, instead of using sparse matrices.

[3]In *https://github.com/tensorflow/tensorflow/issues/46706*, other users of Tensorflow also identified the same problem and it is reproduceable. For example, using the division operator for a dense matrix takes 0.0625s, whereas it takes 0.5718s for a sparse matrix.

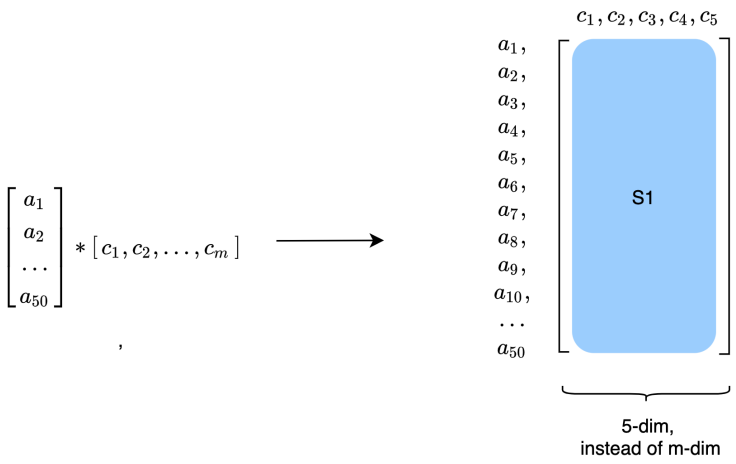

Fig. 4.6 If all groundings of X ($[a_0, a_1, ..., a_n]$) belong to the same subdomain, the big unification matrix of Figure 4.5 can be replaced by the smaller unification matrix shown here, ignoring unmatched subdomains. This simplification can improve computational efficiency.

4.3 The *TSNTP* Solution

In order to process queries in batches efficiently and avoid the computation challenges caused by sparse matrix computation, *TSNTP* restricts both arguments of queries within a query batch to belong to the same respective subdomain. This is achieved by shuffling queries in the *KB* according to subdomains of both arguments. With this restriction, for each query batch *TSNTP* could prove a body literal with a subset of facts in a given topic and subdomain, assuming that their first arguments belong to the same subdomain. Subdomains are used to shuffle queries with respect to both their arguments. This allows the restricted unification to be applied also to *TTRs* such as $\#H(X, Y)$:- $\#T2(Y, X)$. In that case, the subdomain of Y determines which $F_{i,j}$ to choose when unifying the body atoms $\#T2(Y, X)$. This section presents the main features of *TSNTP*: shuffled query generation, simplification of body atom unification with partitioned query batches, computational tree and induction algorithm.

Shuffled Query Generation. An important feature of *TSNTP* is that queries are grouped according to their subdomains. Same as *TNTP*, each epoch of training uses a different query set, which is generated by corrupting facts in *KB* randomly using the closed-world assumption. In *TSNTP* the query set is also partitioned.

Definition 4.3.1. Let *KB* be a knowledge base with $\mathcal{S}$ subdomains, $\mathcal{S}_1$, $\mathcal{S}_2$,...,$\mathcal{S}_S$, and let Q be a query set. The shuffled query set $Q^* = \{Q_{s1,s1}, ..., Q_{s1,sS}, ..., Q_{sS,s1}, ..., Q_{sS,sS}\}$ is the set of queries where $Q_{si,sj}$, includes all queries from Q whose first argument embedding belongs to

subdomain $\mathcal{S}_i$ and second argument embedding belongs to subdomain $\mathcal{S}_j$. For each query $q(a, b)$ in Q, there is a unique si and sj such that $sub(a) = \mathcal{S}_i$ and $sub(b) = \mathcal{S}_j$ and $q(a, b) \in Q_{si,sj}$.

During training, fixed sized batches of queries are created from the set Q^* so that all queries in a batch belong to one group $Q_{si,sj}$. To maintain a fixed batch size, queries may be duplicated in some batches. In this way the fact unification in a branch can be simplified. Given that all queries in a batch belong to a group $Q_{si,sj}$, the fact branch of a *TSNTP* computational tree can unify just with facts whose first argument embedding belongs to $\mathcal{S}_i$, meaning all facts in $\mathcal{F}_{si}$. This is in contrast to *TNTP*, where the fact branch unifies with all facts of a given *KB*. Note that we do not use both subdomains to cut the number of fact branches, because this might cut down too many facts. Selecting facts according to both subdomains could be applied to very large datasets but we leave this as part of our future work.

Body Atom Unification. The shuffled query set and the partition of facts in a given *KB* can be used to define a simpler body atom unification process.

Definition 4.3.2. Let *KB* be a knowledge base with a partitioned $\mathcal{F}^*$ set of facts, and let Q^* be a shuffled set of queries. Let $\#Ti(V1, V2)$ be a body condition of a *TTR* to be proved in a computational tree, where $V1$ is substituted by a constant a whose embedding belongs to subdomain j. The *TSNTP* body atom unification, denoted *topic_subdomain_uni*, computes the unification between $\#Ti(V1, V2)$ and facts in $\mathcal{F}_{i,j}$.

Consider for instance the following set of *TTRs*:

```
#H(X, Y) :- #T1(X, Y).
#H(X, Y) :- #T1(Y, X).
#H(X, Y) :- #T1(X, Z), #T2(Z, Y).
```

The first variable of each body atom appears in the head atom. Since all queries in a batch have first argument's embedding belonging to the same subdomain, it is safe to apply the *TSNTP* body unification given in Definition 4.3.2 above. However, if the first variable of a body atom does not appear in the head atom, as it is the case for the body atom $\#T2(Z, Y)$, the substitutions for Z would depend on the unification of the previous body literal and may well be different across the different unifications of $\#T1(X, Z)$. Hence the body unification for $\#T2(Z, Y)$ has to be the one used by *TNTP*. So for the third *TTR* given above, the two body atoms are proved in two different ways: $\#T1(X, Z)$ is proved using *topic_subdomain_uni* whereas $\#T2(Z, Y)$ is proved using $topic\_based\_unification$.

Computational Tree. We present now an example of computational tree generated by *TSNTP*. We consider a *TTR* rule with transitivity, and a partitioned set of facts F^* generated from two topics $T1$ and $T2$ and two subdomains $S1$ and $S2$. The shuffled batch of queries shown in Figure 4.7 includes queries from $Q_{s1,s1}$. So the body unification of the first body condition is performed by *topic_subdomain_uni* which unifies $\#T1(X, Z)$ with all facts in $\mathcal{F}_{1,1}$. The body unification for the second body condition is instead performed by *topic_based_unification*, which unifies $\#T2(Z, Y)$ with all facts in $\mathcal{F}_2$. Figure 4.7 illustrates a computational tree generated by *TSNTP*, using a copy of a *TTR*.

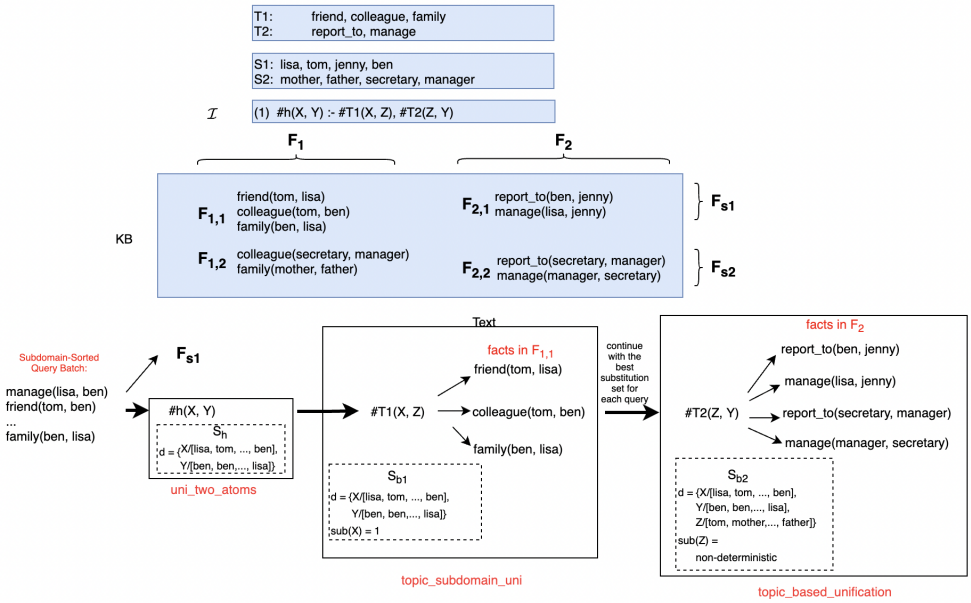

Fig. 4.7 A *TSNTP* computational tree for a transitive *TTR* rule. The knowledge base is partitioned according to topics ($T1$ and $T2$) and subdomains ($S1$ and $S2$). F_{t_s} represent a partition of the *KB* with the topic t and the subdomain of the first argument s. F_i represents all facts in topic i and F_{sj} represents all facts whose first argument is the subdomain j. For a query batch with the same subdomain combinations, the computational tree consists of topical template rules and facts in the matched subdomain (i.e. F_{s1} is used for fact branches, because topics are variables in the query batch due to shuffling). In a body atom, such as $\#T1(X, Z)$, where all groundings of X have the same subdomain, *topic_subdomain_uni* is used to prove this body atom, using facts with the matched topic and subdomain. For a body atom, such as $\#T2(Z, Y)$, *topic_based_unification* is used in the proof, because groundings of Z would be different for queries in the batch.

4.3.1 *TSNTP* Training Algorithm

The high-level algorithm for *TSNTP* is summarised as follows:

```
1 def TSNTP_induction(facts, rules, train_data, topics, subdomains):
2     theta = init_emb(facts, rules)
3     ttrs = build_ttr(rules, topics)
```

```
4    facts = partition_facts(facts, topics, subdomains)
5    train_data = shuffle_queries(train_data, subdomains)
6    tsntp = build_topic_subdomain_computational_tree(facts, ttrs, theta)
7    for i in range(0, TRAIN_EPOCH):
8      theta = train(tntp, theta, train_data)
9    induced_rules = decode_rules(theta, ttrs)
10   return theta, induced_rules
```

Algorithm 4.2 The *TSNTP* induction algorithm. This algorithm differs from the *TNTP* induction algorithm in line 4-6, where facts are partitioned by both topics and subdomains, queries are shuffled by subdomains and a topic and subomain computational tree is used.

TSNTP receives as input a *KB*, a set of templates rules, a training set, and a set of topics and subdomains generated by Algorithm 4.1. This algorithm is similar to that of *TNTP* shown in Algorithm 3.3, except for lines 4-6, where the partitioned $\mathcal{F}^*$ set of facts is generated together with the shuffled Q^* of the query set. The computational tree is then constructed as described in Section 4.3, and trained through gradient descent. The scaling factors and decoding mechanism introduced in *TNTP* applies to *TSNTP* as well.

Summary. *TSNTP* uses topics and subdomains to narrow down the number of facts involved in each body atom unification when traversing the computational tree. In particular, in *topic_subdomain_uni* only a subset of facts $\mathcal{F}_{i,j}$ for a given topic i and subdomain j is used and in *topic_based_unification* only a subset of facts $\mathcal{F}_i$ for a given topic i is used. Note that these subsets are known when constructing the computational tree. This simplification is supported by a shuffle of query batches, which ensures that queries in a given batch have the same subdomain combination for their argument embeddings, and can therefore be processed together. Our experiments demonstrate that *TSNTP* has an improved computation time by at least one order of magnitude (depending on the subdomain number and constant distribution) compared to *TNTP*, while maintaining similar accuracy on the same benchmark datasets. The evaluation of this approach is presented in Chapter 7.

Chapter 5

Negation-as-Failure TSNTP

In the previous chapter we have shown how our *TNTP* framework can be made computationally more efficient by means of subdomain clustering. However, *TNTP* and *TSNTP* can only learn definite rules, which are not expressive enough for representing commonsense. The system, *Negation-as-failure TSNTP* (*NAF TSNTP*), building upon the *TSNTP*, is a first step towards learning first order normal rules.

We take inspiration from the work 'Exception-enriched Rule Learning from Knowledge Graphs' [Gad-Elrab et al., 2016] and propose a method that induces normal rules in two phases. The first phase, consisting of *FNTP* and *TSNTP* induction, induces a set of definite rules. The second phase revises these learned definite rules by adding potential negated atoms and selects the best of such revisions according to a quality function that evaluates their effectiveness in terms of improved accuracy. An assumption is that only learned rules are revised. An overview of the *NAF TSNTP* architecture is given in Figure 5.1.

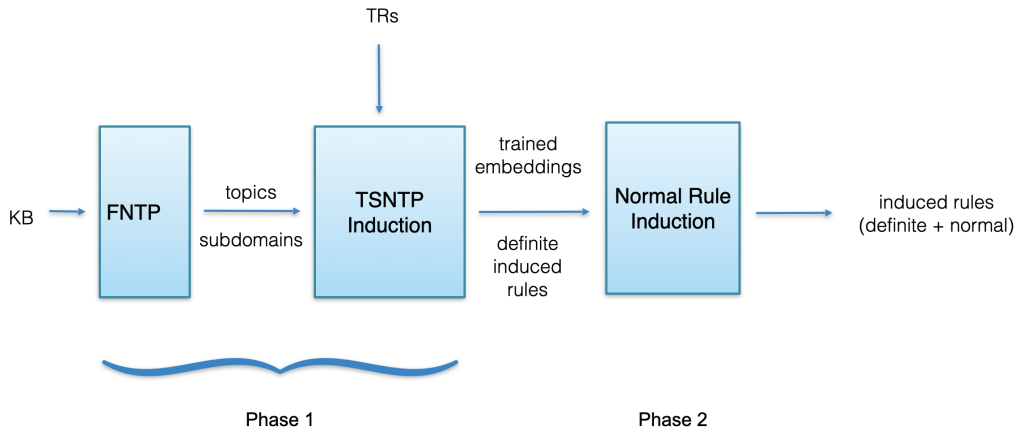

Fig. 5.1 An overview of *NAF TSNTP* architecture. The *NAF TSNTP* consists of two phases. The first phase is the same as *TSNTP*, where a knowledge base (*KB*) is given as input to train *FNTP*, which are then used to create topics and subdomains for *TSNTP* induction. The second phase receives the output of the *TSNTP* induction as inputs, including trained embeddings and definite rules. The normal rule induction phase then creates and selects a set of normal rules to replace the definite rules, generating a mixture of definite and normal induced rules.

In the following sections, we firstly present a summary of [Gad-Elrab et al., 2016] and then introduce the mechanism of *NAF TSNTP*, including the *NAF TSNTP* pipeline, implementation details and a high-level algorithm.

5.1 Exception-enriched Rule Learning from Knowledge Graphs

We summarise here the work in [Gad-Elrab et al., 2016], which has inspired our two-step method for inducing normal rules. Similar to our *TSNTP* system, they also assume body literals only match with facts, not other rules, but in their case they restrict the signature to unary predicates.[1] Their method uses a symbolic approach, which they describe as 'a method for effective revision of learned Horn rules by adding exceptions (i.e., negated atoms) into their bodies', where negated atoms are created using predicates in the signature. For example, a negated atom 'not rainy(X)' could be appended to a Horn rule $goodWeather(X)$:- $sunny(X)$, forming a more specific normal rule $goodWeather(X)$:- $sunny(X)$, $not\ rainy(X)$. They propose a 2-phase approach, where definite Horn rules are first learned and then rule bodies are refined with appropriate negated atoms. To select normal rules that improve data prediction among all potential normal rules, they use a rule ranking measure to evaluate the performance of normal rules. The high-level architecture of their approach is summarised in Figure 5.2.

[1]Some unary atoms are reified binary atoms and others, used in their exception predicates capture exception individuals.

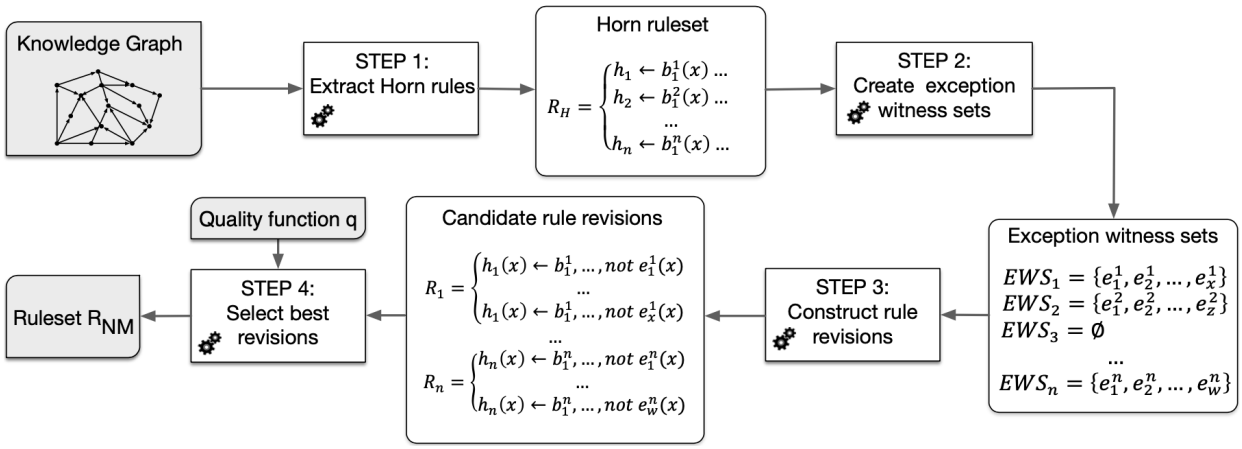

Fig. 5.2 The high-level architecture of 'Exception Enriched Rule Learning' from [Gad-Elrab et al., 2016]. The architecture consists of four steps. In the first step, a set of Horn rules are selected from a knowledge graph. Then, an exception witness set is created for each Horn rule, including the set of possible negated predicates for the rule. In step 3, a set of rules is revised, where all possible negated atoms generated by the exception witness set are appended to each Horn rule respectively. In step 4, the revised rule set for each Horn rule is refined, which selects the best normal rule using a quality function.

The architecture in Figure 5.2 consists of four steps. In Step 1, definite Horn rules are extracted from a given knowledge base represented as a knowledge graph. This extraction method could be performed by any rule mining algorithm, either symbolic or neural, that can return symbolic rules. In their work, the symbolic algorithm *FPGrowth* is used. In Step 2, for each rule r, an exception witness set EWS is created, denoted as $EWS(r, \mathcal{A})$, where $\mathcal{A}$ is the set of facts in the knowledge graph. Assuming a rule has the form $a(X) \leftarrow b_1(X), ..., b_k(X)$, $EWS(r, \mathcal{A})$ consists of a maximal set of unary predicates, such that each predicate e_i in $EWS(r, \mathcal{A})$ satisfies two criteria: (1) there is at least one fact of the form $e_i(c')$ in $\mathcal{A}$, where $a(c')$ can be deduced from r, but it is not in $\mathcal{A}$; (2) for all constants c where $a(c)$ can be deduced from r and $a(c)$ is in $\mathcal{A}$, $e_i(c)$ is not in $\mathcal{A}$. The first criterion ensures that there is at least one exception where $a(c)$ is true according to r, but not in the knowledge graph. This scenario indicates that r might be too general and appending a negated atom might make it more specific. The second criterion prevents the situation that a fact in the knowledge graph, which is also deduced from r, would no longer be deduced after appending the negated atom to r. If a fact can be proved by multiple rules, each rule needs to satisfy the second criterion. In Step 3, it constructs candidate rule revisions for each Horn rule by appending a negated atom with predicates in its EWS. Because of unary atoms, a negated unary predicate captures an exception individual, even though they use the term exception predicates. In step 4, it selects the best rule revisions according to a quality function. Finally, a rule set R_{NM} is generated with both definite rules and normal rules.

Figure 5.3 illustrates an example of normal rule induction using this architecture.

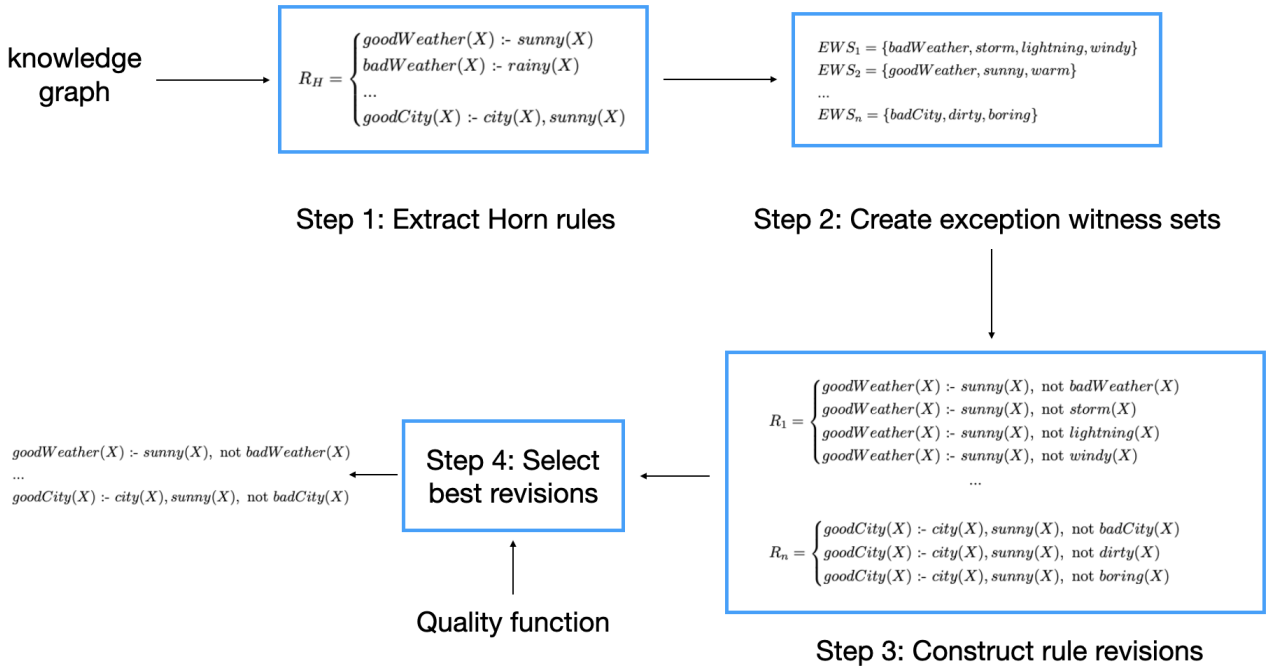

Fig. 5.3 A hypothetical example of 'Exception Enriched Rule Learning'. In step 1, a set of Horn rules is extracted from the knowledge graph. In step 2, an exception witness set is created for each Horn rule, which includes its potential negated predicates. In step 3, each predicate in the exception witness set of a Horn rule is appended to the Horn rule independently. Step 4 selects the best normal rule for each Horn rule using a quality function, which generates a set of output normal rules.

The method of [Gad-Elrab et al., 2016] learns nonmonotonic rules under the open-world assumption (OWA) that assumes an atom that is not in the knowledge graph could be simply missing. This is different from our system which employs the closed-world assumption, namely CWA, that assumes an atom that is not in the knowledge base to be false. In [Gad-Elrab et al., 2016], exceptions, in terms of negated atoms, can be viewed as a theory revision under OWA. It uses a quality function q to quantify whether a normal rule is 'better' than its definite part, by measuring the improvement on prediction accuracy. It proposes a few quality functions. However, none of these functions offer any guarantee on correctness. It could only ensure that given the current knowledge graph, these normal rules represent the knowledge graph better than definite rules, but this might not hold if more facts are added to the knowledge graph, due to the non-monotonic assumption used in this paper.[2]

The negated atoms need to be selected carefully, because extending a definite rule to a normal rule is not always beneficial. By adding negated atoms to definite rules, the number

[2]In our work, under the closed-world assumption, our knowledge base is fixed, so we do not have this problem.

of false positive predictions could be reduced. In other words, the facts deduced by normal rules are a subset of facts deduced by their definite parts. On the other hand, the negated atom could increase the number of false negative predictions. If a wrong negated atom is added, some facts that were deduced by the definite rule might not be deduced any more by the normal rule. As a result, the quality function is important to identify suitable negated atoms.

The work in [Gad-Elrab et al., 2016] makes some further assumptions which are different from those made in our work. In particular, it is based on symbolic rules with hard-unification, so it cannot exploit valuable semantic information gained through *TSNTP* training. They assume different predicates have distinct meanings and a predicate is either in *EWS* or not in it, without a middle ground, so they cannot take the similarity between different symbols into account. Also, it supports unary atoms only, whereas binary atoms are used in most of knowledge base and knowledge graph applications. Use of hard-unification and the unary signature make it hard to be incorporated to our system directly.

5.2 Normal Rule Induction

As mentioned above, our *NAF TSNTP* system induces normal rules also in two phases. The first phase uses *TSNTP* to train embeddings and to induce definite rules. The second phase focuses on inducing normal rules by extending the definite rules with negated atoms. This section focuses on the normal rule induction phase. We firstly present our normal rule induction pipeline, the syntax of normal rules, and how we support normal rules in a computational tree.

5.2.1 Normal Rules Induction Pipeline

Figure 5.4 illustrates the pipeline of the normal rule induction phase in *NAF TSNTP*.

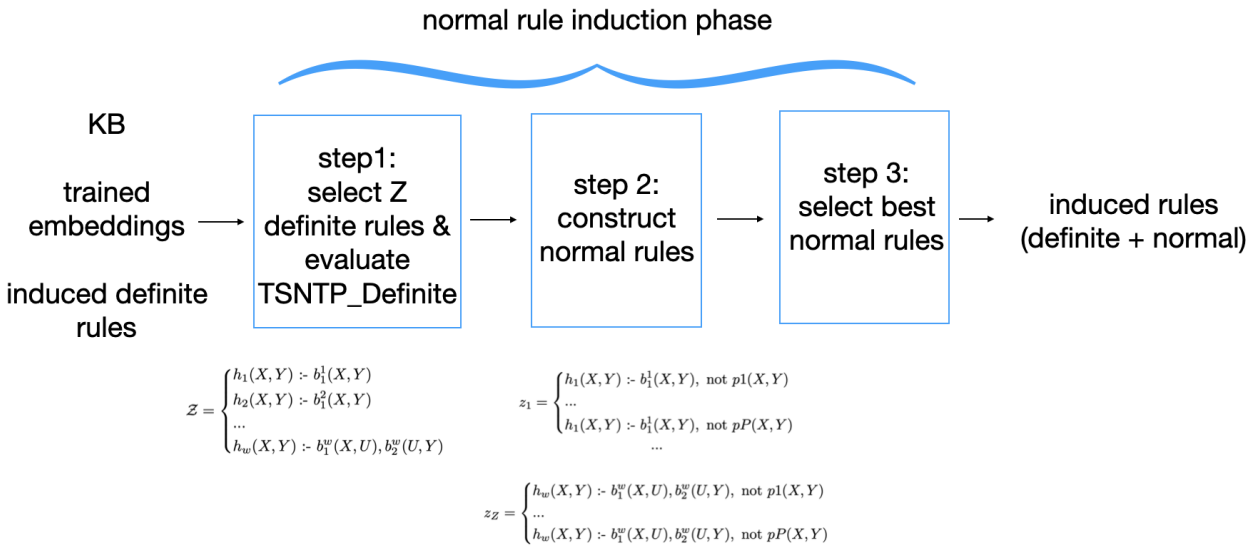

Fig. 5.4 The normal rule induction pipeline of *NAF TSNTP*. The pipeline starts with the outputs of *TSNTP*, namely the *KB*, trained embeddings and induced definite rules. In the first step, Z definite rules are selected from induced definite rules by inspecting the frequency that a rule is used to prove positive queries. In the second step, a set of potential normal rules is created for each rule in the Z definite rules by appending negated atoms created using predicates in the *KB*. In step 3, a best normal rule is selected for each definite rule. In the end, a mixture of definite and normal rules is induced.

As illustrated in Figure 5.4, the normal rule induction starts with a *KB* and the outputs of the *TSNTP* training, including trained embeddings and induced definite rules. The normal rule induction phase consists of three steps and includes no additional embedding training. In Step 1, the Z most frequently used definite rules of corrupted queries are selected from the given induced definite rules to form a set of definite rules, denoted by $\mathcal{Z}$ ($\{z1, ..., zZ\}$). Recall that the set of P known predicates is denoted by $\mathcal{P}$, each zi in $\mathcal{Z}$ is augmented P times, each time with a different possible negated atom from $\mathcal{P}$, thus forming P number of normal rules. This step generates $Z \times P$ normal rules in total. For the efficiency of computation, only one negated atom is added to each induced Horn rule. Otherwise, the number of potential normal rules would grow exponentially, e.g. adding two negated literals would result in $P \times P$ normal rules being created from each Horn rule. In Step 3, all potential normal rules of each selected definite rule zi are evaluated by a quality function ns and the best normal rule of each zi is selected. ns provides a way to quantify the improvement in prediction accuracy for each query, by checking the difference between the proof score generated by the definite rule zi and its extension. If its evaluation score is better than a threshold, it can replace zi. For the efficiency of computation, only one normal rule is selected for each definite rule. If multiple normal rules are generated for each Horn rule, the final computational tree could potentially contain many rule branches. As a result, when this trained computational tree is used to

prove queries, it would not be as computationally efficient as the computational tree with one normal rule per Horn rule. Similarly, for the efficiency of computation, only one negated atom is selected for each definite rule. Note that soft-unification still holds for negated atoms, so the predicate of a negated atom could represent multiple similar predicates. The final output is a set of induced normal rules, which is a mixture of definite rules and selected normal rules.

As shown by Figure 5.4, *NAF TSNTP* has a similar high-level architecture to that presented in [Gad-Elrab et al., 2016]. The 'extract definite rules' step in Figure 5.2 is equivalent to *TSNTP* training plus top-Z definite rules selection (our step 1). We do not have the 'create exception witness set' step, because we use all predicates directly and leave the selection to step 3. We have the same 'construct rule revisions' step and 'select best revisions' step, except that our quality function supports soft-unification.

5.2.2 The Syntax of Normal Rules

In order to induce normal rules that support soft-unification, the computational tree of *NAF TSNTP* needs to support the syntax of normal rules and compute proof scores of negated atoms. In this section, we introduce the syntax of normal rules and in Section 5.2.3, we explain how to compute proof scores of normal rules in a computational tree.

Recall that in *TSNTP*, *TTRs* are definite rules of the form:

```
#H(X, Y) :− #T5(X, Z).
#H(X, Y) :− #T3(X, Z), #T4(Z, Y).
```

where each induced predicate $\#Ti$ represents a predicate in topic i and has its own trained embedding, independent from the embeddings of known predicates.

In *NAF TSNTP*, a normal rule consists of a definite rule augmented with a negated atom of a known predicate (i.e. $p21$, $p32$), such as in the rules below. As the negated literal is added after the completion of the *TSNTP* training, it is the embedding of each predicate that is used (not the English symbol).

```
#H(X, Y) :− #T5(X, Z), not p21(X, Y).
#H(X, Y) :− #T3(X, Z), #T4(Z, Y), not p32(X, Y).
```

Note that, unlike induced predicates with '#', the predicate in each negated atom is known. As they are added once the training of *TSNTP* is complete, these negated (known) predicates can support soft-unification in the same way as positive predicates.

5.2.3 Soft-unification for NAF Literals

We next describe the extension of soft-unification to negated atoms. As explained in Section 5.2.2, to capture NAF, the computational tree is extended to compute the proof score of each negated atom in a way that captures negation-as-failure. Whenever a negated atom is encountered, the proof score of the atom is computed using the function *neg_ts_atom_uni* defined in Definition 5.2.1, instead of the normal *topic_subdomain_uni*.

Definition 5.2.1. Given a negated atom in the form of *not a*, where *a* is an atom, *neg_ts_atom_uni(a)* defines the proof score of '*not a*' as $1 - topic\_subdomain\_uni(a)$.

The score calculation defined in Definition 5.2.1 captures the semantics of negation-as-failure. In Logic programming, 'not a' is true when there is no fact or rule in the knowledge base that can deduce 'a'. In our system, under soft-unification, the proof score of 'a' indicates the 'chance' that 'a' is true in the knowledge base. That is, if 'a' has a high proof score in *topic_subdomain_uni*, 'not a' would be close to zero and vice versa. Figure 5.5 demonstrates two common scenarios under *NAF TSNTP* when proving a negated atom.

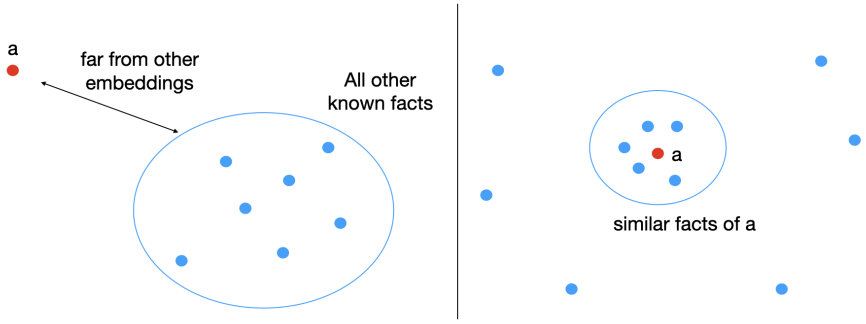

Fig. 5.5 Two typical scenarios when proving a negated atom *not a* in *NAF TSNTP*. In the first scenario, as shown on the left-hand side, *a* is far away from all other known facts in the matched topic and subdomain (according to the predicate and first argument respectively), so the proof score of *a* is close to zero and the proof score of *not a* is close to one. In the second scenario, as shown on the right-hand side, *a* is very close to other facts in the matched topic and subdomain, so the proof score of *a* is close to one and the proof score of *not a* is close to zero.

In Figure 5.5, each point represents a fact. If two points are close, it means that they are similar and have a high unification score. On the left hand side, *a* is far away from all facts, so the proof score of *a* is close to zero, using $not\ a = 1 - a$, $not\ a \approx 1$. On the right hand side, *a* is close to

some facts, so the proof score of a is close to one, then $not\ a \approx 0$. Using $not\ a = 1 - a$ as defined by Definition 5.2.1 fits the semantics of negation-as-failure naturally, because the proof score of a represents the best possible proof score when unifying a with its closest relevant facts selected based on its topic and the subdomain of its first argument. So if these facts cannot prove a, any other fact would be even less likely to prove it. It is therefore sufficient to compute the soft unification of a negated atom with respect to just the topic and first argument subdomain of the atom itself. As a result, $1 - a$ is the proof score of $not\ a$.

5.3 Implementation

In this section we describe how unification for negated literals is utilised in our *NAF TSNTP* architecture. We first discuss in Section 5.3.1 how the definite rules for revision are selected (Step 1 in Figure 5.4), then, in Section 5.3.2, how normal rules are constructed from these selected definite rules (Step 2 of Figure 5.4) and finally, in Section 5.3.3 how normal rules are evaluated to decide which ones are best and should be used to replace the original definite rule that has been revised (Step 3 of Figure 5.4).

5.3.1 Step 1: Select Definite Rules

As shown in Figure 5.4, the starting point of the normal rule induction is a set of induced definite rules. In our case, these definite rules are induced using *TSNTP* training.

TSNTP learns a set of rules, as specified by *TTRs*, but not all of them are useful. Recall that *TTRs* specify the topics of body atoms and each *TTR* explores rules with different topic combinations. If a rule is used infrequently in training, it will not be well-trained, so it is unhelpful for it to be considered for revision. Only those rules that were frequently selected for updating during training, and therefore likely to be useful rules, are considered for revision in our system. We select trained definite rules according to the frequency with which they are used in proving corrupted queries. This is because we aim to reduce the proof scores of corrupted queries, especially of false positives. Recall that the proof score of a rule is the minimum score among the proof score of each atom, so if the score of a negated atom is approximately equal to one, it has negligible effect on the overall score of the rule. On the other hand, if the score of a negated atom is near to zero, the score of the rule would be reduced. For these reasons, the proof score of a normal rule is always less than or equal to that of the definite rule from which it extends. Adding an extra atom to a body condition can never increase the proof score of a

query. As a result, normal rules could be used to reduce false positive predictions. An ideal set of extended normal rules decreases the proof scores of the corrupted queries while keeping the proof scores of positive queries unchanged.

In order to identify the frequently used definite rules induced by *TSNTP*, an evaluation *TSNTP* computational tree is constructed, denoted as *TSNTP_Definite*, using the training dataset as input. This tree is identical to the tree used in *TSNTP* training, but it does not include the fact branch at the top-level, as illustrated in Figure 5.6. Also, this tree includes forward passing only to generate a proof score for each query, unlike the *TSNTP* training tree that also computes loss value and backpropagates gradients.

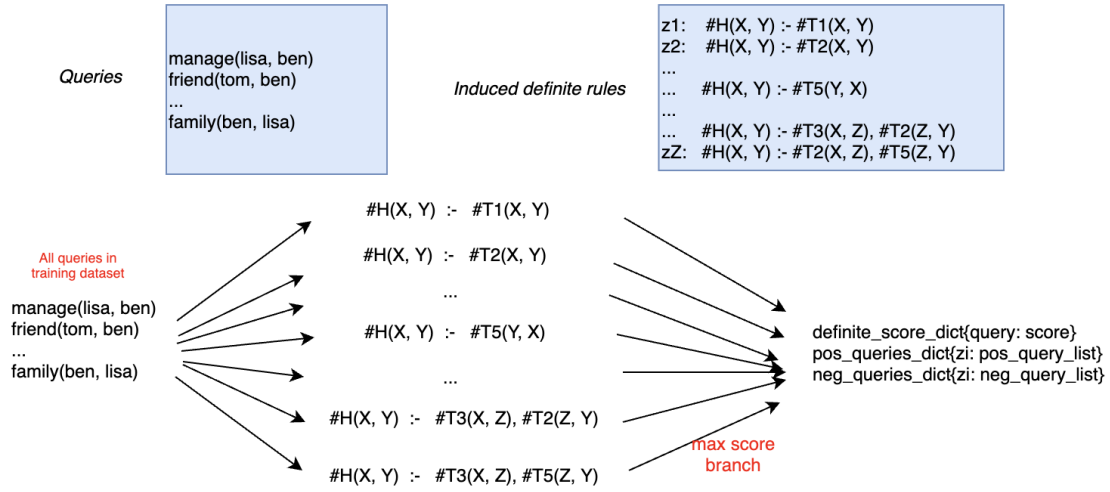

Fig. 5.6 The computational tree of *TSNTP_Definite*. The queries are all queries in the training set, including both positive queries and negative queries. The induced definite rules are the outputs of *TSNTP* induction whose embeddings are trained. The only difference between this computational tree and that of *TSNTP* induction is that this tree does not have fact branches, so all proofs need to be proved by rules. Recall that the proof score of each query is the maximum proof score among all proof branches, as defined by Definition 3.4.2. The score of each query is stored in $definite\_score\_dict$ and for each rule zi the queries for which zi contributed the score of the query is stored in a list associated with zi. There are two such dictionaries, one storing the positive queries proved by each zi, namely $pos\_queries\_dict$, and one storing the negative queries proved by each zi, namely $neg\_queries\_dict$.

Different from the *TSNTP* training tree presented in Chapter 4, the tree in Figure 5.6 does not contain any fact branch (although the same amplification hyperparameters α and β, used in scoring rules, still apply for the consistency of proof scores, so that the proof scores of *TSNTP_definite* can be compared with the proof scores generated by the trained *TSNTP*). Proofs need to be performed by one of the trained definite rules, maximising the involvement of rules. Note that, when proofs are forced to go through rule branches, some queries that were proved by facts in *TSNTP* are proved differently in *TSNTP_Definite*. This is not a problem,

because *TSNTP* only outputs the proof score of the best branch. Without facts, a query could be proved by a rule with a relatively lower score. In normal rule induction process, the absolute values of proof scores are not important, but the differences between the proof scores generated by definite rules and those generated by normal rules (later on) are important.

The evaluation results of *TSNTP_Definite* are stored as baselines and they are used to compare the evaluation results generated by normal rules later on. The evaluation results include: a dictionary $definite\_score\_dict$ that stores the proof score (value) generated by *TSNTP_Definite* for each query (key) and two dictionaries $pos\_queries\_dict$ and $neg\_queries\_dict$ respectively in which each definite induced rule zi acts as a key and stores as values all positive and corrupted (negative) queries that are proved by zi, respectively. We track the proof scores and queries dictionaries of both positive and corrupted queries, because they will be used in Step 3 to compute whether the normal rules improve the performance.

After the evaluation, a set $\mathcal{Z}$ of size Z is created that consists of the definite rules used the most by corrupted queries in *TSNTP_Definite*. Z is a hyperparameter. The rules in $\mathcal{Z}$ are extended to normal rules in Step 2.

5.3.2 Step 2: Build Normal Rule Sets

In Step 2, for each definite rule z in $\mathcal{Z}$, a set of normal rules is constructed by appending a negated atom to the body of z. To maintain the semantic information of predicates, we use all known predicates $\mathcal{P}$ in the KB to build the negated atoms and leave the normal rules selection problem to the next step, selected based on a quality function defined in Section 5.3.3. As a result, for each definite rule z, in the form of $\#H(X,Y)$:- $\#T1(X,Z), ..., P$ number of normal rules would be constructed, denoted as $\{z\_1, ..., z\_P\}$, where P is the number of known predicates $\{p_1, p_2, ..., p_P\}$ in the knowledge base, as listed below.

```
#H(X, Y) :− #T1(X, Z), ..., not p1(X, Y).
#H(X, Y) :− #T1(X, Z), ..., not p2(X, Y).
...
#H(X, Y) :− #T1(X, Z), ..., not pP(X, Y).
```

Note that in order to support the negation-as-failure computation as defined in Definition 5.2.1, we modify the order of the step where the amplification hyperparameter β is applied. Recall that in Definition 3.4.2, β is an amplification hyperparameter that amplifies the proof

scores of rules to improve the chances that rules are selected over facts. In both *TNTP* and *TSNTP*, β is applied after the proof score of a rule is computed. For example, given a rule r of the form h :- $b1, b2$ (h is the head literal and $b1$, $b2$ are body literals), the proof score of a query q using this rule is $tanh(\beta \times rule\_proof\_score(r))$, where the rule proof score of r is $min\{atom\_proof\_score(h), atom\_proof\_score(b1), atom\_proof\_score(b2)\}$. In *NAF TSNTP*, the *tanh* and β are applied to each atom directly and they are no longer applied to the $rule\_proof\_score(r)$. In that case, the proof score of q using the rule is $min\{tanh(\beta \times atom\_proof\_score(h)), tanh(\beta \times atom\_proof\_score(b1)), tanh(\beta \times atom\_proof\_score(b2))\}$, which is mathematically equivalent with the proof score generated by *TNTP* and *TSNTP* due to the monotonic characteristics of *tanh* and multiplying by β. As a result, the proof score returned by $topic\_subdomain\_uni(a)$ of an atom a is already amplified by β. The amplification is applied to each atom separately, so that when proving a negated atom *not* a using $1 - topic\_subdomain\_uni(a)$ according to Definition 5.2.1, the *topic_subdomain_uni* of a is amplified, representing the final score of a.

5.3.3 Step 3: Evaluate and Select Normal Rules

Step 3 is the most important step in normal rule induction. It evaluates each potential normal rule independently and for each definite rule z in $\mathcal{Z}$, it selects the normal rule extending z that has the lowest quality function value. We firstly introduce the quality function that measures the effects of each normal rule and then introduce how we evaluate each potential normal rule independently using the quality function and select normal rules.

Quality function. We now describe the quality function used to select which normal rules to keep from all the potential normal rules constructed in Step 2. To quantify whether a normal rule z_n is better than the definite rule z that it extended from, we need to measure its impacts from two perspectives: the average scores of positive queries proved by z should remain the same and the average scores of corrupted queries proved by z_n should decrease as much as possible compared with that computed using only z. To quantify the changes, we define a normal rule scoring function ns, as a *quality function*, which measures the performance of a normal rule z_n compared with its definite part z:

$$ns(z\_n) = \frac{\sum_{q \in z\_pos}(s_q - s'_q)}{|z\_pos|} + \frac{\sum_{q \in z\_neg}(1 - (s_q - s'_q))}{|z\_neg|} \tag{5.1}$$

where z_pos (z_neg) is the set of positive (corrupted) queries proved by definite rule z respectively (extracted from $pos\_queries\_dict$ and $neg\_queries\_dict$); s_q is the score of query q in $definite\_score\_dict$ and s'_q is the score of query q in $z\_n\_score\_dict$ proved by z_n.

To select the best normal rule, the quality function score, ns, should be minimised. The first term of Equation 5.1 measures the changes of positive queries after adding the negated term. The ideal case is when s_q and s'_q are the same for each positive query, which means that the negated atom does not decrease the proof score of positive queries. The second term of Equation 5.1 measures the changes of proof scores of corrupted queries after adding the negated atom. In this case, the bigger the difference between s_q and s'_q, the better, because it means that false positive predictions are reduced. As a result, the smaller the $\sum_{q \in z\_neg} 1-(s_q-s'_q)$, the better.

The $ns(z\_n)$ measures the improvement of z_n compared to z, in terms of overall scores across all queries proved by z. For each definite rule z in $\mathcal{Z}$, the one with the lowest ns score is selected as best extension, denoted as z_normal. According to Equation 5.1, ns score is a float in the range $[0, 2]$. If $ns(z\_n)$ is close to 2, both terms in Equation 5.1 are close to 1. It means that for each positive query, the negated atom reduces its proof score from 1 to 0 and for all corrupted queries, none of their proof scores is reduced at all. This case is the worst possible case that could result from adding a negated atom. In contrast, if $ns(z\_n)$ is close to 0, both terms in Equation 5.1 need to be close to 0. It means that for all positive queries, their proof scores remain the same as before and for each corrupted query, its proof score decreases from 1 to 0. This case is the ideal case. Note that, such ideal case could nearly never occur in our system, even if a negated atom is ideal. This is because our soft-unification rarely generates a high unification score around 1. A unification score of 0.7 is considered as a high unification score. As a result, ns ~ 0 is not possible in practice. In general, the lower the ns score, the better the quality of the induced normal rule.

In order to decide whether a selected normal rule should replace its definite rule, a hyper-parameter $NAF\_THRESHOLD$ is introduced, which is a float decided by users in the range $[0, 2]$ and reflects how strict the user accepts a normal rule extension. If $ns(z\_normal) < NAF\_THRESHOLD$, the selected normal rule replaces the original definite rule. Similar with ns score, if $NAF\_THRESHOLD$ is close to 0, it only picks the normal rules that improve the performance significantly. If $NAF\_THRESHOLD$ is close to 2, it means that the restriction is loose and all Z normal rules can replace their definite rules respectively, regardless of the

quality of the normal rules. The ideal value of $NAF\_THRESHOLD$ is found empirically via parameter tuning using the validation dataset (in our experiments, $NAF\_THRESHOLD$ is around 1.0). Other definite rules not in $\mathcal{Z}$ are kept as their original form. As a result, a new set of induced rules is generated, with some definite rules and up to Z normal rules. Note that the formula considers each rule independently and does not consider interaction between rules when deciding to retain or replace a particular induced rule.

Evaluate Normal Rules. We now describe how the normal rules constructed in Step 2 are evaluated. This evaluation builds an independent computational tree for each normal rule z_j ($1 \leq j \leq P$), which contains the normal rule branch only. Each z_j is evaluated with queries that were proved by its definite rule z. Recall that in Step 1, the *TSNTP_Definite* procedure stored two dictionaries *pos_queries_dict* and *neg_queries_dict*, to record all queries that were proved by z. Figure 5.7 illustrates a computational tree for evaluating a normal rule.

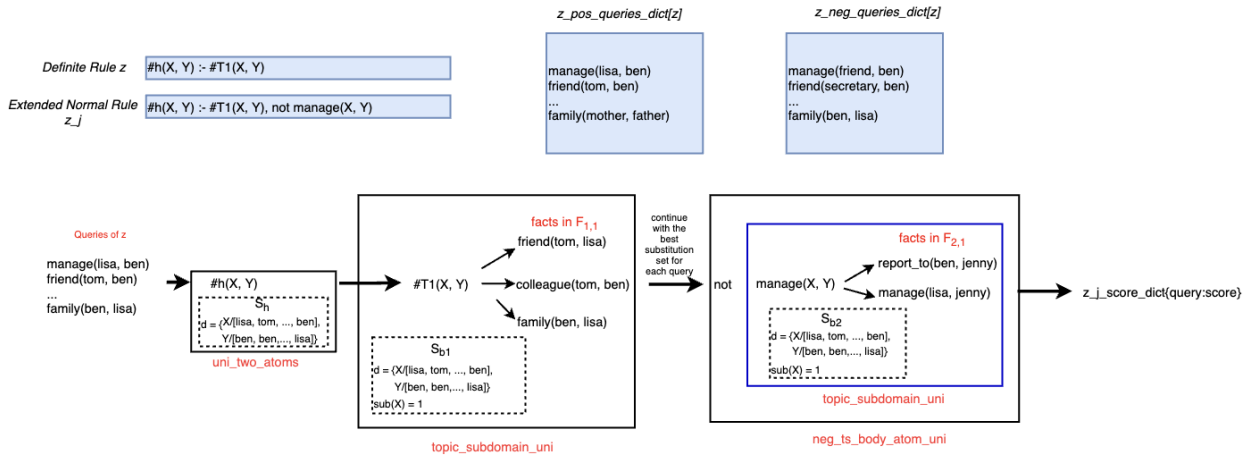

Fig. 5.7 A computational tree constructed to evaluate a normal rule $\#h(X,Y)$:- $\#T1(X,Y)$, not manage(X,Y), a potential extension of a definite rule z, denoted as z_j. The queries include all positive queries that were proved by z, extracted from *z_pos_queries_dict*, and all negative queries that were proved by z, extracted from *z_neg_queries_dict*. This tree only consists of one rule branch, the normal rule branch. The proof of the normal rule is the same as the proof of definite rules in *TSNTP*, except the negated atom. When proving the negated atom *not* $manage(X,Y)$, the computational tree firstly computes $manage(X,Y)$ using *topic_subdomain_uni* in *TSNTP*, which unifies $manage(X,Y)$ with all facts in the selected topic and subdomain and generates a proof score ps. Then, the proof score of *not* $manage(X,Y)$ is $1 - ps$. A *z_j_score_dict* is used to store the score of each query proved by this normal rule.

This tree contains only one branch, the normal rule branch. Thus, the proof scores of queries depend on this rule completely. The queries of the tree are not the full training dataset; they are the list of positive queries and corrupted queries stored in *pos_queries_dict* and *neg_queries_dict*, retrieved by the key 'z'. Thus, the queries of the normal rule z_j are the

queries in $pos\_queries\_dict[z]$ plus $neg\_queries\_dict[z]$. The evaluation results of the tree are stored as $z\_j\_score\_dict$ that records the proof score (value) of each selected query (key). Other normal rules are evaluated in the same way.

Each normal rule generated by Step 2 is evaluated as shown in Figure 5.7. As a result, $Z \times P$ computational trees are constructed, each tree containing one normal rule. These $Z * P$ computational trees can be evaluated in parallel. As a result, although there are many trees, each tree only focuses on a given subset of queries.

Figure 5.8 illustrates the P independent computational trees that are constructed in order to find the best normal rule extension of a definite rule $\#H(X,Y)$:- $\#T1(X,Y)$. Other definite rules in $\mathcal{Z}$ and their normal rule extensions are evaluated in the same way.

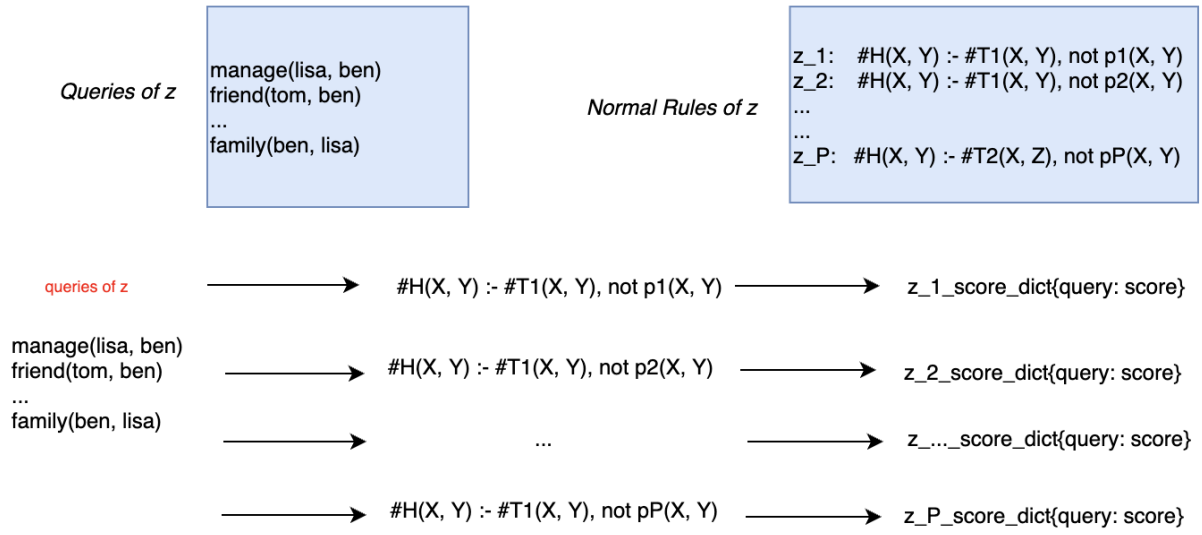

Fig. 5.8 P independent computational trees of normal rule extensions of the definite rule z. Queries that were proved by z are given to each independent computational tree and their proof scores are stored in each $z\_i\_score\_dict$ $(1 \leq i \leq P)$.

Figure 5.8 shows P independent computational trees, one for each normal rule. Each tree uses the same query set, queries of z, and their proof scores are stored in the $z\_j\_score\_dict$ $(1 \leq j \leq P)$. The proof scores of each $z\_j\_score\_dict$ are compared with the proof scores obtained from *TSNTP_Definite* to check whether z_j improves the performance of not, according to the quality function ns, as defined by Equation 5.1. Among these P normal rules, the one with the lowest ns score is selected to replace the definite rule z if it is less than or equal to $NAF\_THRESHOLD$ (see Section 5.3.1).

5.3.4 Algorithm

Algorithm 5.1 captures at a high-level the *NAF TSNTP* induction mechanism.

```
1  def NAF_TSNTP_induction(facts, train_data, trained_emb, definite_rules, pred_ids):
2      z_definite_rules, definite_score_dict, pos_queries_dict, neg_queries_dict
3        = TSNTP_Definite_evaluation(trained_emb, definite_rules, train_data)
4
5      normal_rule_dict = {}
6      for z in z_definite_rules:
7        z_pos = pos_queries_dict[z]
8        z_neg = neg_queries_dict[z]
9        min_ns = NAF_THRESHOLD
10       z_normals = create_normal_rules(z, pred_ids)
11       for z_j in z_normals:
12           z_j_score_dict = evaluate_z_j(z_j, z_pos, z_neg)
13           z_j_ns = compute_ns_score(z_j_score_dict, z_pos, z_neg)
14           if z_j_ns <= min_ns:
15               min_ns = z_j_ns
16               normal_rule_dict[z] = z_j
17
18     new_normal_rules = merge_normal_rule(definite_rules, normal_rule_dict)
19     return new_normal_rules
```

Algorithm 5.1 *NAF TSNTP* induction mechanism. Line 2-3 is the first step that selects definite rules. Then, in the outer for-loop, a set of tracking dictionaries are created and a set of normal rules are created as described in step 2. In the inner for-loop, each normal rule is evaluated independently and its *ns* score is computed and evaluated. In the end, the selected normal rules combine with definite rules, forming the output rule set.

Algorithm 5.1 captures the mechanism of normal rule induction in *NAF TSNTP*. It receives the trained results of *TSNTP* as input, including trained embeddings and induced definite rules, which is also known as the Phase 1 of *NAF TSNTP*.

The algorithm starts with *TSNTP_Definite* evaluation which is Step 1 of normal rule induction phase in Figure 5.4. It forces the queries to be proved by one of these definite rules (line 2-3). Among them, the Z most frequently used definite rules in corrupted (negative) query proofs are selected to build normal rules. In line 5, an empty *normal_rule_dict* is constructed, which is used to store which selected normal rules would replace their definite rules (in line 6-16).

Inside the big for-loop, Lines 7-10 create all possible extensions for each z in $z\_definite\_rules$ and set up the environment for further selection. Lines 7-8 retrieve positive and corrupted queries that were proved by each definite rule z and they are used to evaluate normal rules extensions of z. In line 9, the initial value of min_ns is set to $NAF\_THRESHOLD$ for each z, which tracks the minimum ns value among normal rule extensions of z. The min_ns is set to $NAF\_THRESHOLD$, because only a normal rule with $ns \leq NAF\_THRESHOLD$ would be selected to replace its definite parts. In line 10, each z are extended with all possible negated atoms, to form a set of normal rules, which is equivalent to Step 2 in Figure 5.4.

The inner for-loop (lines 11-16) shows the process of finding the best normal rule with the minimal ns score. This is equivalent to Step 3 in Figure 5.4. In this for-loop, each normal rule z_j is evaluated independently by a computational tree with one branch only and the proof scores generated by the tree (line 12) are used to compute the ns score (line 13). If the ns score of z_j is less than or equal to min_ns, it replaces min_ns and updates the $normal\_rule\_dict$ of the key z with value z_j. (lines 14-16) Recall that the default value of min_ns is $NAF\_THRESHOLD$, so if the ns scores of all normal rule extensions of z are above the threshold, no normal rule would replace z. In that case, there would be no entries in the $normal\_rule\_dict$ for key z, and z would remain in its original definite form in the induced rule set.

After the loop, the $normal\_rule\_dict$ would contain all selected normal rules that will replace their definite counterparts. Line 18 uses each normal rule in $normal\_rule\_dict$ to replace its definite rule and merges with other definite rules, which generates the output 'new_normal_rules' as shown in 5.4. In the end, a set of rules is induced, which includes definite rules and normal rules.

5.4 Decoding Normal Rules

The decoding of a normal rule is similar to the decoding method for a definite rule, as described in Section 3.5. A negated atom 'not *p(X, Y)*' is decoded to its topmost closest neighbours in the vector space (the actual number is specified by the user), where the decoding score of the negated atom is the symbol unification score between p and its neighbours. Note that, since the negated predicate p is selected from known predicates, its closest match is the known predicate, with the score 1.0 (a perfect match). Its other decodings are the neighbours of p.

Here we present an example normal rule induced in *UMLS* dataset (assume three closest neighbours of each literal in the induced rule were considered).

#T1(X, Y) :– #T0(X, Y), not isa(X, Y).

0.89 uses(X, Y) :– produces(X, Y), not isa(X, Y).

0.89 uses(X, Y) :– produces(X, Y), not ingredient_of(X, Y).

0.89 produces(X, Y) :– uses(X, Y), not isa(X, Y).

0.89 produces(X, Y) :– uses(X, Y), not ingredient_of(X, Y).

Recall that the decoding score of a rule is calculated as the minimum score of its predicate decoding scores. The original definite rule is induced by *TSNTP*, which captures the relationship between 'produces' and 'uses' (they correlate highly in the dataset). According to the normal rule induction method, a negated atom 'isa(X, Y)' is selected to extend the definite rule. The negated atom helps to reduce false positive predictions.

Summary. *NAF TSNTP* enables normal rule induction that supports negation-as-failure and soft-unification. The system employs two phases, one phase for a standard *TSNTP* training and a second phase for normal rule induction, which constructs and selects normal rules derived from some of the induced definite rules from *TSNTP*. The system returns a set of definite and normal rules according to the quality function given by Equation 5.1, which reduces the number of false positive predictions while keeping the true positive predictions unaffected. The evaluation of this approach is presented in Chapter 7.

Chapter 6

Related Works

In this chapter, we present some related neural-symbolic systems that perform query answering, where given a knowledge base or knowledge graph, they predict the score for a given ground query or predict the missing entity of a partial query. Various systems have been designed to tackle the query answering task ([Trouillon et al., 2016], [Dettmers et al., 2018], [Guo et al., 2016], [Minervini et al., 2017], [Yang et al., 2015], [Rocktäschel and Riedel, 2017], [Minervini et al., 2018], [Minervini et al., 2020a], [Minervini et al., 2020b], [Das et al., 2018], [Yang et al., 2017][1]) and we have selected typical systems that feature different approaches, such as deep learning, reinforcement learning and adversarial learning. In all systems, embeddings are used to represent symbols in the knowledge base, which encode the knowledge base in a neural form and could take the advantages of both neural networks and symbolic reasoning.

Category	Group	System
Query answering	Query answering without rules	ComplEx
		ConvE
	Query answering with given rules	KALE
		ASR
	Query answering with rule mining	DistMult
Rule induction	NTPs	NTP 2.0
		GNTP
		CTP
	Other systems	NeuralLP
		MINERVA
		RNNLogic

Table 6.1 Related systems considered in this chapter are divided into two main categories: query answering and rule induction. The query answering category is further divided according to if rules are supported and whether the rules are learned or not. The rule induction category is further divided into two groups: systems extending *NTP* and other systems.

[1]Notations used in this chapter follow the original notations used in the original papers. Notations used in each work are independent from other notations and may carry different meanings.

We divide the selected related works to two main categories, according to whether rule induction is involved, as shown in Table 6.1. The first category is earlier systems of neural-symbolic integration which involve relatively basic logic reasoning and perform query answering tasks without inducing rules (see Section 6.1). The second category is query answering systems that induce rules (see Section 6.2). We further divide systems in the first category to three groups: query answering without rules ([Trouillon et al., 2016], [Dettmers et al., 2018]), query answering with given rules ([Guo et al., 2016], [Minervini et al., 2017]) and query answering with rule mining ([Yang et al., 2015]). We further divide systems in the second category to two groups: *NTPs* ([Minervini et al., 2018], [Minervini et al., 2020a] and [Minervini et al., 2020b]) and other works ([Yang et al., 2017], [Das et al., 2018], [Qu et al., 2020]).[2] Given a knowledge base/knowledge graph, each rule induction system aims to induce a set of rules that derives as many positive examples and as few negative examples as possible. This rule induction ability is crucial, because induced rules uncover implicit relationships in the knowledge base, which could be used to deduce more facts. The interpretability brought by induced rules also make the query answering process more explainable, unlike those neural network systems which act as a black-box without giving the reasons behind each prediction [Cyras et al., 2021]. Note that approaches for rule induction can also perform query answering, but they are analysed in this chapter from the rule induction perspective.

There are two kinds of query answering tasks: query prediction and query completion (also known as link prediction). We define the two query answering tasks formally as follows, where the score indicates how likely it is that a query $q(s, o)$ is true given the *KB*.

Definition 6.0.1. Given a knowledge base *KB*, the query answering task for query prediction is to learn a scoring function that generates a prediction score *ps* for each ground query $q(s, o)$.

Definition 6.0.2. Given a knowledge base *KB*, the query answering task for query completion is to learn to identify the missing subject s or object o of each fact $q(s, o)$ in the *KB*, for each partial query in the form of $q(s, ?)$ or $q(?, o)$. In some cases, only the constant with the highest prediction score is returned, and in other cases, multiple constants above a given threshold are returned.

All systems, except *MINERVA* [Das et al., 2018] and *RNNLogic* [Qu et al., 2020], belong to 'query answering task for query prediction'. Furthermore, all systems, except *ComplEx* [Trouillon

[2]Apart from these systems, *AMIE* [Galárraga et al., 2015] and *FastLAS* [Law et al., 2020] can also induce rules, but they are symbolic systems without using embedding representations, so they are not included in the discussion.

et al., 2016] and *ConvE* [Dettmers et al., 2018], use rules to solve query answering tasks, either by using given background rules or by inducing new rules, such that all the rules are function-free definite Horn rules of various allowed forms. We define function-free definite Horn rules and linked definite Horn rules in Definition 6.0.3.

Definition 6.0.3. A function-free definite Horn rule is a rule of the form: h :- $b1, ..., bn$ $(n \geq 1)$, where h and bi are positive binary atoms with variable arguments only and the first body atom shares a variable with the head. Additionally, a linked function-free definite Horn rule also satisfies the following properties.

- There is no variable in the head that does not occur in at least one body atom.
- Each body atom that is not the first or the last shares one argument with its predecessor and the other argument with its successor.
- The last body atom shares one argument with its predecessor and one argument with the head.[3]

Some examples of linked definite Horn rules are shown below.

```
p(X, Y) :− q(Y, X).
p(X, Y) :− q(X, Z), r(Z, W), s(W, Y).
p(X, Y) :− q(Z, X), r(Z, Y).
p(X, Y) :− q(Z, X), r(Y, Z).
```

We summarise in Table 6.2 the various forms of rule syntax supported in these systems. Dashes mean that rules are not supported.

	Rule Syntax
ComplEx	–
ConvE	–
KALE	p(X, Y) :- q(X, Y) p(X, Y) :- q(X, Z), r(Z, Y)
ASR	definite Horn rules
DistMult	linked definite Horn rules
NTP 2.0	definite Horn rules
GNTP	definite Horn rules
CTP	definite Horn rules
NeuralLP	linked definite Horn rules
MINERVA	linked definite propositional/first-order rules
RNNLogic	linked definite Horn rules

Table 6.2 The syntax of rules supported in different systems.

[3]Note that if there is only one body atom, both arguments are shared with the head.

Our systems use the same set of basic template rule structures as *NTP 2.0*, *GNTP* and *CTP*, but modified to topical template rules. Also, instead of definite Horn rules, our *NAF TSNTP* can also induce normal rules.

6.1 Query Answering

We now present systems that focus on query answering for knowledge bases.

6.1.1 Query Answering without Rules

Two examples of query answering systems that do not involve rules are *ConvE* [Dettmers et al., 2018] and *ComplEx* [Trouillon et al., 2016]. These systems are not provided with a set of rules for deduction nor are they able to learn rules during training. Their training goals are to get good embedding representations of the knowledge base and to learn a prediction function that generates a score for each query. The common disadvantages of *ConvE* and *ComplEx* are the lack of interpretability of prediction results and the lack of generality of the learned model. Because no rules are learnt in these systems, there is no extracted human readable justification for predictions, so the decision making process of such systems remain as a blackbox. Also, although it is possible to achieve good query answering results without learning rules, it would be harder for the system to capture more complex relationships, such as transitive relations.

ComplEx

ComplEx uses complex-valued embeddings to capture symmetric and antisymmetric relations through latent factorisation. Using complex values as embeddings is unusual, because embeddings typically consists of real numbers only. In *ComplEx*, both predicates and constants are K-dimensional complex vectors. For each complex vector $\mathbf{u} \in \mathbb{C}^K$, $\mathbf{u} = Re(\mathbf{u}) + iIm(\mathbf{u})$, where $Re(\mathbf{u}) \in \mathbb{R}^K$ is the real part of the complex vector $\mathbf{u}$; $Im(\mathbf{u}) \in \mathbb{R}^K$ is the imaginary part of $\mathbf{u}$; and i is the square root of -1. In implementation, $Re(\mathbf{u})$ and $Im(\mathbf{u})$ are represented by two K-dimension real vectors. The conjugate of $\mathbf{u}$, $\bar{\mathbf{u}} = Re(\mathbf{u}) - iIm(\mathbf{u})$, is denoted $\bar{\mathbf{u}}$.

The scoring function ϕ of the model for each query $r(s, o)$ is defined as:

$$\phi(r, s, o; \Theta) = Re(< \mathbf{w}_r, \mathbf{e}_s, \bar{\mathbf{e}}_o >) \tag{6.1}$$

where $\mathbf{w}_r$ represents the complex embedding of the predicate r; $\mathbf{e}_s$ represents the complex embedding of the subject s; $\bar{\mathbf{e}}_o$ represents the complex conjugate of the object o; and Θ represents the learning parameters of the model (complex embeddings of predicates and constants). The $< \mathbf{a}, \mathbf{b}, \mathbf{c} >$ denotes the standard componentwise multi-linear dot product, such that $< \mathbf{a}, \mathbf{b}, \mathbf{c} >$: $-\Sigma_k a_k b_k c_k$. Thus, the scoring function can be further expanded as:

$$\begin{aligned}\phi(r, s, o; \Theta) =& Re(< \mathbf{w}_r, \mathbf{e}_s, \bar{\mathbf{e}}_o >) \\ =& Re(\Sigma_{k=1}^{K} w_{rk} e_{sk} \bar{e}_{ok}) \\ =& < Re(\mathbf{w}_r), Re(\mathbf{e}_s), Re(\mathbf{e}_o) > + < Re(\mathbf{w}_r), Im(\mathbf{e}_s), Im(\mathbf{e}_o) > \\ &+ < Im(\mathbf{w}_r), Re(\mathbf{e}_s), Im(\mathbf{e}_o) > - < Im(\mathbf{w}_r), Im(\mathbf{e}_s), Re(\mathbf{e}_o) >\end{aligned} \tag{6.2}$$

Complex vectors play an important role in learning antisymmetric relationship, especially the conjugate vector of the object, $\bar{\mathbf{e}}_o$. Because the conjugate vector is used, the scoring function ϕ generates different scores for a query $r(s, o)$ and $r(o, s)$, whereas without using conjugate vectors, $r(s, o)$ and $r(o, s)$ would get the same score. If $\mathbf{w}_r$ is purely imaginary, it indicates that the function is antisymmetric. If $\mathbf{w}_r$ is real, it indicates a symmetric relationship.

Its loss function is the negative log-likelihood of the logistic model with L^2 regularisation on parameters Θ:

$$\min_{\Theta} \sum_{r(s,o) \in \Omega} log(1 + exp(-Y_{rso} \phi(s, r, o; \Theta))) + \lambda \|\Theta\|_2^2 \tag{6.3}$$

where Ω is the training dataset; Y_{rso} is the target label of query $r(s, o)$; and Θ represents complex embeddings of predicates and constants in the knowledge base.

ConvE

ConvE [Dettmers et al., 2018] is a multi-layer convolutional neural network (*CNN*) model for query answering. Predicates and constants in the knowledge base are encoded as embeddings, which are learnt during training. Its high-level architecture is illustrated in Figure 6.1:

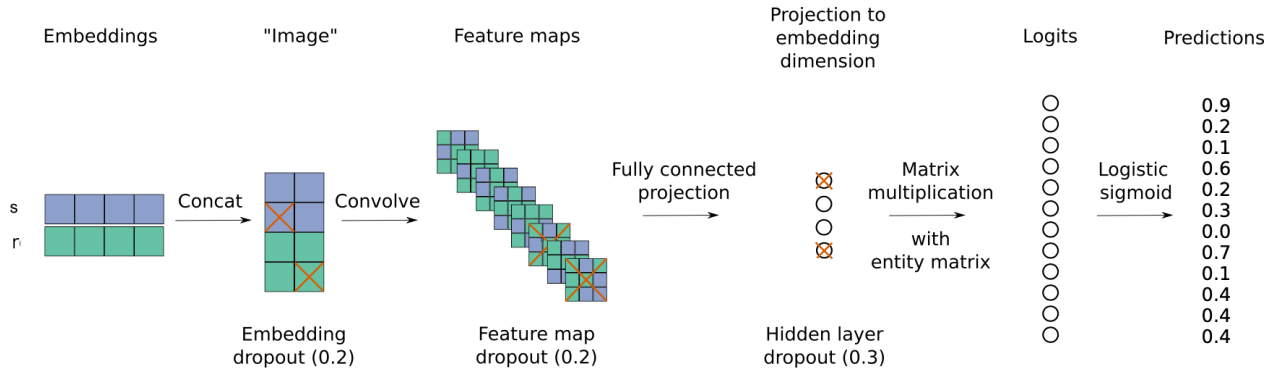

Fig. 6.1 [Dettmers et al., 2018] illustrates how *ConvE* generates prediction scores for all queries in the form of $r(s, E)$ which has a predicate r, a subject s and an object E which can be any constant in the knowledge base. The embeddings are firstly reshaped as 'images' and then converted to multiple feature maps as a part of *CNN* training. Then, the feature maps are projected to embedding dimension and multiply with entity matrix E, forming a prediction score for each entity.

In Figure 6.1, the first step gives the embeddings of r and s as input. The second step reshapes the embeddings r and s, denoted as $\mathbf{r}_r$ and $\mathbf{e}_s$, which after the reshape are denoted as $\bar{\mathbf{r}}_r$ and $\bar{\mathbf{e}}_s$. They are then concatenated. Dropout can be applied to the concatenated embedding during training to improve robustness of the network. At the third step, this concatenated embedding becomes the input of a 2D convolutional layer f. This convolutional layer returns a feature map tensor $\boldsymbol{\tau} \in \mathbb{R}^{c*m*n}$ which contains c number of 2D feature maps, each with dimensions m and n. The convolutional layer also has a filter $\mathbf{w}$ that controls the dropout of the c feature maps. At step four, $\boldsymbol{\tau}$ is reshaped to a vector $vec(\boldsymbol{\tau}) \in \mathbb{R}^{cmn}$ and it is further projected to a k dimensional embedding by the linear transformation matrix $\mathbf{W} \in \mathbb{R}^{cmn*k}$. At the fifth step, the k-dimension projected embedding dot-products with all possible object embeddings (say there are g constants in the knowledge base), to generate a list of g predictions. In the end, a logistic sigmoid function is applied, forming g final prediction scores over all possible queries in the form of $r(s, E)$.

Following the notation used in Figure 6.1, the scoring function $\psi_r(\mathbf{e}_s, \mathbf{e}_o)$ of a query $r(s, E)$, where $E = o$ is defined as below:

$$\psi_r(\mathbf{e}_s, \mathbf{e}_o) = f(vec(f([\bar{\mathbf{e}}_s; \bar{\mathbf{r}}_r] * \mathbf{w}))\mathbf{W})\mathbf{e}_o \tag{6.4}$$

The logistic sigmoid function $\sigma(\cdot)$ is applied to the scoring function, to give a prediction score $q_i = \sigma(\psi_r(\mathbf{e}_s, \mathbf{e}_o))$ for the ith query in the training dataset q, $r(s, o)$, which is used to define the binary cross-entropy loss, as below:

$$\mathcal{L}(q, t) = -\frac{1}{N}\sum_i t_i \cdot log(q_i) + (1 - t_i) \cdot log(1 - q_i) \tag{6.5}$$

where t is set of target labels of queries in q, and q_i and t_i are the prediction score of the ith query and its corresponding target label.

Comparison between ComplEx and ConvE *ComplEx* learns the embedding representations of predicates and constants in a knowledge base capturing semantic meaning and it is able to use semantic similarity to prove similar queries. However, with no involvement of rules, it can only capture symmetric and antisymmetric relationships. As a result, more complicated relationships, such as a transitive relationship, are beyond its scope. On the other hand, its simple architecture makes it more computationally efficient than most rule induction systems if learning embedding representations is the main learning goal. *ConvE* makes an interesting attempt to solve a query answering task with a symbolic knowledge base using a convolutional neural network. It treats the embeddings of predicates and constants as 'images' and converts these 'embedding images' to different feature maps. Although these feature maps might help it to identify non-linear relationships for answering queries, this system has limited connections with logic reasoning. Although neither works support rules, *ComplEx* could capture symmetric and antisymmetric relationships and its embeddings captures semantic information. However, we did not see the claim of learning any kinds of relations or semantic meaning in *ConvE*. Its convolutional architecture with multiple feature maps makes it hard to interpret its decision making process.

6.1.2 Query Answering with Given Rules

The two systems *KALE* [Guo et al., 2016] and *ASR* [Minervini et al., 2017] are able to deal with a set of given background rules. *KALE* uses grounded rules as part of training dataset and *ASR* uses rules to generate adversarial examples to improve overall robustness of the learned embeddings. Both systems are more advanced than *ComplEx* and *ConvE*, because background rules enable them to capture more complex relationships, such as the multi-hop and transitive relations, given these relations that are captured by the background rules. Both systems use background rules to train embeddings, but do not use then to deduce new facts, as is typical in logic programming, for instance.

The main limitation of these two systems is their reliance on given rules. If the quality of known rules is bad or there are only a few background rules, the performance of these systems would be affected. Both systems extend from existing query answering systems that uses embeddings, i.e. *TransE* [Bordes et al., 2013], *ComplEx* [Trouillon et al., 2016] and *DistMult*

[Yang et al., 2015]. If there are no given rules, *KALE* and *ASR* are equivalent to the systems that they extended from.

KALE

KALE [Guo et al., 2016] is a neural-symbolic system that aims to represent and model facts and logical rules in a unified framework. Given a knowledge base, facts are represented as triples of the form (subject, predicate, object) following *TransE* assumption (which will be explained later) and represents Horn rules as complex formulae modelled by the t-norm fuzzy logic ([Rocktäschel et al., 2015] and [Hájek, 1998]). At the time of this work, it was novel to learn embeddings that are compatible with both facts and rules, although this has become a new norm now. The system uses a loss function to be minimised by stochastic gradient descent and the key element of the loss function is the scoring function. The only learning parameter is the embedding matrix of all predicates and constants.

KALE follows *TransE* [Bordes et al., 2013] and it models predicate embedding as a translation between two entity embeddings. i.e. for a fact $p(a, b)$, they assume $\mathbf{e}_a + \mathbf{r}_p \approx \mathbf{e}_b$ where $\mathbf{r}$ is the embedding matrix of predicates and $\mathbf{e}$ is embedding matrix of constants ($\mathbf{r}_p$ refers to the embedding of p and $\mathbf{e}_a$ refers to the embedding of a). They get inspiration from [**?**], which identifies linguistic regularities such as France - Paris = Germany - Berlin. As a result, the scoring function for each grounded triple $k(i, j)$ is defined as:

$$I(\mathbf{e}_i, \mathbf{r}_k, \mathbf{e}_j) = 1 - \frac{1}{3\sqrt{d}} \|\mathbf{e}_i + \mathbf{r}_k - \mathbf{e}_j\|_1 \tag{6.6}$$

where d is the dimension of the embedding. A high value of $I(\mathbf{e}_i, \mathbf{r}_k, \mathbf{e}_j)$ indicates that the triple is true.

KALE is able to use given rules during training. *KALE* claims that it can handle any first-order logic formulae, but in [Guo et al., 2016] it restricts the rules to two types. The first type has the form:

$$\forall x, y : (x, \mathbf{r}_s, y) \Rightarrow (x, \mathbf{r}_t, y)$$

Given a ground rule $f \triangleq (\mathbf{e}_m, \mathbf{r}_s, \mathbf{e}_n) \Rightarrow (\mathbf{e}_m, \mathbf{r}_t, \mathbf{e}_n)$, following the t-norm fuzzy logic ([Rocktäschel et al., 2015] and [Hájek, 1998]), the truth value is computed as:

$$I(f) = I(\mathbf{e}_m, \mathbf{r}_s, \mathbf{e}_n) \cdot I(\mathbf{e}_m, \mathbf{r}_t, \mathbf{e}_n) - I(\mathbf{e}_m, \mathbf{r}_s, \mathbf{e}_n) + 1 \tag{6.7}$$

The second type has the form:

$$\forall x, y, z : (x, \mathbf{r}_{s1}, y) \wedge (y, \mathbf{r}_{s2}, z) \Rightarrow (x, \mathbf{r}_t, z)$$

Given a ground rule $f \triangleq (\mathbf{e}_l, \mathbf{r}_{s_1}, \mathbf{e}_m) \wedge (\mathbf{e}_m, \mathbf{r}_{s2}, \mathbf{e}_n) \Rightarrow (\mathbf{e}_l, \mathbf{r}_t, \mathbf{e}_n)$, following the t-norm fuzzy logic, the truth value is computed as:

$$I(f) = I(\mathbf{e}_l, \mathbf{r}_{s1}, \mathbf{e}_m) \cdot I(\mathbf{e}_m, \mathbf{r}_{s2}, \mathbf{e}_n) \cdot I(\mathbf{e}_l, \mathbf{r}_t, \mathbf{e}_n) - I(\mathbf{e}_l, \mathbf{r}_{s1}, \mathbf{e}_m) \cdot I(\mathbf{e}_m, \mathbf{r}_{s2}, \mathbf{e}_n) + 1 \tag{6.8}$$

Note that all rules need to be grounded before training and only ground rules including at least one fact in the knowledge base would be used. For example, a ground rule $(\mathbf{e}_l, \mathbf{r}_{s_1}, \mathbf{e}_m) \wedge (\mathbf{e}_m, \mathbf{r}_{s2}, \mathbf{e}_n) \Rightarrow (\mathbf{e}_l, \mathbf{r}_t, \mathbf{e}_n)$ would only be selected if at least one atom in $\{(\mathbf{e}_l, \mathbf{r}_{s_1}, \mathbf{e}_m), (\mathbf{e}_m, \mathbf{r}_{s2}, \mathbf{e}_n), (\mathbf{e}_l, \mathbf{r}_t, \mathbf{e}_n)\}$ is in the knowledge base. As a result, although it uses this heuristic, the size of the ground rule set is still large.

The loss function is defined as:

$$\begin{aligned} &\min_{\{e\},\{r\}} \sum_{f^+ \in \mathcal{F}} \sum_{f^- \in \mathcal{N}_{f^+}} [\gamma - I(f^+) + I(f^-)]_+, \\ &\text{s.t.} \|\mathbf{e}\|_2 \leq 1, \forall \mathbf{e} \in \boldsymbol{\epsilon}; \|r\|_2 \leq 1, \forall \mathbf{r} \in \mathcal{R}. \end{aligned} \tag{6.9}$$

where γ is a margin that separates positive and negative formulae; $[x]_+ \triangleq max\{0, x\}$, $\|\mathbf{x}\|_2 = \sqrt{\sum_i x_i^2}$; $\boldsymbol{\epsilon}$ and $\mathcal{R}$ are the sets of constants and predicates in the given signature respectively. The $\mathcal{F}$ represents the positive training dataset set. However, different from other systems in this chapter, where training dataset usually contains facts only, the training dataset $\mathcal{F}$ in *KALE* contains both facts and ground rules, where each rule contains at least one fact in $\mathcal{F}$. $\mathcal{N}_{f^+}$ is the negative training dataset, which is created by corrupting facts and ground rules. One item of negative training data is created for each $f^+ \in \mathcal{F}$. For a fact $p(a, b)$, its corrupted atom is created by randomly changing either a or b, such that the corrupted atom does not belong to known facts. Rule corruption is created by randomly replacing the predicate of a head atom. For instance, given a ground rule $located\_in(paris, france)$:- $capital\_of(paris, france)$, one of its potential corruptions could be: $far\_from(paris, france)$:- $capital\_of(paris, france)$.

Compared to *ComplEx* and *ConvE*, that do not support logic reasoning, *KALE* is able to support background rules and use these background rules to answer queries. However, this also means that the quality of query answering prediction in *KALE* depends on the quality of background rules. Although it still cannot induce rules, these background rules could be used

to express more complicated relationships, such as transitive and multi-hop relations. In [Guo et al., 2016] the authors claim that 'the *KALE* framework is general enough to handle any rules that can be represented as first-order logic formulae', but only two types of rule are used in the paper as specified by Equation 6.7 and 6.8. *KALE* only uses these rules for answering queries by computing the I value as defined in Equation 6.7 and Equation 6.8. Also, *KALE* did not use background rules in the same way as logic programming for deducing more facts. Its training goal is not to deduce more facts, but to make sure the I value of a positive fact or a positive ground rule is higher than the I value of a negative fact or a negative ground rule. Furthermore, *KALE* only supports grounded rules, so any first-order rules need to be grounded first, which could generate a long list of rules.

ASR

ASR [Minervini et al., 2017] is an adversarial system that trains jointly with other neural-symbolic learning systems and uses adversarial examples of given Horn rules to regularise the training process. It has to be used jointly with another neural query answering system which performs link prediction tasks. In [Minervini et al., 2017], the performance of *ASR* is evaluated when it is used with *DistMult* [Yang et al., 2015] and *ComplEx* [Trouillon et al., 2016].

The training objective of an adversarial system is a minimax problem, which is a zero-sum game. It consists of two parts: an adversary that identifies the most violating examples and a discriminator that tries to minimise the loss. During *ASR* training, given a knowledge base with a set of first-order Horn rules as assumptions, an adversary identifies the most offending adversarial constant groundings for variables in the given rules which outputs the biggest inconsistent loss. These adversarial examples are then used as inputs to the discriminator. The high-level architecture of the *ASR* adversary part is illustrated in Figure 6.2.

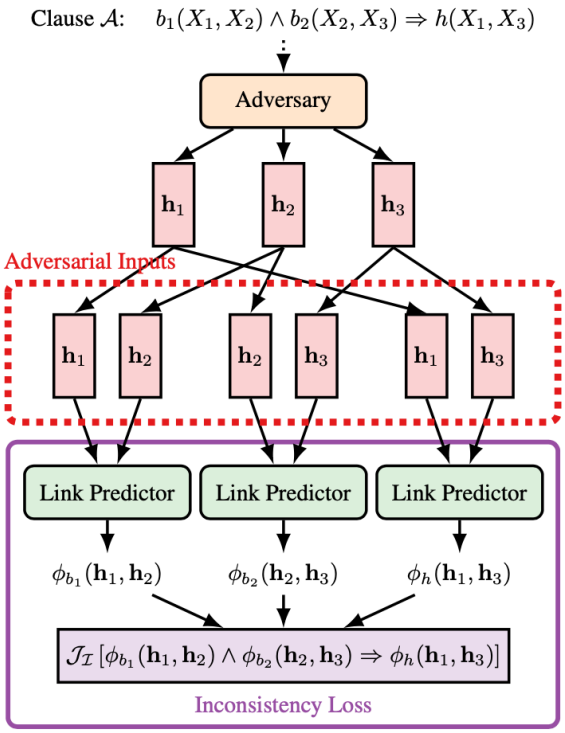

Fig. 6.2 The high-level architecture of the *ASR* adversary component [Minervini et al., 2017]. Given a rule, the adversary identifies the most violating constant groundings for variables. These constant groundings are fed into link predictors to compute the inconsistency loss. The training goal is to minimise the inconsistency loss.

Given a rule A with variables $\{X_1, X_2, X_3\}$, the adversary part finds the set of constant embeddings $\{\mathbf{h_1}, \mathbf{h_2}, \mathbf{h_3}\}$ for the variables that generate the biggest inconsistent loss score. Using these adversary constant embeddings, the system generates a prediction score for each literal of A using a link predictor (either *ComplEx* or *DistMult*), where the prediction score of each literal in the form of $p(s, o)$ is represented as $\phi_p(s, o)$. The prediction scores of body atoms can be combined using Gödel t-norm, a continuous generalisation of the conjunction operator in logic [Gupta and Qi, 1991]. The combined score of body literals in A, denoted as $\phi(BODY)$, is then used to compute the inconsistency loss, where the inconsistency loss for each Horn rule assumption (of the form BODY $\Rightarrow$ HEAD) is defined as:

$$\mathcal{J}_{\mathcal{I}}(\text{BODY} \Rightarrow \text{HEAD}) \triangleq [\phi(\text{BODY}) - \phi(\text{HEAD})]_+ \tag{6.10}$$

$\phi(X)$ is the neural link prediction score generated for X. $[]_+$ is the *ReLu* function that only returns the positive result of the computations inside the [] and returns 0 if the computation result is negative. A high loss indicates that the rules are false (i.e. the body is true and the head is false). The inconsistency loss helps maintain consistency with the given assumptions in $\mathcal{A}$.

After the adversary component, the discriminator component (not shown in Figure 6.2) is trained by minimising the joint loss function $\mathcal{J}(\mathcal{G}, \mathcal{A}; \boldsymbol{\theta}, \boldsymbol{\gamma}, \mathcal{S})$ between fact loss $\mathcal{J}_{\mathcal{F}}$ and the inconsistency loss $\mathcal{J}_{\mathcal{I}}$. This is defined in Equation 6.11:

$$\mathcal{J}(\mathcal{G}, \mathcal{A}; \boldsymbol{\theta}, \boldsymbol{\gamma}, \mathcal{S}) \triangleq \mathcal{J}_{\mathcal{F}}(\mathcal{G}; \boldsymbol{\theta}, \boldsymbol{\gamma}) + \alpha \mathcal{J}_{\mathcal{I}}(\mathcal{A}; \boldsymbol{\theta}, \mathcal{S}) \tag{6.11}$$

where $\mathcal{G}$ represents the knowledge graph; $\mathcal{A}$ represents assumptions (function-free Horn rules); $\mathcal{S}$ represents the adversarial input set; $\boldsymbol{\theta}$ represents embeddings of constants and $\boldsymbol{\gamma}$ represents embeddings of predicates; $\alpha \in \mathbb{R}$ controls the weight given to the assumption rules in the optimisation process. $\mathcal{J}_{\mathcal{F}}(\mathcal{G}; \boldsymbol{\theta}, \boldsymbol{\gamma})$ is the loss function over training dataset, which can be replaced by any neural link predictors. For instance, it can be replaced by the loss function in *ComplEx*. $\mathcal{J}_{\mathcal{I}}(\mathcal{A}; \boldsymbol{\theta}, \mathcal{S})$ is specific to the *ASR* architecture.

The training goal of *ASR* can be viewed as a minimax problem, where the adversary aims to identify input set $\mathcal{S}$ with maximal inconsistency and the discriminator minimises the inconsistency through parameter $\boldsymbol{\theta}$ and $\boldsymbol{\gamma}$. The minimax problem can be defined formally as:

$$\min_{\theta,\gamma} \max_{\mathcal{S}} \mathcal{J}(\mathcal{G}, \mathcal{A}; \boldsymbol{\theta}, \boldsymbol{\gamma}, \mathcal{S}) \tag{6.12}$$

This means that it firstly identifies the $\mathcal{S}$ with biggest inconsistency and then optimises parameters $\boldsymbol{\theta}, \boldsymbol{\gamma}$ to minimise this inconsistency and query answering loss.

The strength of *ASR* is its ability to use background rules (in the form of Horn rules) to generate adversarial examples, which improves the robustness of the system through adversarial training. However, *ASR* is a regulariser that has to work with other systems in order to train embeddings. This is unlike our systems which work as independent systems that train embeddings and induce rules. Its performance also relies on the quality of given background rules. If given background rules are wrong, wrong adversary examples would be found, making the training more noisy.

6.1.3 Query Answering with Rule Mining

DistMult is a query answering system that uses rule mining. However, the rule extraction in *DistMult* is performed as a post-processing step, making it different from other query answering methods in Section 6.2, where the rule extraction happens during the training process. Its rule mining algorithm enables it to learn linked Horn rules from a given knowledge base. Since

DistMult can learn rules, it could be used to capture unknown relationships in a knowledge base, making it more advanced than systems that use just the given background rules (such as *ASR* and *KALE*).

DistMult consists of two parts: the embedding learning part, which involves training, and the rule extraction part which uses the trained embeddings. In the embedding learning part, no rules are involved and the training target is to find embedding representations for predicates and constants in the knowledge base that help to solve the query answering task. Instead of initialising the embeddings randomly and training them by gradient descent, *DistMult* uses the one-hot encoding to represent constants. These constant vectors are fixed values, where the one-hot index vector of each constant e_i is denoted as $\mathbf{x}_{e_i}$. The fixed one-hot vector is then projected to a n-dimension vector: the learned constant representation $\mathbf{y}_{e_i}$, which could carry semantic information through training. This transformation is defined as:

$$\mathbf{y}_{e_i} = f(\mathbf{W}\mathbf{x}_{e_i}) \tag{6.13}$$

where f can be a linear or non-linear function and $\mathbf{W}$ is a projection matrix to be learnt.

Each predicate r is represented by a 2D matrix operator $\mathbf{M}_r \in R^{n*n}$. To reduce the learning parameters, *DistMult* restricts $\mathbf{M}_r$ to be a diagonal matrix. Using these representations, the scoring function of a query $r(e_1, e_2)$ is computed by a basic bilinear scoring function S:

$$S_{(e_1,r,e_2)} = \mathbf{y}_{e_1}^T \mathbf{M}_r \mathbf{y}_{e_2} \tag{6.14}$$

The training objective is to minimise the following margin-based ranking loss:

$$\mathcal{L}(\Omega) = \sum_{(e_1,r,e_2)\in T} \sum_{(e_1',r,e_2')\in T'} max(S_{(e_1',r,e_2')} - S_{(e_1,r,e_2)} + 1, 0) \tag{6.15}$$

where (e_1, r, e_2) is a positive query in the positive training dataset T and (e_1', r, e_2') is a corrupted query of the positive query (e_1, r, e_2) in the negative training dataset T'.

Note that *DistMult* can represent predicate embeddings also in the form of vectors (as in *TransE* [Bordes et al., 2013]), in which case the relation composition score is calculated as addition. Otherwise, using the matrix representations (as in *Bilinear* mentioned earlier), the relation composition score is calculated as multiplication. After learning the embedding representations of

predicates and constants, the rule extraction is performed by selecting a set of rules with high relation composition scores among possible rules. The score indicates the confidence level of the rule.

Its rule extraction algorithm is summarised below:

1: **Input:** $KB = \{(e_1, r, e_2)\}$, relation set R
2: **Output:** Candidate rules Q
3: **for each** r in R **do**
4: Select the set of start relations $S = \{s : \mathcal{X}_s \cap \mathcal{X}_r \neq \varnothing\}$
5: Select the set of end relations $T = \{t : \mathcal{Y}_t \cap \mathcal{Y}_r \neq \varnothing\}$
6: Find all possible relation sequences
7: Select the K-NN sequences $P' \subseteq P$ for r based on $dist(\mathbf{M}_r, \mathbf{M}_{p_1} \circ \cdots \circ \mathbf{M}_{p_n})$
8: Form candidate rules using P' where r is the head relation and $p \in P'$ is the body in a rule
9: Add the candidate rules into Q
10: **end for**

Fig. 6.3 The rule extraction algorithm of *DistMult* from [Yang et al., 2015]. The algorithm extracts K rules (of the form '2-hop transitive rule') for each predicate in the KB. For each predicate, it selects a set of start relation S and a set of end relation T, such that these relations have domain overlap. Then, it finds all possible relation sequences and select the K best sequences according to the matrix computation. The rule is formed using the predicate as the head relation and the sequence as bodies.

The input of the rule extraction algorithm is the knowledge base and its output is a set of candidate rules Q. K possible rules are extracted for each predicate r. The expected rule of predicate r is a linked Horn rule with r as the head predicate (only 2-hop or 3-hop rules are considered in their experiments), such as $r(X,Y) : -s(X,Z), t(Z,Y)$, where s and t are predicates in the knowledge base. In line 4-6, the algorithm finds all possible relation sequences in the body atoms for each such rule. Line 4 selects the set of predicates that could be the predicates of the first body atom. Since the variable X is the subject of both r and s, it restricts that the subject domain of s and r must overlap. ($\mathcal{X}$ refers to the domain of the subject and $\mathcal{Y}$ refers to the domain of the object.) Similarly, in the last relation t in the body, $\{t : \mathcal{Y}_t \cap \mathcal{Y}_r \neq \emptyset\}$ in line 5 forces that the object domains of t and that of r must overlap. In line 6, the algorithm generates all possible body atom sequences P of r by computing different combinations of body atoms whose predicates fit this restriction. For example, one potential rule could be $r(X,Y) : -s(X,Z), q(Z,W), t(W,Y)$. In line 7, the relation composition of each sequence $P' \subseteq P$ is computed by adding or multiplying each predicate in the sequence. (i.e. The relation composition of $r(X,Y) : -s(X,Z), q(Z,W), t(W,Y)$ is $\mathbf{M}_s \cdot \mathbf{M}_q \cdot \mathbf{M}_t$, where $\cdot$ can be either addition or multiplication depending on whether embeddings are represented by vectors or matrices.) Then, the distance between $\boldsymbol{M}_r$ and relation composition is computed. The sequences with K closest distance are selected. The distance metric provides a way to rank potential sequences according to how relevant their composition is to the target relation. Each

extracted rule is created by combining head relation r and the selected sequence as body of the rule. (i.e. if the distance between $\mathbf{M}_r$ and $\mathbf{M}_s \cdot \mathbf{M}_q \cdot \mathbf{M}_t$ is the K closest distances among all possible sequences as body of the rule, then $r(X, Y) : -s(X, Z), q(Z, W), t(W, Y)$ is added to Q.)

DistMult is a more advanced system in terms of reasoning ability than the other systems described in this section because it is able to learn rules from a knowledge base, whereas other systems either do not support rules or rely on given background rules. However, we do not classify *DistMult* as a rule induction system, unlike systems in Section 6.2, because instead of generating rules during the training process, rules in *DistMult* are generated after training, which means that embeddings are fixed in the rule extraction process. Moreover, the demonstration of extracted rules in its evaluation shows that its extracted rules are limited to 2-hop/3-hop linked Horn rules. In particular rules capturing the similarity of predicates p and q (say), such as $p(X, Y)$:- $q(X, Y)$ are not able to be learned. As a result, *DistMult* can be considered to be more like a rule-mining system than a rule induction system.

6.2 Rule Induction

This section presents systems that induce rules and is divided to two subsections, these systems that are direct extensions of *NTP* (denoted as *NTPs* and covering *NTP 2.0* [Minervini et al., 2018], *GNTP* [Minervini et al., 2020a], *CTP* [Minervini et al., 2020b]) covered in Section 6.2.1, and other relevant learning systems (namely *NeuralLP* [Yang et al., 2017], *MINERVA* [Das et al., 2018] and *RNNLogic* [Qu et al., 2020]) covered in Section 6.2.2.

6.2.1 NTPs

There are three extensions of *NTP* with consistent high-level architecture as *NTP*, namely *NTP 2.0*, *GNTP* and *CTP*, which were introduced to tackle scalability limitations of *NTP*. Each system gets as inputs a knowledge base and template rules which define the allowed structures of the learned Horn rules as inputs and produces as outputs learnt embeddings, induced rules, proof scores for queries. The following paragraphs show how they differently tackle the scalability problems.

NTP 2.0

NTP 2.0 [Minervini et al., 2018] is a more efficient version of *NTP*. Its key idea is to reduce the number of facts unified with a body atom in the computational tree when answering a query. At

each body atom unification, given a body atom subgoal g (g is the body atom grounded by the substitution set), $NTP2.0$ identifies the K nearest atoms of g, instead of all facts. This reduces the unification facts from the full fact set to K facts, making it more scalable. To search for the nearest atoms, *NTP 2.0* uses the Hierarchical Navigable Small World *HNSW* [Malkov et al., 2013] [Boytsov and Naidan, 2013], which is a graph-based Approximate Nearest Neighbour Search algorithm *ANNS*. *HNSW* is used to construct the computational tree dynamically when answering each query.

NTP 2.0 uses a subset of facts to prove body literals. It finds the nearest neighbours using the *HNSW* algorithm based on randomly initialised embeddings, thus it is likely that the nearest neighbours would miss many good matches especially at the beginning of the training, so it might fall into a local optimum. Our systems also improve computational efficiency by selecting a subset of facts used in each body atom unification, but different from the *HNSW* algorithm that selects facts dynamically, our systems choose the subset of facts before rule induction training based on topics and subdomains. Thus, our selection method is more controllable, because we can check which predicates/constants are in a given topic/subdomain whereas their nearest neighbours keep changing and are affected by noise due to randomisation.

GNTP

GNTP [Minervini et al., 2020a] is a more scalable extension of *NTP 2.0*.

GNTP has three new features to improve the scalability: fact selection, rule selection and attention mechanism. It uses the same Nearest Neighbour Search (*NNS*) method as *NTP 2.0* to select the K closest facts for each body atom, except it uses a different variant of NNS algorithm [Johnson et al., 2017] based on exact L2-nearest neighbour search. The search index of *NNS* is updated every 10 batches, because it assumes that small updates made by gradient descent do not affect search indexes much. Apart from fact reduction, *GNTP* also reduces the number of rules used to prove each query. Given a query q, *GNTP* only unifies rules where head predicates are in the neighbourhood of q according to *NNS*. In addition, it introduces an attention mechanism to reduce the number of learning parameters when the predicate number in the given knowledge base is lower than the dimension of embedding vectors. The size of the attention vector is equivalent to the number of predicates in the knowledge base. Hence, if the predicate number is low, it is more parameter efficient to use attentions compared with embeddings. In attention-based embedding mode, induced predicates are defined

by attending over known predicates dynamically during training. Since *GNTP* is the closest related work to ours, we compare it with *TSNTP* in terms of scalability and template rule syntax.

Both *GNTP* and *TSNTP* tackle the scalability limitations of *NTP*. Both systems reduce the number of facts involved in body atom unification by selecting a relevant subset of facts, but in different ways. *GNTP* selects facts by keeping updating the nearest neighbours during training. However, this selection would be very noisy at the beginning, when all embeddings are randomly initialised. Thus, it usually leads to more epochs of training, i.e. 100 epochs is used in *GNTP* training. Also, its nearest neighbours need to be updated every 10 batches per epoch, which causes computation overheads. On the other hand, we use topics and subdomains generated by *FNTP* to select facts. These topics and subdomains do not update during training, because they are clustered based on well-trained embeddings. As a result, symbols with similar semantic meaning are selected to unify with each other from the beginning of the training. With this semantic information, we require fewer epochs of training for rule induction. We report training results after 50 epochs of *FNTP* training (around 7 times faster than rule induction training) and 50 epochs of rule induction. Apart from fact selections, *GNTP* also reduces the number of rules to prove each query. For each query $p(a, b)$, *GNTP* selects rules whose head predicates are neighbours of p. However, this rule selection method is very noisy at the beginning, because all embeddings are randomly initialised without semantic meanings. We believe that this might cause *GNTP* to fall into a local optimum at the early stage of the training. Our systems use all rules in the *TTRs*. Reducing the number of rules can be an extension to our approaches too. Furthermore, *GNTP* always finds the K nearest neighbours, where K is a hyperparameter and $K = \{1, 3, 5\}$ in its evaluation. However, selecting only K facts could miss some closely relevant facts in unification. In our systems, the reduced number of facts for unification depends on how many facts there are in the given topic and subdomain. So this number varies for different queries, depending on the size of the cluster they belong to. In our evaluation, the fact number given a topic and a subdomain is around 100 on average, more than K used in *GNTP*, but covering more highly related facts. However, although we unify with more facts at each body unification, we only expand the best branch for each atom while *GNTP* expands K branches with different substitution sets. Overall, we achieve similar scalability improvement as *GNTP* and the details of the scalability comparison is presented in Chapter 7.

In terms of template rule syntax, *GNTP* only supports definite non-topical template rules, which is likely to learn duplicate rules as shown by experiment results. Our works support more

flexible template rules by allowing topics to be specified in the rule body atoms. Thus, each copy of a *TTR* can focus on inducing rules with respect to given topics, avoiding duplication. Also, our *NAF TSNTP* supports normal rules with the negation-as-failure syntax, which could make definite rules more specific by adding a negated atom.

CTP

CTP [Minervini et al., 2020b] is an extension of *NTP* that generates a set of rules dynamically during learning for proving each query, instead of using all rules in the given template rules. It extends from the original *NTP* [Rocktäschel and Riedel, 2017], not the more scalable *GNTP* [Minervini et al., 2020a] which also reduces the facts in unification, so *CTP* is less scalable than *GNTP*.

The main change of *CTP* is the *OR* module, as illustrated in Figure 6.4.

1: **function** $\texttt{or}(G, d, S)$
2: **for** $\mathbf{H} :- \mathbb{B} \in \texttt{select}_\theta(G)$ **do**
3: **for** $S \in \texttt{and}(\mathbb{B}, d, \texttt{unify}(\mathbf{H}, \mathbf{G}, S))$ **do**
4: **yield** S

Fig. 6.4 The *OR* module of *CTP*. The only difference between *CTP* and *NTP* is the second line. *CTP* has a *select* module that selects a set of rules according to the query G, while *NTP* uses all rules in the template rules.

Given a ground query, the *select* module can generate a set of rules according to the given template rules. For example, given a template rule $\#1(X, Y) : -\#2(X, Z), \#3(Z, Y)$ and a query $x(a, b)$, *CTP* generates a rule for the query according to its predicate x, in the form of $f_0(\mathbf{x})(X, Y)$:- $f_1(\mathbf{x})(X, Z), f_2(\mathbf{x})(Z, Y)$, where $f_i(\mathbf{x})$ is a differentiable function that, given an embedding $\mathbf{x}$, returns the embedding of the i-th induced predicate. The same set of rules would be generated for queries with the same predicate. Each query is linked to a set of template rules.

CTP proposes three different definitions of the function f_i: neural goal, attentive goal and memory-based function. In each definition, $f_i(\mathbf{x})$ includes some training parameters, so that this function is optimised during training. For example, in the neural goal function, $f_i(\mathbf{x}) = \mathbf{W}_i\mathbf{x} + \mathbf{b}$, where $\mathbf{x} \in \mathbb{R}^k$, $\mathbf{W}_i \in \mathbb{R}^{k*k}$ and $\mathbf{b} \in \mathbb{R}^k$. In this function, $\mathbf{W}_i$ and $\mathbf{b}$ are learning parameters to be optimised and the output of this $f_i(\mathbf{x})$ is an induced embedding. In the attentive goal function, $f_i(\mathbf{x}) = \boldsymbol{\alpha}\boldsymbol{E}_R$, where $\mathbf{E}_R$ represents the embedding matrix of all predicates and α is an attention vector to be learned. In the memory-based function, $f_i(\mathbf{x}) = \boldsymbol{\alpha}\boldsymbol{M}_i$, where $\mathbf{M}_i$ represents the ith predicates of a set of given rules and $\boldsymbol{\alpha}$ is an attention vector to be learnt.

The main strength of *CTP* is that it can select a set of rules by differentiable functions, instead of using all rules. N rules are generated specifically for each predicate based on the f_i function, so each generated rule is a relevant rule of a predicate by default. However, this means that if there are many predicates in the knowledge base, many rules would be created. Its scalability is still limited, because *CTP* needs to unify with all facts at each body atom unification. Although our works do not select rules, they also tackle the scalability limitation by unifying with facts with matched topics and subdomains in each body atom unification. In our work, the number of rules is fixed by the number of template rules and these are used in a passive way; all rules are used in each query's proof and the most relevant rule is selected.[4]

6.2.2 Other Works

We present here three systems that induce rules, namely *NeuralLP*, *MINERVA* and *RNNLogic*. We have selected these systems to illustrate three different approaches to rule induction, namely recurrent neural networks, reinforcement learning and EM-based optimisation.

NeuralLP

NeuralLP [Yang et al., 2017] is another system that performs query answering for query completion. For a partial complete query, it identifies the missing constants by finding a set of linked Horn rules, generated by a recurrent neural network, without the addition of template rules. *NeuralLP* takes a knowledge base *KB* with facts stored as triplets as an input, and outputs a set of induced first-order rules.

NeuralLP uses the same representation as *TensorLog* [Cohen et al., 2017], namely that each entity in the entities set E is presented by a $|E|$-dimension one-hot embedding, $\mathbf{v}_i$. Each Relation r is presented by a $|E| * |E|$ matrix, $\mathbf{M}_r$, with values either 0 or 1, such that its (i, j) entry is 1 if and only if $r(i, j)$ is in the knowledge base, where i is the i-th entity and j is the j-th entity. For any entity $X = x$, the logical rule inference for each rule l, in the form of $r(Y, X) \rightarrow p(Y, Z) \wedge q(Z, X)$, can be computed by matrix multiplication $\mathbf{M}_p \cdot \mathbf{M}_q \cdot \mathbf{v}_x$, which generates the score of l, denoted as $\mathbf{s}_l$. The non-zero entries of the vector $\mathbf{s}_l$ equals the set of y where there exists z that $p(y, z)$ and $q(z, x)$ is true. As a result, given a partial query with

[4]The rest of comparison made when comparing *GNTP* with our work holds for *CTP* as well.

entity x, the score for missing entity y is given by Eq. 6.16:

$$\mathbf{s} = \sum_{l} \alpha_l \mathbf{s}_l, \qquad score(y|x) = \mathbf{v}_y^T \mathbf{s} \tag{6.16}$$

where $\mathbf{v}_y$ is the embedding of y; $\mathbf{s}$ represents the scores over all constants conditioned on entity x of the probability that these constants are the missing entity. These scores are the weighted average of $\mathbf{s}_l$ over all rules $l \in \mathcal{L}$, where the weighted average is controlled by a learnable attention vector $\boldsymbol{\alpha}$. However, this computation of $\mathbf{s}$ is based on known rules.

To learn logical rules, a recurrent neural network is used to model a set of rules with the maximum length T. In the recurrent formulation, it uses an auxiliary memory vector $\mathbf{u}_t$ to record the proof score up until the t-th atom of each rule. At each inference step, the partial inference results is held in the memory, i.e. $\{\mathbf{u}_0, ..., \mathbf{u}_t, ..., \mathbf{u}_{T+1}\}$. Initially, the memory vector is set to the given entity $\mathbf{v}_x$.

$$\mathbf{u}_0 = \mathbf{v}_x \tag{6.17}$$

At each step as described in Equation 6.18, the model firstly computes a weighted average of previous memory vectors using the memory attention vector $\mathbf{b}_t$. Then, the model applies all predicates to the weighted average of previous memory, so that the previous rules are extended with all possible predicates. The weighted average is controlled by a trainable attention vector $\mathbf{a}_t$. This formulation allows the model to apply the TensorLog operators on all previous partial inference results, instead of just the last step's.

$$\mathbf{u}_t = \sum_{k}^{|R|} a_t^k \mathbf{M}_{R_k} (\sum_{\tau=0}^{t-1} b_t^\tau \mathbf{u}_\tau) \text{ for } 1 \leq t \leq T \tag{6.18}$$

The attention vectors $\{\mathbf{a}_t | 1 \leq t \leq T\}$ and $\{\mathbf{b}_t | 1 \leq t \leq T+1\}$ are learnable parameters and $\mathbf{M}_{R_k}$ is the operator of the k-th relation. a_t^k is the attention value of predicate k at the t-th step of the *RNN*. b_t^τ is the attention value of memory vector $\mathbf{u}_\tau$, at the t-th step of the *RNN*. This $\mathbf{u}_t$ creates a set of rule with length t. At the final step, the model computes a weighted average of all memory vectors, as shown in Equation 6.19, thus using attention to select the proper rule length.

$$\mathbf{u}_{T+1} = \sum_{\tau=0}^{T} b_{T+1}^\tau \mathbf{u}_\tau \tag{6.19}$$

The learning objective is to maximise $\mathbf{v}_y^T \mathbf{u}$ and therefore the $score(y|x)$ (see Eq. 6.16).

NeuralLP is similar to our work in that it learns first-order rules through *ANNs* and links

each induced predicate to a score, though by a different method. In *NeuralLP*, the score of an induced predicate is a parameter to be learnt, which is controlled by an attention vector. The score of a query is the aggregated scores of multiple rules controlled by the recurrent neural networks as specified by Equation 6.19. The multiple attention vectors involved in answering each query using the recurrent neural networks make it hard to understand the decision making process. In our system, the score of an induced predicate is not learnt directly and it is determined by unifying it with other predicates. The learning parameters are the embeddings and these embeddings can be visualised to inspect whether semantic meaning is captured correctly. Also, although both systems use embeddings to represent the knowledge base, embeddings are used in different ways. *NeuralLP* uses one-hot embeddings and essentially they are just ID encodings, so embeddings do not carry any semantic information and they can only be hard-unified. In contrast, our system learns embeddings and embeddings carry semantic information. Embeddings that are close according to Euclidean distance indicates they are similar and can be used interchangeably in soft-unification. In addition, *NeuralLP* only supports definite linked Horn rules, whereas our systems do not restrict that and support normal rules.

MINERVA

MINERVA [Das et al., 2018] is a query answering system for query completion. Its input is a knowledge base KB with facts stored as triplets, which then constructs a knowledge graph $\mathcal{G}$ with both relations and inverse relations. Given an incomplete query, *MINERVA* aims to find paths in a knowledge graph to identify the missing entity. A path is the sequence of edges involved in a knowledge graph from the start node to the end node, which can be viewed as a linked propositional rule. After training, it outputs a sequence of ground paths for each query and trained embeddings of predicates and constants. The path of each partial query is identified by reinforcement learning.

MINERVA is a neural reinforcement learning system on a knowledge graph. Given an incomplete query, in the form of $q(a, ?)$ or $q(?, a)$ and the missing entity $? = b$, the agent starts from the node a and aims to reach the node b within the path length restriction by finding viable paths in $\mathcal{G}$. The learning environment is a finite horizon, deterministic partially observed Markov decision process on $\mathcal{G}$, which is a 5-tuple (S, O, A, δ, R) that specifies states, observations, actions, transitions and rewards correspondingly. Each state S encodes the incomplete query, the missing entity of the query and the current node of the agent. The observations O contain

the known information for the agent, including the incomplete query and the current node of the agent, but the answer b remains unknown for the agent. Actions A of the state S include all outgoing edges the agent could take from the current node of the agent. The transition function δ returns a new state S', given a state and an action. In the end, if the agent is at the expected node b, it would receive a reward $+1$ for R.

In order to solve the finite horizon deterministic partially observable Markov decision process mentioned above, a randomised non-stationary history-dependent policy $\pi = (\mathbf{d}_1, \mathbf{d}_2, ..., \mathbf{d}_{T-1})$ is devised, where $\mathbf{d}_t$ is a function that receives a history state at time t, H_t, as an input, and outputs the probabilistic distribution of taking each action in $\mathbf{A}_{S_t}$; history $H_t = (H_{t-1}, A_{t-1}, O_t)$ records the sequence of observations and actions taken up until time t. These policies are parameterised by long short-term memory network (LSTM) [Hochreiter and Schmidhuber, 1997]. Given a query relation, the policy networks select a discrete action from all available actions ($\mathbf{A}_{S_t}$) based on the history embedding $\mathbf{h}_t$. This decision is made by a two-layer feed-forward network with ReLU nonlinearity.

For each policy network π_θ, the training aims to maximise the expected reward using *REINFORCE* algorithm [Williams, 1992]:

$$J(\theta) = \mathbb{E}_{(e_1,r,e_2)\sim D}\mathbb{E}_{A_1,...,A_{T-1}\sim\pi_\theta}[R(S_T)|S_1 = (e_1, e_1, r, e_2)] \tag{6.20}$$

where (e_1, r, e_2) is a query in the training dataset D and the subject e_2 is missing; $S_1 = (e_1, e_1, r, e_2)$ represents the initial state of the agent, where given the query $r(e1, ?)$, it starts from the node $e1$ and aims to reach the expected destination, the node e_2; $A_1, ..., A_{T-1} \sim \pi_\theta$ is the sequence of actions generated by the policy network π_θ for each query; $R(S_T)$ indicates whether the query gets a reward, depending on whether the node e_2 is reached at step T.

MINERVA uses reinforcement learning to perform rule induction. It optimises the path selection via reinforcement learning for each query, which starts from the given constant of a partial query and keeps identifying the best path (action) to take from all actions so that it could reach the missing entity of the query. In this process, the path taken by each query can be viewed as a linked propositional rule, starting with the ground query as the head of the rule, followed by the sequence of actions (including inverse relations) taken as predicates of the body atoms in the rule. Although the authors of [Das et al., 2018] claim that *MINERVA* can induce first-order rules, the paper did not state how first-order rules are generated. If first-order rules

are generated by turning the actions in propositional rules into atoms with variables abstracting the entities used between hops, then these rules could be over-generalised. The path exploration of *MINERVA* is in contrast to our systems that learn embedding representations via a backward chaining computational tree. In our systems, each body atom unification is equivalent with taking a path in *MINERVA*, which identifies the best matched fact among a set of facts selected by topics and subdomains. This topic and subdomain based fact selection helps the systems to narrow down the number of facts considered during rule induction. This fact reduction is similar with the process of exploring a subset of paths that is likely to lead to the missing constant in *MINERVA*.

RNNLogic

RNNLogic [Qu et al., 2020] is a neural symbolic system that induces rules through EM-based optimisation. It performs query answering tasks for query completion. Given a knowledge graph $\mathcal{G}$ with partial complete queries, each partial query q in the form of $q = r(s, ?)$ or $q = r(?, o)$, where the answer a of the question mark ? is the expected constant t, the goal of *RNNLogic* is to learn to predict the correct answer $a = t$. *RNNLogic* identifies the answer a by modelling the probabilistic distribution $p(\mathbf{a}|\mathcal{G}, \mathbf{q})$ through learning a set of linked Horn rules. In summary, *RNNLogic* receives a knowledge graph as input and outputs a set of learnt compositional logic rules and the missing entity of each partial complete query.

In *RNNLogic*, a knowledge graph is represented by embeddings, where each predicate and constant is linked to an embedding. The following paragraphs will explain how rules are induced in *RNNLogic*, including the probabilistic formalisation of the link predication task and the EM-based optimisation process.

The overall objective function of *RNNLogic* is identifying the target distribution $p(\mathbf{a}|\mathcal{G}, \mathbf{q})$ of each query $\mathbf{q}$, where $\mathbf{a}$ is the expected missing entity. This objective is achieved by jointly training a rule generator and a reasoning predictor together, where each has its own learning parameters, $\boldsymbol{\theta}$ and $\mathbf{w}$ respectively. In *RNNLogic*, a set of linked Horn rules $\mathbf{z}$ is generated by a recurrent neural network $RNN_{\boldsymbol{\theta}}$ where $\boldsymbol{\theta}$ are trainable parameters in the *RNN* network. The maximum length of each rule and the number of rules generated for each predicate are given by users. Using the predicates in the given *KB*, the *RNN* network generates a sequence of predicates to be converted to Horn rules. For example, it could generate a sequence p, t, s, which could be converted to $p(X, Y)$:- $t(X, Z), s(Z, Y)$. For each query $\mathbf{q}$ with predicate h,

the rule generator $p_{\boldsymbol{\theta}}$ defines a multinomial distribution over the rule set $\mathbf{z}$ conditioned on $\mathbf{q}$, denoted as $p_{\boldsymbol{\theta}}(\mathbf{z}|\mathbf{q})$, to select rules for $\mathbf{q}$.

$$p_{\boldsymbol{\theta}}(\mathbf{z}|\mathbf{q}) = Mu(\mathbf{z}|N, RNN_{\boldsymbol{\theta}}(\cdot|h)) \tag{6.21}$$

where Mu indicates multinomial distributions, N is a hyperparameter for the size of the set $\boldsymbol{z}$, and $RNN_{\boldsymbol{\theta}}(\cdot|h)$ is the distribution over all potential generated rules with a rule head h. The reasoning predictor $p_{\mathbf{w}}$ defines $p_{\mathbf{w}}(\mathbf{a}|\mathcal{G}, \mathbf{q}, \mathbf{z})$, such that for a query $\mathbf{q}$, it uses a set of selected rules $\mathbf{z}$ to reason on a knowledge graph $\mathcal{G}$ and generates the likelihood that the answer is $\mathbf{a}$. Thus, $p(\mathbf{a}|\mathcal{G}, \mathbf{q})$ is computed using Equation 6.22:

$$\begin{aligned} p_{\mathbf{w},\boldsymbol{\theta}}(\mathbf{a}|\mathcal{G}, \mathbf{q}) =& \sum_{\mathbf{z}} p_{\mathbf{w}}(\mathbf{a}|\mathcal{G}, \mathbf{q}, \mathbf{z}) p_{\boldsymbol{\theta}}(\mathbf{z}|\mathbf{q}) \\ =& \mathbb{E}_{p_{\boldsymbol{\theta}}(\mathbf{z}|\mathbf{q})}[p_{\mathbf{w}}(\mathbf{a}|\mathcal{G}, \mathbf{q}, \mathbf{z})] \end{aligned} \tag{6.22}$$

The objective function is to maximise the $p_{\mathbf{w},\boldsymbol{\theta}}(\mathbf{a}|\mathcal{G}, \mathbf{q})$ for each query $\mathbf{q}$ in the training dataset p_{data}, as defined by Equation 6.23:

$$\begin{aligned} max_{\boldsymbol{\theta},\mathbf{w}} \mathcal{O}(\boldsymbol{\theta}, \mathbf{w}) &= \mathbb{E}_{(\mathcal{G},\mathbf{q},\mathbf{a})} \sim p_{data}[log p_{\mathbf{w},\boldsymbol{\theta}}(\mathbf{a}|\mathcal{G}, \mathbf{q})] \\ &= \mathbb{E}_{(\mathcal{G},\mathbf{q},\mathbf{a})\sim p_{data}}[log \mathbb{E}_{p_{\boldsymbol{\theta}}(\mathbf{z}|\mathbf{q})}[p_{\mathbf{w}}(\mathbf{a}|\mathcal{G}, \mathbf{q}, \mathbf{z})]] \end{aligned} \tag{6.23}$$

where $\boldsymbol{\theta}$ and $\mathbf{w}$ are the training parameters.

The optimisation of *RNNLogic* can be summarised by the algorithm below.

---

while not converge **do**
  For each instance, use the rule generator p_θ to generate a set of rules $\hat{\boldsymbol{z}}$ ($|\hat{\boldsymbol{z}}| = N$).
  For each instance, update the reasoning predictor p_w based on generated rules $\hat{\boldsymbol{z}}$
    E-step:
  For each instance, identify K high-quality rules $\hat{\boldsymbol{z}}_I$ from $\hat{\boldsymbol{z}}$ according to $H(rule)$
    M-step:
  For each instance, update the rule generator p_θ according to the identified rules
end while
During testing, for each query, use p_θ to generate N rules and feed them into p_w for prediction.

Fig. 6.5 The workflow of *RNNLogic* [Qu et al., 2020]. It repeats the EM optimisation process until convergence. It firstly use a rule generator to generate rules. Then, it updates the reasoning predictor based on generate rules. At the E-step, it identifies the K best rules from generated rules. At the M-step, it update the rule generator according to the identified rules.

The optimisation involves four steps repeatedly. The first step uses the rule generator $p_{\boldsymbol{\theta}}$ to generate a set of rules $\hat{\boldsymbol{z}}$ according to the *RNN* network. The second step uses $\hat{\boldsymbol{z}}$ to update the

reasoning predictor $p_{\mathbf{w}}$. This update is achieved by maximising the objective function $\mathcal{O}(\boldsymbol{\theta}, \mathbf{w})$ (Equation 6.23) with respect to $\mathbf{w}$. The third step, the *E-step*, identifies K high-quality rules $\hat{\boldsymbol{z}}_{\boldsymbol{I}}$ from $\hat{\boldsymbol{z}}$ according to a heuristic scoring function: $H(rule)$. Details of $H(rule)$ can be found in [Qu et al., 2020]. In the final step, the *M-step*, the rule generator $p_{\boldsymbol{\theta}}$ is updated using the K high-quality rules $\hat{\boldsymbol{z}}_I$. This update is achieved by maximising the objective function $\mathcal{O}(\boldsymbol{\theta}, \boldsymbol{w})$ (Equation 6.23) with respect to $p_{\boldsymbol{\theta}}$.

RNNLogic uses EM-based algorithm for optimisation, which is unique among other works in this chapter. EM-algorithm enables *RNNLogic* to optimise the reasoning predictor and rule generator separately at each epoch, but it is not end-to-end differentiable. Similar to *NeuralLP*, *RNNLogic* also uses multiple rules to prove queries, where the proof score of each query is the aggregated proof score over multiple rules generated by the *RNN* network (which is a set of linked Horn rules). Although the system could induce rules using the rule generator as specified by Equation 6.21, the involvement of multiple rules for proving each query makes the system less interpretable, unlike our systems where one query is proved by one rule or one fact.

Summary. In Table 6.3 we summarise all the systems described in this chapter with respect to five main features: architecture, learning of embedding representations, if rules are supported, and interpretability. A more detailed quantitative evaluation of these systems is given in Chapter 7.

	Architecture	Learn Embedding Representations	If Rules are Supported?	Interpretability
ComplEx	ANN	Yes	No	No
ConvE	CNN	Yes	No	No
KALE	ANN	Yes	Given Background rules	No
ASR	Adversarial training	Yes, when used jointly with other systems	Given Background rules	No
DistMult	Rule mining	Yes	Rule mining as post-processing	Yes
NTP 2.0	ANN	Yes	Rule induction	Yes
GNTP	ANN	Yes	Rule induction	Yes
CTP	ANN	Yes	Rule induction	Yes
NeuralLP	RNN	No, use predefined embeddings	Rule induction	Yes
MINERVA	Reinforcement learning	Yes	Rule induction	Yes
RNNLogic	EM-based optimisation	Yes	Rule induction	Yes

Table 6.3 The summary of related works covered in this chapter from five aspects: architectures, learning of embedding representations, if rules are supported, rule syntax and interpretability.

Chapter 7

Evaluation

In this chapter, we present the evaluation results of our three architectures, namely *TNTP*, *TSNTP* and *NAF TSNTP*. They are evaluated using benchmark datasets and compared with other related state-of-art baseline systems.

In the following sections, we firstly describe the datasets used in our experiments. We then describe our experimental settings, including the evaluation metrics, evaluation procedure, hyperparameters and template rules used. After this, we present our evaluation results, in terms of accuracy of query answering prediction, time and space improvement and rule induction quality. We conclude with an empirical analysis of the effects of some hyperparameters on the performance of our systems.

7.1 Datasets

This section covers the benchmark datasets used in our experiments, namely *Countries* [Bouchard et al., 2015], *Nations*, *Kinship*, *UMLS* [McCray, 2003] and *FB122* [Guo et al., 2016], since these are widely used in the evaluation of our baselines.

Recall that, as discussed in Chapter 2, all facts in a dataset are considered to be true. False facts are those which are absent, thus following the closed-world assumption. Facts in a dataset are divided into a training set, a validation set and a test set. The training set is used to learn embeddings and induce rules, whereas the validation set is used for tuning the hyperparameters. The test set is used for evaluating the trained embeddings and the induced rules. The *HITS* and *MRR* metrics are used for this purpose (see Section 2.4.2).

Table 7.1 summarises the key features of these datasets. The details and analysis of each dataset are presented in the following paragraphs.

	Countries	Nations	Kinship	UMLS	FB122
Predicate #	2	56	26	49	122
Constant #	272	14	104	135	9,738
Positive Facts #	1,159	2,565	10,686	6,529	112,476
Maximum Facts # according to the Signature	149,058	10,976	281,216	893,025	11,569,094,568
% of the Positive Fact Number over the Maximum Fact Number	0.7%	23.4%	3.8%	0.7%	0.0009%
Size of Training Set	S1 1111 S2 1063 S3 979	1,592	8,544	5,216	91,638
Size of Validation Set	24	199	1,068	652	9,595
Size of Test Set	24	201	1,074	661	S1 5,057 S2 6,186 S3 11,243
Identifies Expected Rules	Yes	No	No	No	Yes

Table 7.1 A summary of key features of datasets used in our evaluation, including the predicate number, the constant number, the positive fact number, the maximum fact number according to the signature, the percentage of the positive fact number over the maximum fact number, the size of training/validation/test sets and if there are expected rules.

The first two rows state the number of predicates and constants in the signature of the dataset. The third row states the number of facts in the knowledge base. The fourth row is the maximum number of facts that could be constructed from the knowledge base signature, which is given by the formula $number\ of\ predicates \times (number\ of\ constants)^2$ (recall that all facts are binary atoms[1]). The fifth row is the percentage of the positive fact number over the maximum fact number. The remaining three rows specify the sizes of the training, validation and test sets respectively. Note that the *Countries* dataset includes three training sets, S1, S2, S3 and one test set, whereas the *FB122* dataset has one training set and three test sets. The last row indicates whether the knowledge base includes also any 'expected' rules.

Countries. This dataset was initially proposed in [Bouchard et al., 2015] and extended in [Nickel et al., 2015] to three training tasks to test the reasoning ability of the systems. It is a geographical dataset that records the location and neighbourhood relations between countries, subregions and regions, through two predicates $located\_in$ and $neighborOf$. The signature contains 272 constants, comprising of 244 countries, 5 regions (continents) and 23 subregions (such as Eastern Europe). In this small dataset it is assumed that each country is located in

[1]Note that extra constraints might be applied to restrict some predicate and constant combinations. This number does not consider these constraints and indicates a theoretic maximum number, which might be larger than the facts number allowed under various constraints.

exactly one subregion; each subregion is located in exactly one region; however, each country may have zero or more neighbours.

In order to evaluate the reasoning ability of a learning system, the dataset includes the three tasks, $S1$, $S2$, $S3$, of increasing difficulty. These have the same learning goal, i.e. predict $locatedIn(c, r)$, where c ranges over all countries and r ranges over the 5 regions. However, the training sets of these tasks are slightly different. *Countries* has a 'basic' training set consisting of all facts in the knowledge base. To increase the level of difficulty across the three tasks, some selected facts are removed from the 'basic' training set, so they must be proved using some learned rules. The dataset comes with an 'expected rule' for each task, which is used to inform the structure of the template rules that need to be induced. The task $S1$ could be solved by a transitive rule *locatedIn(X, Y) :- locatedIn(X, Z), locatedIn(Z, Y)*. The task $S2$ removes ground facts of $locatedIn(c, s)$, where c is a test country and s is a subregion, so queries cannot be answered using the rule in $S1$ and needs to be answered by a rule *locatedIn(X, Y) :- neighborOf(X, Z), locatedIn(Z, Y)*. In task $S3$, more facts are removed and queries needs to be answered by a three-hop rule *locatedIn(X, Y) :- neighborOf(X, Z), neighborOf(Z, W), locatedIn(W, Y)*.

Nations. The *Nations* dataset consists of a set of facts about nations and their features. The dataset was originally proposed in [Rummel, 1992] with a mixture of unary and binary predicates, but in [Rocktäschel and Riedel, 2017], all unary facts were removed. To be consistent with our baselines, *NTP* and its extensions[2], we use the same version of *Nations* dataset presented in [Rocktäschel and Riedel, 2017].

Nations is a good dataset on which to evaluate our systems. With 56 binary predicates and the given rule templates, it has a much bigger search space compared to *Countries*. On the other hand, it includes only 14 constants, so the use of subdomains might not be very effective, possibly making it less than ideal for evaluating *TSNTP*.

Kinship. The *Kinship* dataset [Denham and White, 2005] consists of a set of facts about kinship relations among people from the Alyawarra tribe in Central Australia. There are 26 binary predicates (kinship relation) and 104 constants (person) in the knowledge base. All predicates and constants are anonymised, so facts have the forms such as $term0(person0, person45)$.

[2]Other systems that we compare with do not use *Nations* dataset, because they focus on binary facts.

Kinship is a good dataset on which to evaluate our systems, especially *TSNTP* and *NAF TSNTP*, since it has sufficient predicates and constants to demonstrate how topics and subdomains can be used to improve computational efficiency. Differently from *Nations*, where the knowledge base contains 23.4% of facts out of all potential facts that can be generated from the signature, the *Kinship* knowledge base contains only 3.8% of potential facts. In this case, high *HITS* values of positive queries indicate a more robust trained model, because the number of corrupted facts in the test set is bigger. Another strength of *Kinship* is that the data is anonymised, such that a human cannot understand the meaning of each symbol. Thus, there is no human bias in selecting topics and subdomains and interpreting decoded rules.

UMLS. The *UMLS* dataset consists of a set of facts from the Unified Medical Language System [McCray, 2003], a biomedical ontology. Similarly to *Kinship*, *UMLS* is a good dataset on which to evaluate our systems. It has similar numbers of predicates and constants as *Kinship*, but the knowledge base contains only 0.7% of facts among potential facts given the signature. *UMLS* is not an anonymous dataset, so its decoded rules and clustering are interpretable for humans to check whether the induced rules make sense.

Freebase122. The *FB122* dataset [Guo et al., 2016] is a subset of the *Freebase* dataset [Bollacker et al., 2008] which stores general facts about the world. It is by far the largest knowledge base used in our evaluation. *FB122* stores facts about people, location and sports. There are 47 rues about triples in *FB122* that are known to be true and the dataset comes with two test sets, depending on whether they can be proved by one of the 47 rules or not. There is a single training set, so the training process is the same with the same hyperparameters and rule induction. *Test-I* is the harder of the two tests, consisting of 5,057 positive facts that cannot be proved by any of the 47 rules, whereas *Test-II* consists of 6,186 positive facts all of which can be proved by the 47 rules. Note that, although there are 47 known rules, to test the induction ability of our system, we do not use these rules[3]. For each test set the evaluation is based on our induced rules and trained embeddings, without the 47 known rules. Therefore, for *Test-II* our system has to learn some or all of the 47 rules in order to achieve a good accuracy, whereas for *Test-I* it has to induce some other rules. A third test set, *Test-III* is formed by joining *Test-I* and *Test-II* and its evaluation result represents the weighted average of evaluation results of *Test-I* and *Test-II*.

[3]Other systems, such as *GNTP*, also test the three datasets without using the known rules. We compare with them under the same settings.

FB122 is a good dataset on which to evaluate our systems and test their scalability. The number of facts in the *KB* is 17.2 times more than that of *UMLS* (112,476 vs 6,529), which is a standard benchmark dataset to test rule induction. In terms of predicate and constant numbers, its 122 predicates and 9,738 constants are also greater than the 49 predicates and 135 constants in *UMLS*. This dataset motivated us to develop *TSNTP* that aims to support large knowledge bases.

7.2 Experiment Settings

In this section, we present the settings of our experiments, including the evaluation metrics used, evaluation procedure, hyperparameters involved and the template rules used for each dataset.

7.2.1 Evaluation Metrics

We consider two main metrics: *AUC_PR* [Davis and Goadrich, 2006] and *HITS* [Bordes et al., 2011]. Both are standard metrics used to evaluate accuracy of query answering predictions: *AUC_PR* is used for the *Countries* dataset and *HITS* is used for the other datasets. For a fair comparison, these choices of metrics are the same as those used by our baselines. In what follows, we introduce the details of the two metrics.

AUC_PR

Following [Rocktäschel and Riedel, 2017], the accuracy of query answering predictions in the *Countries* dataset is evaluated using *AUC_PR* [Davis and Goadrich, 2006] (see Chapter 2.4.2 for details). This metric measures the area under the curve (AUC) of a precision-recall curve (PR). According to [Davis and Goadrich, 2006], *AUC_PR* is particularly good at evaluating datasets with an unbalanced number of positive and negative (corrupted) examples. Recall from Section 2.2, *precision* is very sensitive to the number of false positive predictions and *recall* is very sensitive to the number of false negative predictions, especially when the number of true positive predictions is low. The *PR* curve better represents the performance over a dataset with more negative examples than positive examples, because *precision* and *recall* are more sensitive to the quality of predicting positive examples. Note that in our experiments, negative examples are corrupted examples.

HITS

Following [Rocktäschel and Riedel, 2017], the *Nations*, *Kinship*, *UMLS* and *FB122* datasets are evaluated using the *HITS* metric. This metric measures the rank of a (positive) test query when its proof score is compared with the proof scores of its corrupted queries (see Chapter 2.4.2). Ideally, the rank of each positive query should be 1 when compared with its corruptions, meaning its proof score is the highest.

7.2.2 Evaluation Procedure

In what follows, we describe our evaluation procedure.

As stated in Chapters 3, 4, and 5, *TNTP* and *TSNTP* rely on the topics and subdomains information generated using the embeddings learned by *FNTP*, whereas *NAF TSNTP* relies on trained embeddings and rules generated by *TSNTP*. In our evaluation, we maximise the common parts among the various systems, such that for each dataset, *TNTP* and *TSNTP* use the same topics, and *NAF TSNTP* uses the trained results from *TSNTP* as its starting point.

In all cases, the training set is used to train the learning systems and the validation set is used to tune hyperparameters. For each system and each dataset, the hyperparameters vary within fixed ranges (see Section 7.2.3). Finally, after selecting the best hyperparameters, the results of the query answering task are generated using the test set. These are generated by performing 5 runs that use different random embedding initialisations. We consider 5 runs only, because our standard errors are small. We also record the best result from the 5 runs, because most of systems that we compare with reported one result only, rather than a mean value for several repetitions.

7.2.3 Hyperparameters

To have a fair comparison with our baselines, and in paticular with the systems presented in [Rocktäschel and Riedel, 2017] and [Minervini et al., 2020a], we use the same hyperparameters as theirs whenever possible[4]. In the following paragraphs, we firstly introduce the common features used by all systems and the hyperparameters for topic and subdomain generation. Then, we present the hyperparameters specific to *TNTP/TSNTP/NAF TSNTP*.

[4]Most of their experiment settings are the same. When they differ, we use the experiment setting of [Minervini et al., 2020a], because it is the latest version.

Common Features. All systems use ADAM [Kingma and Ba, 2015] with a mini-batch size of 50 queries for optimisation. We use l_2 regularisation to optimise all trainable parameters and clip gradient values between $[-1.0, 1.0]$. All embeddings are initialised randomly using Xavier initialisation [Glorot and Bengio, 2010]. For each (positive) query, four corrupted queries are used during training and these corrupted queries are recreated at each epoch. For a fair comparison with other systems, we consider the same set of template rules and same number of copies as that used by the baselines (see details in Section 7.2.4).

Hyperparameters for Topic and Subdomain Generation. Since the aim of *FNTP* is to train the embeddings from which to generate topics and subdomains, achieving the highest accuracy of query answering predictions is not its goal. Thus, we keep the parameter tuning of *FNTP* as simple as possible. *FNTP* uses a fixed learning rate at 0.001, a fixed l_2 regularisation value at 0.001 and a fixed batch size, 50 queries per batch. Although using fixed hyperparameters, *FNTP* is still able to reach high accuracy of query answering prediction, which is a good basis to generate topics and subdomains.

After *FNTP* is trained, topics and subdomains are generated by clustering the trained embeddings. The number of topics and subdomains are also hyperparameters, whose ranges are $\{5, 10, 20\}$ and $\{10, 20, 50, 100, 200\}$ respectively. Note that the upper bounds of topic number and subdomain number are dependent on the numbers of predicates and constants in the dataset. When creating topics or subdomains using the trained embeddings of *FNTP*, we apply both *K-means* and *Agglomerative* clustering algorithms to generate different number of topics and subdomains. We found that the topics generated by *K-means* algorithm usually lead to better accuracy of query answering predictions, except for the *UMLS* dataset where the clusters generated by *K-means* concentrate on two topics whereas the clusters generated by *Agglomerative* are distributed more uniformly. As a result, except for the *UMLS* dataset, other datasets use the *K-means* clustering algorithm.

As shown in Section 7.4.3, which compares the different evaluation results for the query answering task with respect to different numbers of topics and subdomains, fewer topics and/or subdomains give better accuracy of query answering predictions. When more topics and subdomains are used, the computation time is faster with less memory usage, but the accuracy of query answering predictions could be slightly lower. When we compare our systems against the baselines, we pick the numbers of topics and subdomains that generate best accuracy of

query answering predictions. Once the topics and subdomains are selected, we then fine-tune the *TNTP* or *TSNTP* induction with respect to other hyperparameters.

Simplified Hyperparameters. We fine-tune the two hyperparameters, learning rate and l_2, only within a subset of the ranges used by *GNTP*, and keep the batch size fixed, due to limited computational resources. These are summarised in Table 7.2.

Hyperparameters	TNTP/TSNTP Ranges
Learning Rate	$\{0.001, 0.01, 0.1\}$
l_2	$\{0.001, 0.0001\}$
Batch Size	50

Table 7.2 The ranges of the three hyperparameters used by our systems, including the learning rate, l_2 value and batch size.

We first tune the learning rate using a fixed l_2 regularisation value and then use the best learning rate to further tune l_2. Although this method does not guarantee the best hyperparameter combination, it proved sufficient to achieve good overall accuracy of query answering predictions.

New Hyperparameters. Specific to our systems, we introduced four tunable hyperparameters, namely the number of topics and subdomains, α and β (for amplifying the rule learning) and two fixed hyperparameters for *NAF TSNTP*, namely Z and $NAF\_THRESHOLD$. The range of values for these hyperparameters is given in Table 7.3. The hyperparameter Z controls how many definite rules are converted to normal rules. We fix Z to 10 to concentrate on the 10 most frequently used rules. Z could in principle cover all $TTRs$, but the computation time grows linearly when more definite rules are considered. Similarly, for simplicity, we keep $NAF\_THRESHOLD$ fixed to 1.0, which is the mid-point of the allowed range [0, 2]. Recall that in *NAF TSNTP*, each normal rule has a ns score which measures the quality of the normal rule. If the ns score of a selected normal rule is less than or equal to $NAF\_THRESHOLD$, the normal rule could replace the definite rule from which it extends.

Hyperparameters	Ranges
α	$\{20, 50, 100, 200\}$
β	$\{3, 5, 100, 1000\}$
Z	10
NAF_THRES	1.0

Table 7.3 The range of four hyperparameters specific to our systems, including α, β, Z and NAF_THRES.

Changed Hyperparameters. We also changed the two hyperparameters, k_{max} and the number of epochs for our systems, leading us to select $k_{max} = 1$ (except for the *Countries* dataset, which would have poor accuracy of query answering predictions if $k_{max} = 1$ is used, due to its small data size) and running for 50 epochs for training *FNTP* and 50 epochs for training *TNTP/TSNTP*.

Note that *NTP* sets k_{max} to 10 and *GNTP* tunes it in the range $\{1, 3, 5\}$.[5] In our systems, we are able to set k_{max} to 1 and still induce rules with good quality, because we use α and β to encourage queries to be proved by rules. Without these amplification hyperparameters, the rule induction quality cannot be maintained when $k_{max} = 1$. The empirical evidence is presented in Section 7.4.1.

We use 50 epochs to train *FNTP* and another 50 epochs to train *TNTP/TSNTP* induction, whereas *NTP* uses 100 epochs directly. Note that *FNTP* training around 7 times faster than *TSNTP* induction training, because it does not involve rule induction. With given topic and subdomain information as induction bias, *TNTP/TSNTP* induction only requires 50 epochs to induce rules. In contrast, *NTP* does not have this information, so it needs more epochs to induce rules.

7.2.4 Template Rules

For each dataset, we list the template rules with the numbers of copies in brackets that we use in our evaluation (see Section 3.3.2 for details of template rules). These are the same template rules as those used by our baselines where applicable.

For *Countries* dataset, the template rules are as follows. Recall that there are only two predicates, so no topics are introduced.

(3) #1(X, Y) :− #2(Y, X).
(3) #1(X, Y) :− #2(X, Z), #2(Z, Y).
(3) #1(X, Y) :− #2(X, Z), #3(Z, Y).
(3) #1(X, Y) :− #2(X, Z), #3(Z, W), #4(W, Y).

In task $S1$, the first two template rules are given. In task $S2$, the first three template rules are given. In task $S3$, all template rules are given.

[5] In their work, k_f is used to denote the number of branches that will expand in the proof. It is similar with k_{max}.

The template rules for *Nations*, *Kinship*, and *UMLS* datasets are listed below.

(20) #1(X, Y) :– #2(X, Y).
(20) #1(X, Y) :– #2(Y, X).
(20) #1(X, Y) :– #2(X, Z), #3(Z, Y).

The template rules for *FB122* dataset are listed below.

(20) #1(X, Y) :– #2(X, Y).
(20) #1(X, Y) :– #2(Y, X).
(20) #1(X, Y) :– #2(X, Y), #3(X, Y).

Note that we our third template rule is different from the template rule $\#1(X, Y)$:- $\#2(X, Z)$, $\#3(Z, Y)$ used by *GNTP*.

For each of these four datasets, the template rules are converted to *TTRs*, as described in Section 3.3.2. As our evaluation is based on 5 runs, we use newly generated sets of *TTRs* in each run.

7.3 Evaluation Results

We compare the accuracy results of the query answering task of our systems against three groups of baselines: the first group includes the family of *NTP-based* systems (*NTP* [Rocktäschel and Riedel, 2017], *NTP 2.0* [Minervini et al., 2018], *GNTP* [Minervini et al., 2020a] and *CTP* [Minervini et al., 2020b]), the second group refers to systems that are capable of doing rule induction (*NeuralLP* [Yang et al., 2017], *MINERVA* [Das et al., 2018] and *RNNLogic* [Qu et al., 2020]) and the third group includes systems that cannot do rule induction (*ComplEx* [Trouillon et al., 2016], *KALE* [Guo et al., 2016], *ASR* [Minervini et al., 2017] and *DistMult* [Yang et al., 2015]). On the other hand, we compare the time and memory performance of our systems with respect only to *NTP* and *GNTP*, since our systems are inspired by the *NTP* approach. Then, we also discuss the quality of our induced rules and the effects of *NAF*.

7.3.1 Accuracy of Query Answering Predictions

We summarise the accuracy results for the query answering prediction task in four tables (Table 7.4-Table 7.7).

Table 7.4 shows the accuracy of the query answering prediction task on the *Countries* dataset for our *TNTP* system, based on the best result out of 5 runs using only one topic[6], and compares it with that of baselines for which results are published. We found that the limited number of predicates (2 only) in the Countries dataset affects the training by *FNTP* of embeddings of constants such that the subdomain clustering does not reflect their (geographical) semantic meaning. For this reason we considered just 1 subdomain, making *TSNTP* identical to *TNTP*, hence we have not included results for *TSNTP*. Also, we did not run *NAF TSNTP* since the expected rules for this dataset are all definite rules.

Dataset		Metrics	TNTP	NTP	GNTP (standard)	GNTP (attention)	CTP	MINERVA	NeuralLP	ComplEx
Countries	S1	AUC_PR	**100.00**	90.83	99.98	**100.00**	**100.00**	**100.00**	**100.0**	99.37
	S2	AUC_PR	89.67	87.40	90.82	**93.48**	91.81	92.36	75.1	87.95
	S3	AUC_PR	88.49	56.68	87.70	91.27	94.78	**95.10**	92.2	48.44

Table 7.4 The accuracy of the query answering prediction task of the *Countries* dataset using the *AUC_PR* evaluation metric. We present the average result over 5 runs for our systems. We quote the average results of *NTP*, *GNTP*, *CTP*, *MINERVA*, *NeuralLP* and *ComplEx*, where results of *NTP* and *GNTP* are reported in [Minervini et al., 2020a], the result of *CTP* is reported in [Minervini et al., 2020b] and results of *MINERVA*, *NeuralLP* and *ComplEx* are reported in [Minervini et al., 2020a].

As shown in Table 7.4, the *TNTP* accuracy of query answering predictions is comparable or better than that of most baselines, except for *GNTP (attention)*, *CTP* and *MINERVA* for tasks S2 and S3. *GNTP (attention)* and *CTP* have better accuracy of query answering predictions in *Countries* dataset, because their systems are more likely to perform well when there are fewer predicates. Both systems use attention vectors over embeddings of known predicates. When there are only two known predicates, learning the attention vector is easier. However, *GNTP (attention)* and *CTP* have scalability limitations, which are explained in detail after Table 7.5. Although *MINERVA* learns embeddings of predicates and constants, it does so not through template rules but through reinforcement learning and chaining relations. Since there are only two predicates, this method is more effective than inducing template rules. As shown later, this is not the case for datasets that include more predicates.

We have used for this dataset k_{max} equal to 10 instead of 1. This is due to the fact that the leaned embeddings of constants do not reflect well their semantic meaning and therefore the choice of a single proof branch might land on wrong substitutions. Also, since there is only 1 topic, which does not require 50 epochs of *FNTP* training, the rule induction part is trained for

[6]This is because the signature includes only two predicates and topic clustering is not needed.

100 epochs, the same number of epochs used by other systems.

In Table 7.5, we present the accuracy for the query answering prediction task of our three systems (*TNTP*, *TSNTP* and *NAF TSNTP*) using both the *MRR* and *HITS* metrics over the three datasets *Nations*, *Kinship* and *UMLS*. We compare these results against the baseline *NTP* [Rocktäschel and Riedel, 2017], as well as the extended systems *GNTP (standard)*, *GNTP (attention)* [Minervini et al., 2020a] and *CTP* [Minervini et al., 2020b]. Among these related approaches, the system most relevant to our systems is *GNTP (standard)*, which among the other baseline systems has been shown to achieve high accuracy whilst maintaining scalability.

Dataset	Metrics	TNTP (best)	TNTP (average)	TSNTP (best)	TSNTP (average)	NAF TSNTP (best)	NAF TSNTP (average)	NTP	GNTP (standard)	GNTP (attention)	CTP
Nations	MRR	**0.689**	0.684 ± 0.002	-	-	-	-	0.61	0.658	0.645	**0.709**
	HITS@1	**0.535**	0.528 ± 0.003	-	-	-	-	0.45	0.493	0.490	**0.562**
	HITS@3	**0.803**	0.792 ± 0.002	-	-	-	-	0.73	0.781	0.736	**0.813**
	HITS@10	**0.993**	0.991 ± 0.001	-	-	-	-	0.87	0.985	0.975	**0.995**
Kinship	MRR	0.746	0.740 ± 0.002	0.747	0.746 ± 0.001	**0.751**	0.749 ± 0.001	0.35	0.719	0.759	**0.764**
	HITS@1	0.632	0.619 ± 0.004	0.630	0.627 ± 0.002	**0.638**	0.631 ± 0.002	0.24	0.586	0.642	**0.646**
	HITS@3	0.840	0.837 ± 0.001	0.842	0.839 ± 0.001	**0.848**	0.844 ± 0.001	0.37	0.815	0.850	**0.859**
	HITS@10	0.953	0.943 ± 0.002	0.954	0.951 ± 0.001	**0.958**	0.953 ± 0.001	0.57	0.958	**0.959**	0.958
UMLS	MRR	0.839	0.837 ± 0.001	0.845	0.839 ± 0.002	**0.847**	0.841 ± 0.002	0.80	0.841	**0.857**	0.852
	HITS@1	0.737	0.730 ± 0.002	0.750	0.735 ± 0.004	**0.754**	0.738 ± 0.004	0.70	0.732	**0.761**	0.752
	HITS@3	0.938	0.933 ± 0.001	0.935	0.931 ± 0.001	**0.939**	0.933 ± 0.002	0.88	0.941	**0.947**	**0.947**
	HITS@10	**0.986**	0.982 ± 0.002	0.984	0.981 ± 0.001	0.984	0.981 ± 0.001	0.95	**0.986**	0.983	0.984

Table 7.5 The average and the best evaluation results for the query answering prediction task over five runs, of each of our systems on *Nations*, *Kinship*, *UMLS* dataset. The average performance is represented as the mean value plus its standard error. The dash means that the results are the same as that in *TNTP*. We highlight the best accuracy over all our systems and also the best accuracy over the baselines *NTP*, *GNTP(standard)*, *GNTP(Attention)* and *CTP*. The results of these baselines are quoted from other works as stated in [Minervini et al., 2020a] and [Minervini et al., 2020b].

Among our three systems, *TNTP* is the basic system. The more optimised *TSNTP* system yields better accuracy of query answering predictions. *TSNTP* possesses two properties that are somewhat opposing in their effects: although it selects fewer facts in each body atom unification than *TNTP*, which means it could miss more information, on the other hand the selected facts, based on topics and subdomains, may be more accurate than the facts selected by topics only, so the proof of each body atom is more targeted. The accuracy results of our experiments demonstrate that the second effect appears to overrule the first effect. We did not provide the accuracy of query answering predictions of *TSNTP* and *NAF TSNTP* on *Nations* dataset, because the *Nations* dataset has only 14 constants, making subdomains not very effective.

For the *Kinship* and *UMLS* datasets, *NAF TSNTP* improves *TSNTP* further, by inducing normal rules from *TSNTP* induced definite rules. Due to our normal rule induction algorithm, if a normal rule has more negative impact on the accuracy of query answering predictions than positive impact, the normal rule would not be used and its associated definite rule would be used instead. As a result, the accuracy of query answering predictions of *NAF TSNTP* is always higher than or equal to the accuracy of query answering predictions of *TSNTP*. For example, for the *Nations* dataset, *NAF TSNTP* does not find better normal rules than the definite rules induced by *TNTP* (or *TSNTP* using one subdomain), possibly due to the nature of the *KB*, so *NAF TSNTP* has the same accuracy of query answering predictions as that of *TNTP*.

Overall, among our three systems, *NAF TSNTP* also outperforms *GNTP(standard)*, the system most similar to our systems since it uses embeddings in the same way as our systems do, in nearly all cases.

Among all systems, *GNTP (attention)* and *CTP* have similar or slightly better performance than our systems. However, both systems suffer scalability problems. In *GNTP (attention)*, the embedding of an induced predicate is the weighted average of all embeddings of known predicates, which is controlled by an attention vector with the dimension size equivalent to the number of known predicates. If there are 1000 known predicates, each induced predicate needs to learn an attention vector with 1000 dimensions, whereas we keep the embedding size to 100. Consequently, *GNTP (attention)* can quickly become inefficient when there are over 100 known predicates, as stated in [Minervini et al., 2020a]. As for *CTP*, it is equivalent to the original *NTP*, except that it learns to select a subset of rules that are most relevant to the predicate of each specific query. Although it performs slightly better, it has scalablity issues as well, because it needs to consider all facts in each body atom unification. As a result, it has never been applied to *FB122* dataset.

Table 7.6 compares the accuracy for the query answering prediction task of our systems with respect to baselines other than those in the *NTP* family. In this table as these other baselines report one result only, we also report one result, namely the best accuracy from 5 runs for each of our systems.

Dataset	Metrics	TNTP	TSNTP	NAF TSNTP	MINERVA	RNNLogic	NeuralLP	ComplEx
Nations	MRR	**0.689**	-	-	-	-	-	0.60
	HITS@1	**0.535**	-	-	-	-	-	0.46
	HITS@3	**0.803**	-	-	-	-	-	0.67
	HITS@10	**0.993**	-	-	-	-	-	0.97
Kinship	MRR	0.746	0.747	**0.751**	0.720	0.722	0.619	0.46
	HITS@1	0.632	0.630	**0.638**	0.605	0.598	0.475	0.34
	HITS@3	0.840	0.842	**0.848**	0.812	0.814	0.707	0.49
	HITS@10	0.953	0.954	**0.958**	0.924	0.949	0.912	0.74
UMLS	MRR	0.839	0.845	**0.847**	0.825	0.842	0.778	0.58
	HITS@1	0.737	0.750	0.754	0.728	**0.772**	0.643	0.47
	HITS@3	0.938	0.935	**0.939**	0.900	0.891	0.869	0.63
	HITS@10	**0.986**	0.984	0.984	0.968	0.965	0.962	0.80

Table 7.6 The accuracy for the query answering prediction task of each system on *Nations*, *Kinship*, *UMLS* dataset compared to *MINERVA*, *RNNLogic*, *NeuralLP* and *ComplEx*. The result of *RNN* logic are reported in [Qu et al., 2020] and results of *MINERVA*, *NeuralLP* and *ComplEx* are reported in [Minervini et al., 2020a]. *TSNTP* has the same results as *TNTP* in the *Nations* dataset, because only one subdomain is required for its 14 constants (Thus, it is equivalent with *TNTP*). Also, in this dataset, no normal rules that could lead to better accuracy of query answering prediction compared with definite induced rules are induced, so the accuracy of *NAF TSNTP* is the same as that of *TNTP*. We did not find the evaluation results of *MINERVA*, *RNNLogic* and *NeuralLP* for this dataset, so we use dashes to represent that the results are not available.

As shown in Table 7.6, our systems have higher performance than *ComplEx*, *MINERVA*, *NeuralLP* and *RNNLogic* with respect to all metrics, except $HITS@1$ of the *UMLS* dataset, where *RNNLogic* outperforms our systems.

In Table 7.7, we present the accuracy for the query answering prediction task of our system *TSNTP* on *FB122* dataset [Guo et al., 2016]. *FB122* is the largest dataset used in this thesis, with over 11.6 billion potential facts in its signature. We evaluate just *TSNTP* as this is our most efficient system. The objectives of this evaluation are to demonstrate that *TSNTP* is scalable enough to learn rules over large *KBs*, and to test the level of accuracy that it can reach in solving query answering prediction tasks over large KBs.

Due to our limited computation resources we were not able to run multiple experiments to fine-tune our hyperparameters, as we did in the previous experiments where the *KBs* were much smaller. We only run our system once. Hence the results reported in Table 7.7 are just an indication of what *TSNTP* is capable of achieving. In this experiment, we used a fixed learning rate at 0.001, a fixed l_2 at 0.001, 5 topics and 20 subdomains, which were the best hyperparameter values found in the previous experiments. (Note that if more topics are used,

more template rules are required to support different topic combinations.) We used also a limited validation set to fine-tune the hyperparameters α and β. Their final values were 20 and 5 respectively. Also, note that differently from the *GNTP* experiments, our template rules do not contain transitive rules. This might have an impact on the overall accuracy and needs to be taken into account when comparing our results with that of GNTP.

Table 7.7 shows accuracy results of two classes of experiments, one where the *FB122 KB* and known background rules are given as input and the other where only the *FB122 KB* is given as input. Our systems belong to the second class. In each of these experiments, we consider two given test sets: *Test-I* where test queries are not provable by the given rules and *Test-II* where test queries are provable using the given rules. We compare the performance of our *TSNTP* against baseline systems that perform inference without the ability of inducing rules (i.e., *KALE*, *ASR-DistMult*, *ASR-ComplEx*, *DistMult* and *ComplEx*), and the *GNTP* system which instead is capable of learning rules.

		Test-I				Test-II			
		HITS@3	HITS@5	HITS@10	MRR	HITS@3	HITS@5	HITS@10	MRR
With Background Rules as input	KALE	**0.384**	**0.447**	**0.522**	0.325	0.797	0.841	0.896	0.684
	ASR-DistMult	0.363	0.403	0.449	0.330	0.980	0.990	0.992	0.948
	ASR-ComplEx	0.373	0.410	0.459	**0.338**	**0.992**	**0.993**	**0.994**	**0.984**
Without Background Rules as input	DistMult	0.360	0.403	0.453	0.313	**0.923**	**0.938**	**0.947**	0.874
	ComplEx	**0.370**	**0.413**	**0.462**	**0.329**	0.914	0.919	0.924	**0.887**
	GNTP	**0.337**	**0.369**	**0.412**	**0.313**	**0.982**	**0.990**	**0.993**	**0.977**
	TSNTP	0.218	0.262	0.337	0.206	0.884	0.933	0.980	0.789

Table 7.7 The accuracy for the query answering prediction task of *TSNTP* on *FB122* [Guo et al., 2016], compared with baseline systems *KALE* [Guo et al., 2016], *ASR* [Minervini et al., 2017], *DistMult* [Yang et al., 2015], *ComplEx* [Trouillon et al., 2016] and *GNTP (standard)* [Minervini et al., 2020a]. The results of these systems were summarised in [Minervini et al., 2020a]. Among these systems, the first three systems are given background rules as their inputs, whereas the bottom four systems do not use these rules. Among the four systems, *TSNTP* falls into the category that could induce interpretable first-order rules during training.

The *Test-I* experiment is mainly indicative of the completeness and generality of the induced rules. The performance of our systems, indeed that of all baseline systems, is very poor. We believe this is the case because the test queries in *Test-I* are mostly facts that are less likely to be proved by the limited number of rules that can be induced, considering also that the facts in the *KB* might not be fully representative of knowledge needed.

In our opinion, the most interesting experiment is that over *Test-II* when known background rules are not given as input and the training relies only on the facts included in the *FB122 KB*. In this case, we can truly evaluate the correctness of the induced rules, since 'target' learnable rules for the queries in *Test-II* are known but not used during training. We discuss some of the results in this experiment. As shown in Table 7.7, in the first class, the systems with background rules as input perform well, especially *ASR-ComplEx*, which takes the advantages of both given background rules and adversarial training. However, its success is based on the given rules, which might not be provided by other systems. In the second class, the *DistMult* and *ComplEx* systems perform reasonably well despite the fact that they are not capable of inducing rules. This might, at first sight, appear to be unexpected. But we believe that their good performance is due to the fact that learned embeddings of related constants are very close in vector space and can be unified interchangeably, allowing these systems to use learned embedding representations to answer queries correctly, without the need of learning rules. Because of the characteristic of the *FB122 KB* hypothesised for the relatively good results for *ComplEx* and *DistMult*, which do not induce rules, both the *GNTP* system and our *TSNTP* system are trained using specific regimes. *GNTP* learns just the embeddings of induced predicates and the selection strategy over induced rules during the first 95 epochs and then learns and fine-tunes the embeddings of known predicates, constants and unknown predicates only in the last 5 epochs of training. The purpose is, in our opinion, that in this way during the 95 epochs, a very specific 'abstract' space of potential rules is trained, which is then fine-tuned (or instantiated) over the known predicates and constants while their embeddings are learned during the last 5 epochs. In our *TSNTP* we use a different training regime. We freeze the (randomly initialised) embeddings of constants for the first 30 epochs, in order to 'force' the system to learn the embeddings of both known and induced predicates together. We then allow the constant embeddings to be updated during the remaining 20 epochs. We believe that this difference in the training regime is why *GNTP* slightly outperforms our *TSNTP* system. The computational task that *GNTP* solves when learning unknown predicates only is 'simpler' than the one computed by *TSNTP*, which has to learn embeddings of both known and unknown induced predicates at the same time. Nevertheless, it is interesting to note that the trained *TSNTP* proves all test queries in *TestII* using only its induced rules, some of which are the same as the given background rules known to be useful for proving the test queries.

7.3.2 Runtime and Space Evaluation

We now evaluate the time and memory efficiency of our systems and compare with those of *GNTP (standard)*. We track the CPU computation time and maximum memory used for each system, using the hyperparameters that lead to the accuracy results reported in Table 7.5. Specifically, we report the improvement of our systems with respect to *NTP* as well as that of *GNTP* with respect to *NTP*. We do so by re-running *NTP* and *GNTP (standard)* using the configuration ranges specified in [Minervini et al., 2020a]. The improvement of time and memory efficiency are presented in Table 7.8. Since the authors in [Minervini et al., 2020a] did not specify the configuration that lead to the best accuracy, we have tested the time and space improvements for all the given ranges of values with respect to number of rules and number of facts selected in the computational tree for each query. Therefore, the table includes for *GNTP (standard)* the range of time and space improvement with respect to *NTP*. All systems are run using the same hardware, namely 8 CPU with 96 GB memory, allocated by Imperial Cloud. The computational time is measured in terms of CPU time.

Time	Nations	Kinship	UMLS
TNTP	21.2	20.9	23.9
TSNTP	–	47.0	44.1
NAF TSNTP	–	43.3	35.0
GNTP (standard)	10.4 - 25.8	15.15 - 60.8	15.25 - 52.8

Space	Nations	Kinship	UMLS
TNTP	2.4	5.4	5.0
TSNTP	–	5.4	5.2
NAF TSNTP	–	2.9	1.6
GNTP (standard)	4.8 - 9.3	7.3 - 13.0	7.5 - 18.1

Table 7.8 The improvement of time and memory efficiency of our systems with respect to *NTP*, when using 5 topics and 10 subdomains (except 1 subdomain for *Nations* dataset). The number indicates how many times more efficient a system is, compared with *NTP*. The higher the number, the better. We compare our systems with *GNTP (standard)* and report their improvement as a range using their given configurations. The time and memory improvement of our systems are the average value over five runs and the ranges of improvement of *GNTP (standard)* are based on a single run for each value in the range.

The computation time of *TNTP* and *TSNTP* includes 50 epochs of *FNTP* training and 50 epochs of rule induction. The time also includes topic and/or subdomain generation, but these take less than 0.01 second, which is negligible compared with the training time which is measured by hours. Note that, on average across the different datasets, our systems use more facts and rules in the unification process than the maximum configuration considered by *GNTP (standard)*. The computation time of *NAF TSNTP* includes *TSNTP* computation time plus the normal rule induction time. As shown in Table 7.8, our systems show time speed up from about 20.9 to 47 times faster than *NTP*, whereas the time speed up ranges of *GNTP* indicate that for some configurations *GNTP* performs better than our systems but for others it performs

worse. Similarly, our systems show memory efficiency improvement up from 1.6 to 5.4 times more memory efficient than *NTP*, which is slightly worse than *GNTP*. Note that the memory improvement of *NAF TSNTP* is worse than *TSNTP* due to extra memory used to construct multiple independent computational trees for selecting the rules to revise. Note also that when more topics and subdomains are used the speedup and memory utilisation improve further, but at a cost of slightly lower accuracy. We report in Table 7.11 experiments that demonstrate this.

7.3.3 Rule Induction

Our systems can induce rules that generate high accuracy of query answering predictions comparable with other state-of-the-art systems, while also being human-interpretable. To illustrate this we present here the five most frequently used induced rules and their decoding from *Nations* dataset, using our decoding algorithm described in Section 3.5.

1st important rule, used by 11.1% test queries

0.465759: intergovorgs3(X, Y) :− intergovorgs(X, Y).

2nd important rule, used by 8.8% test queries

0.42706525: relintergovorgs (X, Y) :− relngo(X, Y).

3rd important rule, used by 6.8% test queries

0.76974094: emigrants3(X, Y) :− tourism3(X, Y).

0.7543415: ngo(X, Y) :− tourism3(X, Y).

0.71589917: reltourism(X, Y) :− tourism3(X, Y).

4th important rule, used by 6.8% test queries

0.33827525: severdiplomatic(X, Y) :− treaties (Y, X).

5th important rule, used by 6.8% test queries

0.89173555: negativebehavior(X, Y) :− negativecomm(X, Y).

0.80859405: accusation(X, Y) :− negativecomm(X, Y).

0.89173555: negativebehavior(X, Y) :− warning(X, Y).

0.80859405: accusation(X, Y) :− warning(X, Y).

0.6028473: negativecomm(X, Y) :− warning(X, Y).

0.65975684: negativebehavior(X, Y) :− expeldiplomats(X, Y).

0.65975684: accusation(X, Y) :− expeldiplomats(X, Y).

0.6028473: negativecomm(X, Y) :− expeldiplomats(X, Y).

As shown by the above examples, the decoded induced rules reflect human commonsense. For example, the last rule (the 5th important rule) describes the negative relationships between two

entities. Note also that because of the learned embedding representations, this induced rule yields to many decoded rules with similar meaning, thus demonstrating how compact induced rules are in capturing information.

In *Nations* dataset, all queries in the test set are proved by induced rules and the accuracy of query answering predictions of *TNTP* is very high compared with other systems. This illustrates that our induced rules capture all relations in the knowledge base. These rules can be used to aid knowledge base completion when some facts are missing. In *UMLS* and *Kinship* dataset, around 30%-40% test queries are proved by induced rules and other test queries are proved by facts. This lower percentage of rule involvement is acceptable for the following reason. Given a specific query, there may be facts in the *KB* that are similar to the query. In this case, since proving with a similar fact will generally give a higher score than proving with a rule, and the selected branch is the one with the highest score, the query would be proved by the fact. As a result, the percentage of rule involvement is also affected by the nature of the knowledge base.

7.3.4 Effects of NAF

As presented in Table 7.5, inducing normal rules using *NAF TSNTP* improves the accuracy of query answering predictions compared to *TSNTP*. In particular, the *MRR* and *HITS@1* is higher, even though in every dataset we only convert 10 definite induced rules ($Z = 10$) to normal rules, over the 60 induced rules. Appending a negated atom can improve the accuracy of query answering predictions, because the negated atom may reduce the proof scores of some false positive predictions. In the evaluation over the test set, fewer numbers of false positive queries improve the rankings of the true positive queries. In *UMLS* and *Kinship* dataset, 9 and 10 qualified normal rules (respectively) are induced from the 10 most frequently used definite rules for proving positive training queries. With these induced normal rules, in *UMLS*, about 1.5% of corrupted queries in the test set receive lower proof scores compared to those generated by the associated definite rules. For *Kinship*, it is about 5% of corrupted queries that result to have lower scores.

7.4 Effects of Hyperparameters

We present here some experiments to demonstrate the effects of some hyperparameters, including α and β amplification hyperparameters, k_{max} values, topic and subdomain numbers.

7.4.1 Effects of Alpha and Beta Amplification Hyperparameters

As presented in Section 3.4, we introduced two scaling factors, α and β, to encourage more rules to be used during training. In this section, we present empirical results that demonstrate how these two hyperparameters indeed lead to better rules being learned.

Table 7.9 presents the effects on the accuracy of query answering predictions and the use of induced rules for different α and β. The four experiments are run under the same setting, except for these two scaling factors.

	α	β	MRR| @1 | @3 | @10	Rule Involvement (%)
Experiment 1	1	1	0.744|0.622|0.844|0.951	2
Experiment 2	500	1	0.669|0.564|0.744|0.843	41
Experiment 3	1	3	0.751|0.632|0.846|0.955	2
Experiment 4	500	3	0.754|0.645|0.834|0.949	34

Table 7.9 The effects on the accuracy of query answering predictions and the use of induced rules for different α and β, using *TSNTP* on *Kinship* dataset. The rule involvement indicates the percentage of test queries that are proved using induced rules. $\alpha = 500$ and $\beta = 3$ are the optimal values we found and subsequently used in evaluating $TSNTP$ on *Kinship* dataset.

This table illustrates the importance of the two hyperparameters. Without the two hyperparameters (i.e. $\alpha/\beta = 1$ implies that they are not used), although the MRR and $HITS$ results are good, only 2% of test queries are proved using the induced rules, indicating that they are of limited use. In fact, in such case the system is like a *FNTP*, with much more training time for little effect on rule induction. Experiment 2 and 3 use either α or β, but they either have low accuracy of query answering predictions (Experiment 2) or have low rule involvement (Experiment 3). Experiment 4 uses both hyperparameters and it has the highest accuracy and also high rule involvement. These experiments validate that these hyperparameters can encourage induction of useful rules.

7.4.2 Effects of k_{max}

	MRR| @1 | @3 | @10	Rule Involvement
$\mathrm{k}_{max} = 1$	0.741|0.615|0.841|0.960	0.43
$\mathrm{k}_{max} = 3$	0.740|0.618|0.836|0.948	0.33
$\mathrm{k}_{max} = 5$	0.727|0.603|0.828|0.941	0.36

Table 7.10 The effects of k_{max} on *TSNTP* using the *Kinship* dataset, in terms of accuracy of query answering predictions and rule involvement.

Table 7.10 presents the impact on *MRR*, *HITS* and rule involvement of different k_{max} values. The three experiments are run under the same setting, except for the k_{max} value. Surprisingly, when k_{max} is higher, the accuracy of query answering predictions decreases, whereas it might have been expected to increase. When k_{max} is low, although there are fewer proof branches, each branch gets updated more frequently, so leading to better rule induction. A higher k_{max} means that more branches are expanded. The k_{max} value does not affect template rules with one body literal. However, for template rules with two body literals, such as $\#H(X,Y)$:- $\#T1(X,Z), \#T2(Z,Y)$, the second body literal $\#T2(Z,Y)$ needs to be proved k_{max} times following the k_{max} best proved branches of $\#T1(X,Z)$, each time with a fresh binding for the free variable Z. In theory, the k_{max} expansions cover the search space better, compared to $k_{max} = 1$, which prove $\#T2(Z,Y)$ with only one binding of Z. However, when $k_{max} = 1$, although Z only binds with one constant, because of soft-unification, such binding actually is representative of bindings with many similar constants. As shown by the results, a higher k_{max} does not necessarily lead to higher accuracy of query answering predictions. The best accuracy of query answering predictions is when $k_{max} = 1$.

7.4.3 Effects of Topics and Subdomains

We present here the effects that different numbers of topics and subdomains have on accuracy of query answering predictions, computational time and use of induced rules. We consider only the *TSNTP* system.

Topic Num	Subdomain Num	MRR| @1 | @3 | @10	Time Speedup	Rule Involvement (%)
5	10	0.845|0.750|0.935|0.977	1.00	35
5	20	0.835|0.731|0.923|0.977	1.20	40
5	30	0.782|0.675|0.873|0.939	1.22	41
10	10	0.834|0.730|0.927|0.975	1.31	45
10	20	0.822|0.722|0.911|0.969	1.43	35
10	30	0.785|0.685|0.865|0.919	2.12	37
15	10	0.831|0.725|0.929|0.976	1.67	43
15	20	0.834|0.726|0.935|0.978	1.79	48
15	30	0.799|0.694|0.887|0.952	2.14	53

Table 7.11 The effects of topics and subdomains on accuracy of query answering predictions, time and rule involvement, for *TSNTP* and the *UMLS* dataset.

The results in Table 7.11 show that the differences in MRR and $HITS$ results are not significant when using settings with different numbers of topics and subdomains. All experiments achieve high accuracy of query answering predictions, except for the cases when the subdomain number is 30. This maybe because when the subdomain number is high, constants are more distributed,

possibly forcing similar facts to belong to different subsets of facts, F_{t_s}.

The speedup of a setting is calculated as the ratio of the computation time of *TSNTP* with 5 topics and 10 subdomains to the computation time of the setting. Thus the speedup is 1.0 in the first row and for 10 topics and 30 subdomains the computation of the test set is just over twice as fast as for 5 topics and 10 subdomains. The results show that the more topics and subdomains, the faster the computation, but the lower the accuracy.

Summary. All our systems, especially *NAF TSNTP*, are able to induce interpretable (normal) rules that can be used to answer queries with high accuracy. We achieve the highest accuracy of query answering predictions in most cases, comparing to *GNTP (standard)* and the other baselines. Furthermore, our *NAF TSNTP* is the only system that can induce normal rules.

Chapter 8

Conclusion

We developed three neural-symbolic systems that learn rules and embeddings at the same time, namely *TNTP*, *TSNTP* and *NAF TSNTP*. They can induce first-order (normal) rules efficiently and answer queries from *KBs* with good accuracy. Our systems take the advantages of both neural network and symbolic reasoning. Each system represents a symbolic knowledge base using embeddings that are trained to capture semantic meaning, so that soft-unification can be used to unify similar symbols, unlike the hard-unification used in symbolic logic where a symbol can only unify with an exact match. Our systems use a computational tree to prove queries in a backward chaining approach which models the proof method used in logic programming. This computational tree architecture not only enables our systems to answer queries, but also supports logic reasoning, in particular rule induction where rules are induced by learning embedding representations. To control the size of the computational tree, our systems are able to use topics and subdomains, generated by unsupervised clustering algorithms. They identify a subset of most relevant facts during the proof, thus improving the computation efficiency. In addition to definite rules, our *NAF TSNTP* also induces normal rules, as a first step towards learning more expressive rules, where negated atoms can be added to a definite rule to make the rule more specific. These induced rules can be decoded to first-order rules, which are human-interpretable and can be used to explain the knowledge base. Using these induced rules and trained embeddings, our systems are shown to achieve high accuracy of query answering predictions.

With the normal rule induction ability and the embedding-based architectures, our systems could be extended in many directions to tackle challenges in neural-symbolic reasoning. For example, our systems can be extended to support background rules and using these background rules to deduce more facts, forming an integrated system that perform deduction and induction at the

same time. Our systems can also be used to tackle the commonsense reasoning challenges, where our systems could be used to identify the missing information that are ignored as 'commonsense' via rule induction and these induced rules can be viewed as 'explanations' of the knowledge base.

In the following paragraphs, we firstly summarise the key features of the three systems, from the perspectives of rule induction, query answering and computational efficiency. Then, we present the potential directions of our systems.

Rule Induction. All our systems induce first-order (normal) rules that aim to cover as many positive examples and as few negative examples as possible. These rules enable complex relations, such as transitive relations, to be captured to answer relevant queries and explain how a proof score is generated for a query. In our systems, each induced rule is trained through gradient descent by learning the embedding representations of the predicates that appear in the copies of the given topical template rules (TTR). This is done through backward chaining and soft-unification with the known facts given in the KB. Using the specification of topics in the $TTRs$ enables more targeted soft-unification of the body conditions. The topic specification also encourages the induction of rules over a larger search space within the signature of the KB. After training, embeddings capture the semantic meaning of these symbols, so that embeddings with similar meanings are close in vector space and their corresponding symbols can be used interchangeably. Thus, each induced rule can be interpreted by decoding the predicate embeddings to their nearest known predicates in vector space. To amplify the chances of $TTRs$ being used during the backward chaining, all three systems use two hyperparameters.

Out of the three systems, the *NAF TSNTP* is the one capable of learning more expressive rules. Indeed, this is the first ever purely differentiable system able to learn rules with negation-as-failure in the body to solve query answering tasks. As demonstrated in the evaluations, the increased expressivity of negation-as-failure leads to increased accuracy of the query answering prediction task.

Query Answering. All three systems are able to answer symbolic queries from given KBs. A query is answered by constructing a computational tree and computing a proof score through backward chaining and soft-unification. A high score indicates that the query is provable given the current information. The system is trained so that positive queries are predicted with high scores and negative queries (i.e. corrupted facts) are predicted with low scores. The query

answering prediction task is then evaluated using the *MRR* and *HITS* metrics. Note that this evaluation does not care about the absolute value of the score, but only its relative rank among the scores of the corruptions. High *MRR* and *HITS* scores indicate that the system is able to distinguish positive queries from their corruptions, where positive queries consistently have higher scores than the scores of their corruptions. We compare the accuracy of solving query answering tasks against that of existing related state-of-the-art sub-symbolic systems. As shown in Chapter 7, our approaches are indeed in most cases more accurate. Among the three systems, *NAF TSNTP* performs the best, by using both definite and normal rules to answer queries.

Computation Efficiency. Our systems are able to induce rules efficiently. To improve the computational efficiency, the three systems make use of the notion of topical template rules (*TTRs*). Each copy of a *TTR* is given specified topics, which helps to prevent multiple copies of a *TTR* inducing similar rules, by forcing specified topics. Using topics, *TNTP* only unifies each body atom with a selected subset of facts, instead of all facts in the *KB*, so reducing the computation time and space. This computational efficiency is improved even further in *TSNTP* where constants are grouped into subdomains, constraining even further the soft-unification process. This improved computational efficiency does not, however, cause a reduction in accuracy, as both *TNTP* and *TSNTP* report similar performance on the various datasets. On the other hand, *NAF TSNTP* also maintains the features of *TSNTP*, allowing a negated atom to be proved by just checking that its positive part cannot be proved within the context of its specific topic and subdomain. Furthermore, *NAF TSNTP* achieves better accuracy by the virtue of its ability to learn normal rules. Our evaluation in Chapter 7 shows that our systems have significant improvement in speed and memory efficiency compared to *NTP*, and comparable computational efficiency with *GNTP*.

8.1 Future Works

There are many potential directions in which this work could be taken further. In what follows, we discuss in a broad sense three directions that look at how our systems can be used for commonsense reasoning, explainableAI and interacting with other symbolic/neural/neural-symbolic systems. Then, we discuss further extensions of our systems, including incorporating background rules in the training process; improving the computational performance even further by paralleling the training process, and exploring the possibility of inducing rules only from part of a large *KB* and then fine-tuning them over the full *KB*.

- Commonsense Reasoning

  Our system could be extended to support commonsense reasoning [Davis, 2017]. A knowledge base might not capture some basic information that human considers as commonsense knowledge and would not state explicitly, such as 'sister' is a subset of 'siblings'. The missing of such important yet implicit information makes logic reasoning harder. Our systems can identify these implicit information by inducing rules, where given an incomplete knowledge base, our systems learn rules that capture relationships in the knowledge base, so that queries can be answered correctly. Our systems, unlike humans, have no bias on these explicit or implicit relations: all predicates are treated in the same way by learning their embedding representations during the training process and predicates that correlate are likely to be induced as a rule. The induced rules can be added to the knowledge base as background rules for further deductions and logic reasoning. In this way, the implicit information becomes interpretable 'explicit rules'. A next step would be to explore how to incorporate our systems in commonsense reasoning systems.

- ExplainableAI

  Our systems are able to induce first-order (normal) rules for human-interpretation, so they could be applied to solve the explainability challenges in the neural network field, as addressed in *Explainable AI* [Cyras et al., 2021]. Although some neural network systems are able to achieve good prediction results, its prediction mechanism remains as a blackbox with many matrices that are hard for human to interpret. Our systems can be used as surrogate models. They can use other systems' prediction results as prediction labels and induce some rules using their queries and labels. These induced rules can provide some insights on how neural networks make decisions. The challenge here is that our systems can only induce rules from knowledge bases, so the training set needs to be converted to a knowledge base first. So future works could be needed to explore how to support other forms of training data, apart from knowledge bases.

- Further Use of Induced Rules and Embeddings

  Our systems support both symbolic representations and neural representations, which enable our systems to interact with both neural systems and symbolic systems. Our induced rules, in symbolic representation, can be used by other logic reasoning systems for further reasoning. The trained embeddings can be used by other neural systems, as they capture semantic meanings that are trained via backward chaining.

- Use of Background Rules

  Background information can provide many useful information to a learning system, for example background rules can be used for answering queries through backward chaining, whereas without these rules queries might not be answered correctly. The injection of background rules can improve the efficiency of a learning system, which can focus on learning other unknown relations and use the given rules directly. In this thesis, our proposed systems learn rules from only facts in the KB and $TTRs$. However, the computation process is general enough to support also the inference over given background rules. These could be processed in the same way as induced rules, except that the predicate embeddings in the background rules would be 'known'. Different background rules involving the same predicates would then share predicate embeddings. This differs from induced predicates, where embeddings are independent from each other across different TTR copies. In this way, the reasoning process over the computational tree would help the training of embeddings to converge faster and hopefully improve the accuracy even further. This of course may come with additional computational challenges in the construction of computational trees, e.g. they would be much larger. These could be overcome by exploring how existing proof methods developed for pure symbolic inference could be adapted and applied to the context of pure differentiable settings.

- Parallelisation

  Our systems could be optimised further by splitting the training of induction rules into multiple tasks, each with different combinations of $TTRs$. Each task would learn embeddings independently. As typical of a parallelisation process, the challenge is on how to 'recombine' the outcome of each computational thread. In our case, we would need to develop mechanisms for merging the embeddings of the symbols in the signature of the KB, which have been trained in each independent task. One possibility is to select the $TTRs$ that are most frequently used in each parallel task and retrain the system using these selected $TTRs$. In this way, each parallel task could use fewer epochs in training, so helping to improve the overall performance.

- Induction from a Subset of a KB

  Our systems currently use the whole KB to induce rules. However, for a very large KB it would be ideal if a subset of such a KB could be used for learning induced rules. For example, given a kinship KB, we would only need a fraction of the KB to learn the kinship relations, which would be applicable to the whole KB. The challenge would be to identify

the 'most relevant' subset of the *KB*. In a pure symbolic setting, concepts such as strongly connected components of a *KB* have often be used to improve the inference process. Some heuristics similar to this could be developed and applied to the differentiable setting and evaluated against pure sampling methods.

The above extensions would help develop effective and scalable differentiable inference systems underpinned by principles informed from the well established symbolic reasoning field.

References

Martín Abadi, Paul Barham, Jianmin Chen, Z. Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek Gordon Murray, Benoit Steiner, Paul A. Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zhang. Tensorflow: A system for large-scale machine learning. *arXiv*, abs/1605.08695, 2016. URL https://arxiv.org/abs/1605.08695.

Amina Adadi and Mohammed Berrada. Peeking inside the black-box: A survey on explainable artificial intelligence (xai). *IEEE Access*, 6:52138–52160, 2018. URL https://doi.org/10.1109/ACCESS.2018.2870052.

Srinivasan Ashwin. Aleph manual [third print.]. 2007. URL http://www.comlab.ox.ac.uk/activities/machinelearning/Aleph/aleph.html.

Yaniv Aspis, Krysia Broda, Jorge Lobo, and Alessandra Russo. Embed2sym - scalable neuro-symbolic reasoning via clustered embeddings. *Proceedings of the Nineteenth International Conference on Principles of Knowledge Representation and Reasoning*, 2022. URL https://proceedings.kr.org/2022/44/.

Samy Badreddine, Artur d'Avila Garcez, Luciano Serafini, and Michael Spranger. Logic tensor networks. *Artificial Intelligence*, 303:103649, 2022. URL https://doi.org/10.1016/j.artint.2021.103649.

Tarek R. Besold, Artur S. d'Avila Garcez, Sebastian Bader, Howard Bowman, Pedro M. Domingos, Pascal Hitzler, Kai-Uwe Kühnberger, Luís C. Lamb, Daniel Lowd, Priscila Machado Vieira Lima, Leo de Penning, Gadi Pinkas, Hoifung Poon, and Gerson Zaverucha. Neural-symbolic learning and reasoning: A survey and interpretation. *abs/1711.03902*, 2017. URL http://arxiv.org/abs/1711.03902.

Kurt Bollacker, Colin Evans, Praveen Paritosh, Tim Sturge, and Jamie Taylor. Freebase: A collaboratively created graph database for structuring human knowledge. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, pages 1247–1250, 2008. URL http://doi.acm.org/10.1145/1376616.1376746.

Antoine Bordes, Jason Weston, Ronan Collobert, and Yoshua Bengio. Learning structured embeddings of knowledge bases. In *Proceedings of the Twenty-Fifth AAAI Conference on Artificial Intelligence*, AAAI'11, pages 301–306, 2011. URL http://dl.acm.org/citation.cfm?id=2900423.2900470.

Antoine Bordes, Nicolas Usunier, Alberto Garcia-Duran, Jason Weston, and Oksana Yakhnenko. Translating embeddings for modeling multi-relational data. In *Advances in Neural Information Processing Systems*, volume 26, 2013. URL https://proceedings.neurips.cc/paper/2013/file/1cecc7a77928ca8133fa24680a88d2f9-Paper.pdf.

Guillaume Bouchard, Sameer Singh, and Théo Trouillon. On approximate reasoning capabilities of low-rank vector spaces. In *2015 AAAI Spring Symposia, March 22-25, 2015*, 2015. URL http://www.aaai.org/ocs/index.php/SSS/SSS15/paper/view/10257.

Leonid Boytsov and Bilegsaikhan Naidan. Engineering efficient and effective non-metric space library. In *Proceedings of the 6th International Conference on Similarity Search and Applications - Volume 8199*, SISAP 2013, pages 280–293, 2013. URL https://doi.org/10.1007/978-3-642-41062-8_28.

Keith L. Clark. Negation as failure. In *Logic and Data Bases*, pages 293–322, 1978. URL https://doi.org/10.1007/978-1-4684-3384-5_11.

William W. Cohen, Fan Yang, and Kathryn Mazaitis. Tensorlog: Deep learning meets probabilistic dbs. *arXiv: abs/1707.05390*, 2017. URL http://arxiv.org/abs/1707.05390.

Kristijonas Cyras, Antonio Rago, Emanuele Albini, Pietro Baroni, and Francesca Toni. Argumentative XAI: A survey. In *Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence, IJCAI 2021, Virtual Event, 19-27 August 2021*, pages 4392–4399, 2021. URL https://doi.org/10.24963/ijcai.2021/600.

Wang-Zhou Dai and Stephen H. Muggleton. Abductive knowledge induction from raw data. In *Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence, IJCAI 2021, Virtual Event / Montreal, Canada, 19-27 August 2021*, pages 1845–1851, 2021. URL https://doi.org/10.24963/ijcai.2021/254.

Rajarshi Das, Shehzaad Dhuliawala, Manzil Zaheer, Luke Vilnis, Ishan Durugkar, Akshay Krishnamurthy, Alex Smola, and Andrew McCallum. Go for a walk and arrive at the answer: Reasoning over paths in knowledge bases using reinforcement learning. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*, 2018. URL https://openreview.net/forum?id=Syg-YfWCW.

Artur S. d'Avila Garcez, Krysia Broda, and Dov M. Gabbay. *Neural-symbolic learning systems - foundations and applications*. Perspectives in neural computing. 2002. ISBN 978-1-85233-512-0. URL https://doi.org/10.1007/978-1-4471-0211-3.

Artur S. d'Avila Garcez, L. Lamb, and Dov M. Gabbay. *Neural-Symbolic Cognitive Reasoning*. 01 2009. ISBN 978-3-540-73245-7. URL https://doi.org/10.1007/978-3-540-73246-4.

Artur S. d'Avila Garcez, Marco Gori, L. Lamb, Luciano Serafini, Michael Spranger, and S. Tran. Neural-symbolic computing: An effective methodology for principled integration of machine learning and reasoning. *arXiv*, abs/1905.06088, 2019. URL https://arxiv.org/abs/1905.06088.

Ernest Davis. Logical formalizations of commonsense reasoning: A survey. *J. Artif. Intell. Res.*, 59:651–723, 2017. URL https://doi.org/10.1613/jair.5339.

Jesse Davis and Mark Goadrich. The relationship between precision-recall and roc curves. In *Proceedings of the 23rd International Conference on Machine Learning*, ICML '06, page 233–240, 2006. URL https://doi.org/10.1145/1143844.1143874.

Woodrow W. Denham and Douglas R. White. Multiple measures of alyawarra kinship. *Field Methods*, 17(1):70–101, 2005. URL https://doi.org/10.1177/1525822X04271610.

Tim Dettmers, Pasquale Minervini, Pontus Stenetorp, and Sebastian Riedel. Convolutional 2d knowledge graph embeddings. In *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, (AAAI-18)*, pages 1811–1818, 2018. URL https://www.aaai.org/ocs/index.php/AAAI/AAAI18/paper/view/17366.

Ivan Donadello, Luciano Serafini, and Artur S. d'Avila Garcez. Logic tensor networks for semantic image interpretation. In *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI 2017, August 19-25, 2017*, pages 1596–1602, 2017. URL https://doi.org/10.24963/ijcai.2017/221.

Richard Evans and Edward Grefenstette. Learning explanatory rules from noisy data. *J. Artif. Int. Res.*, 61(1):1–64, January 2018. URL http://dl.acm.org/citation.cfm?id=3241691.3241692.

João Ferreira, Manuel de Sousa Ribeiro, Ricardo Gonçalves, and João Leite. Looking inside the black-box: Logic-based explanations for neural networks. In *Proceedings of the 19th International Conference on Principles of Knowledge Representation and Reasoning, KR 2022, Haifa, Israel. July 31 - August 5, 2022*, 2022. URL https://proceedings.kr.org/2022/45/.

Dov M Gabbay, C J Hogger, and J A Robinson. *Handbook of Logic in Artificial Intelligence and Logic Programming, Volume2, Deduction Methodologies*. 1994. ISBN 0-19-853746-8.

Mohamed H. Gad-Elrab, Daria Stepanova, Jacopo Urbani, and Gerhard Weikum. Exception-enriched rule learning from knowledge graphs. In *The Semantic Web – ISWC 2016: 15th International Semantic Web Conference, Kobe, Japan, October 17–21, 2016, Proceedings, Part I*, page 234–251, 2016. URL https://doi.org/10.1007/978-3-319-46523-4_15.

Luis Galárraga, Christina Teflioudi, Katja Hose, and Fabian M. Suchanek. Fast rule mining in ontological knowledge bases with amie+. *The VLDB Journal*, 24(6):707–730, dec 2015. URL https://doi.org/10.1007/s00778-015-0394-1.

Jorma Rissanen George Cybenko, Dianne P. O'Leary. *The Mathematics of Information Coding, Extraction and Distribution*. ISBN 9780387986654.

Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. *Journal of Machine Learning Research - Proceedings Track*, 9:249–256, 01 2010. URL http://proceedings.mlr.press/v9/glorot10a.html.

John C. Gower and Gavin J. S. Ross. Minimum spanning trees and single linkage cluster analysis. *Journal of The Royal Statistical Society Series C-applied Statistics*, 18:54–64, 1969. URL https://sites.cs.ucsb.edu/~veronika/MAE/mstSingleLinkage_GowerRoss_1969.pdf.

Shu Guo, Quan Wang, Lihong Wang, Bin Wang, and Li Guo. Jointly embedding knowledge graphs and logical rules. In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*, pages 192–202, November 2016. URL https://aclanthology.org/D16-1019.

Madan M. Gupta and J. Qi. Theory of t -norms and fuzzy inference methods. *Fuzzy Sets and Systems*, 40:431–450, 1991.

Petr Hájek. Metamathematics of fuzzy logic. In *Trends in Logic*, 1998. ISBN 978-0-7923-5238-9.

Barbara Hammer and Pascal Hitzler. *Perspectives of Neural-Symbolic Integration*, volume 77. 01 2007. ISBN 978-3-540-73953-1. doi: 10.1007/978-3-540-73954-8.

Alexander Hinneburg and Daniel A. Keim. Optimal grid-clustering: Towards breaking the curse of dimensionality in high-dimensional clustering. In *Proceedings of the 25th International Conference on Very Large Data Bases*, VLDB '99, page 506–517, 1999.

Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Comput.*, 9(8): 1735–1780, November 1997. URL https://doi.org/10.1162/neco.1997.9.8.1735.

Jeff Johnson, Matthijs Douze, and Hervé Jégou. Billion-scale similarity search with gpus. *arXiv*, abs/1702.08734, 2017. URL http://arxiv.org/abs/1702.08734.

Ian Jolliffe. *Principal Component Analysis and Factor Analysis*, pages 115–128. 01 1986. ISBN 978-1-4757-1906-2. doi: 10.1007/978-1-4757-1904-8_7.

John M. Jumper, Richard Evans, Alexander Pritzel, Tim Green, Michael Figurnov, Olaf Ronneberger, Kathryn Tunyasuvunakool, Russ Bates, Augustin Zídek, Anna Potapenko, Alex Bridgland, Clemens Meyer, Simon A A Kohl, Andy Ballard, Andrew Cowie, Bernardino Romera-Paredes, Stanislav Nikolov, Rishub Jain, Jonas Adler, Trevor Back, Stig Petersen, David A. Reiman, Ellen Clancy, Michal Zielinski, Martin Steinegger, Michalina Pacholska, Tamas Berghammer, Sebastian Bodenstein, David Silver, Oriol Vinyals, Andrew W. Senior, Koray Kavukcuoglu, Pushmeet Kohli, and Demis Hassabis. Highly accurate protein structure prediction with alphafold. *Nature*, 596:583 – 589, 2021.

Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015. URL http://arxiv.org/abs/1412.6980.

Mark Law, Alessandra Russo, and Krysia Broda. Inductive learning of answer set programs. In *Proceedings of the 14th European Conference on Logics in Artificial Intelligence - Volume 8761*, pages 311–325, 2014. URL http://dx.doi.org/10.1007/978-3-319-11558-0_22.

Mark Law, Alessandra Russo, Elisa Bertino, Krysia Broda, and Jorge Lobo. Fastlas: Scalable inductive logic programming incorporating domain-specific optimisation criteria. In *The Thirty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2020*, pages 2877–2885, 2020. URL https://aaai.org/ojs/index.php/AAAI/article/view/5678.

James B. Macqueen. Some methods for classification and analysis of multivariate observations. In *In 5-th Berkeley Symposium on Mathematical Statistics and Probability*, pages 281–297, 1967.

Yu Malkov, Alexander Ponomarenko, Andrey Logvinov, and Vladimir Krylov. Approximate nearest neighbor algorithm based on navigable small world graphs. *Information Systems*, 45: 61–68, 01 2013. URL https://doi.org/10.1016/j.is.2013.10.006.

Robin Manhaeve, Sebastijan Dumancic, Angelika Kimmig, Thomas Demeester, and Luc De Raedt. Deepproblog: Neural probabilistic logic programming. *arXiv*, abs/1805.10872, 2018. URL http://arxiv.org/abs/1805.10872.

Alexa McCray. An upper-level ontology for the biomedical domain. *Comparative and functional genomics*, 4:80–4, 01 2003. URL https://doi.org/10.1002/cfg.255.

Christian Meilicke, Melisachew Chekol, Daniel Ruffinelli, and Heiner Stuckenschmidt. Anytime bottom-up rule learning for knowledge graph completion. pages 3137–3143, 08 2019. URL https://doi.org/10.24963/ijcai.2019/435.

Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg Corrado, and Jeffrey Dean. Distributed representations of words and phrases and their compositionality. In *Proceedings of the 26th International Conference on Neural Information Processing Systems - Volume 2*, NIPS'13, pages 3111–3119, 2013. URL http://dl.acm.org/citation.cfm?id=2999792.2999959.

Pasquale Minervini, Thomas Demeester, Tim Rocktäschel, and Sebastian Riedel. Adversarial sets for regularising neural link predictors. In Gal Elidan, Kristian Kersting, and Alexander T. Ihler, editors, *Proceedings of the Thirty-Third Conference on Uncertainty in Artificial Intelligence, UAI 2017, Sydney, Australia, August 11-15, 2017*, 2017. URL http://auai.org/uai2017/proceedings/papers/306.pdf.

Pasquale Minervini, Matko Bosnjak, Tim Rocktäschel, and Sebastian Riedel. Towards neural theorem proving at scale. *arXiv*, abs/1807.08204, 2018. URL http://arxiv.org/abs/1807.08204.

Pasquale Minervini, Matko Bosnjak, Tim Rocktäschel, Sebastian Riedel, and Edward Grefenstette. Differentiable reasoning on large knowledge bases and natural language. In *The Thirty-Fourth AAAI Conference on Artificial Intelligence*, pages 5182–5190, 2020a. URL https://aaai.org/ojs/index.php/AAAI/article/view/5962.

Pasquale Minervini, Sebastian Riedel, Pontus Stenetorp, Edward Grefenstette, and Tim Rocktäschel. Learning reasoning strategies in end-to-end differentiable proving. In *Proceedings of the 37th International Conference on Machine Learning, ICML 2020, 13-18 July 2020, Virtual Event*, volume 119 of *Proceedings of Machine Learning Research*, pages 6938–6949, 2020b. URL http://proceedings.mlr.press/v119/minervini20a.html.

Stephen Muggleton. Inductive logic programming. *New Generation Computing*, 8:295–318, 1991.

Stephen H. Muggleton, Dianhuan Lin, and Alireza Tamaddoni-Nezhad. Meta-interpretive learning of higher-order dyadic datalog: predicate invention revisited. *Machine Learning*, 100 (1):49–73, Jul 2015. URL https://doi.org/10.1007/s10994-014-5471-y.

Maximilian Nickel, Lorenzo Rosasco, and Tomaso A. Poggio. Holographic embeddings of knowledge graphs. *arXiv*, abs/1510.04935, 2015. URL http://arxiv.org/abs/1510.04935.

Pouya Ghiasnezhad Omran, Kewen Wang, and Zhe Wang. Scalable rule learning via learning representation. In *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI 2018, July 13-19, 2018*, pages 2149–2155, 2018. URL https://www.ijcai.org/Proceedings/2018/0297.pdf.

Meng Qu, Junkun Chen, Louis-Pascal A. C. Xhonneux, Yoshua Bengio, and Jian Tang. Rnnlogic: Learning logic rules for reasoning on knowledge graphs. *arXiv*, abs/2010.04029, 2020. URL https://arxiv.org/abs/2010.04029.

Tim Rocktäschel and Sebastian Riedel. End-to-end differentiable proving. In *Advances in Neural Information Processing Systems 30*, pages 3788–3800. 2017. URL http://papers.nips.cc/paper/6969-end-to-end-differentiable-proving.pdf.

Tim Rocktäschel, Sameer Singh, and Sebastian Riedel. Injecting logical background knowledge into embeddings for relation extraction. In *Proceedings of the 2015 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 1119–1129, May–June 2015. URL https://aclanthology.org/N15-1118.

Peter Rousseeuw. Silhouettes: A graphical aid to the interpretation and validation of cluster analysis. *J. Comput. Appl. Math.*, 20(1):53–65, November 1987. URL http://dx.doi.org/10.1016/0377-0427(87)90125-7.

David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. *Learning Representations by Back-Propagating Errors*, page 696–699. 1988. ISBN 0262010976.

Rudolph J. Rummel. Dimensionality of nations project: Attributes of nations and behavior of nation dyads, 1950-1965. *MI: Inter-university Consortium for Political and Social Research*, pages 80–4, 1992. URL https://doi.org/10.3886/ICPSR05409.v1.

Prithviraj Sen, Breno W. S. R. de Carvalho, Ryan Riegel, and Alexander G. Gray. Neuro-symbolic inductive logic programming with logical neural networks. In *Thirty-Sixth AAAI Conference on Artificial Intelligence, AAAI 2022*, pages 8212–8219, 2022. URL https://ojs.aaai.org/index.php/AAAI/article/view/20795.

Luciano Serafini and Artur S. d'Avila Garcez. Logic tensor networks: Deep learning and logical reasoning from data and knowledge. In *Proceedings of the 11th International Workshop on Neural-Symbolic Learning and Reasoning (NeSy'16) co-located with the Joint Multi-Conference on Human-Level Artificial Intelligence (HLAI 2016), July 16-17, 2016*, volume 1768 of *CEUR Workshop Proceedings*, 2016. URL http://ceur-ws.org/Vol-1768/NESY16_paper3.pdf.

Tom Silver, Ashay Athalye, Joshua B. Tenenbaum, Tomas Lozano-Perez, and Leslie Pack Kaelbling. Learning neuro-symbolic skills for bilevel planning. *arXiv*, abs/2206.10680, 2022. URL http:arxiv.org/abs/2206.10680.

Richard Socher, Danqi Chen, Christopher D. Manning, and Andrew Y. Ng. Reasoning with neural tensor networks for knowledge base completion. In *Proceedings of the 26th International Conference on Neural Information Processing Systems - Volume 1*, NIPS'13, pages 926–934, 2013. URL http://dl.acm.org/citation.cfm?id=2999611.2999715.

Théo Trouillon, Johannes Welbl, Sebastian Riedel, Éric Gaussier, and Guillaume Bouchard. Complex embeddings for simple link prediction. *arXiv*, abs/1606.06357, 2016. URL http://arxiv.org/abs/1606.06357.

Efthymia Tsamoura, Timothy M. Hospedales, and Loizos Michael. Neural-symbolic integration: A compositional perspective. In *Thirty-Fifth AAAI Conference on Artificial Intelligence, AAAI 2021*, pages 5051–5060, 2021. URL https://ojs.aaai.org/index.php/AAAI/article/view/16639.

Wenguan Wang and Yi Yang. Towards data-and knowledge-driven artificial intelligence: A survey on neuro-symbolic computing. *arXiv*, abs/2210.15889, 2022. URL http://arxiv.org/abs/2210.15889.

Ronald J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Mach. Learn.*, 8(3–4):229–256, May 1992. URL https://doi.org/10.1007/BF00992696.

Shuang Xia, Krysia Broda, and Alessandra Russo. Topical neural theorem prover that induces rules. In *GCAI 2020. 6th Global Conference on Artificial Intelligence (GCAI 2020)*, volume 72 of *EPiC Series in Computing*, pages 107–120, 2020. URL https://easychair.org/publications/paper/mFsC.

Bishan Yang, Wen-tau Yih, Xiaodong He, Jianfeng Gao, and Li Deng. Embedding entities and relations for learning and inference in knowledge bases. In *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015. URL http://arxiv.org/abs/1412.6575.

Fan Yang, Zhilin Yang, and William W. Cohen. Differentiable learning of logical rules for knowledge base reasoning. In *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017*, pages 2319–2328, 2017. URL https://proceedings.neurips.cc/paper/2017/hash/0e55666a4ad822e0e34299df3591d979-Abstract.html.

Zhun Yang, Adam Ishay, and Joohyung Lee. NeurASP: Embracing neural networks into answer set programming. In *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI 2020*, pages 1755–1762, 2020. URL https://doi.org/10.24963/ijcai.2020/243.

Yifeng Zhu, Jonathan Tremblay, Stan Birchfield, and Yuke Zhu. Hierarchical planning for long-horizon manipulation with geometric and symbolic scene graphs. *2021 IEEE International Conference on Robotics and Automation (ICRA)*, pages 6541–6548, 2020. URL https://arxiv.org/abs/2012.07277.