RHEINISCHE FRIEDRICH-WILHELMS-UNIVERSITÄT BONN
INSTITUT FÜR INFORMATIK III

Relational Patterns: Explicit vs Implicit Learning

Master Thesis

Reviewer 1: Prof. Dr. Jens Lehmann
Reviewer 2: Steffen Lohmann

Nadezhda Vassilyeva

February 2020

Declaration of Authorship

I hereby declare that all the work described within this Master thesis is the original work of the author. Any published (or unpublished) ideas or techniques from the work of others are fully acknowledged in accordance with the standard referencing practices.

Nadezhda Vassilyeva

Signed: _____

Date: _____

To Hendrik Siebe,
whose practical and moral support and belief in me made this work possible.

Acknowledgements

First of all, I would like to thank my supervisors, Mojtaba Nayyeri and Sahar Vahdati. This thesis was developed in close collaboration with the work of Mojtaba Nayyeri, and his support and guidance were invaluable to this work. Both his and Sahar Vahdati's dedication to research served as a source of inspiration and encouragement to me, and taught me the value of integrity, curiosity, and commitment in research.

I would also like to express gratitude to my examination committee for their time and attention in evaluating this thesis.

Last, but not least, I am grateful to my friends and family, who supported and encouraged me during this work.

Contents

Chapter 1

Introduction

Graphs are one of the most natural structures to encode knowledge. They afford us an overview of not only the data, but more importantly, they shows us how the data connect or *relate* to each other. Maps of public transport, taxonomies, and mind maps are all examples of uses of graphs to represent information.

Knowledge graphs are a formalization of this intuitive approach. A knowledge graph (KG) is a knowledge base organized as a multi-relational graph $\mathcal{G} = (\mathcal{E}, \mathcal{R})$. The nodes $\mathcal{E}$ represent entities and the labeled edges $\mathcal{R}$ define relations between them. By representing relations as labeled edges, KGs treat them as first-class citizens [1]. This makes KGs particularly well-suited for modeling complex and highly interconnected data, and data that has many different types of relations. They are also well-suited to modeling data that frequently changes – to add or delete a piece of data, one only needs to add/remove relevant edges and/or entities in the KG. There is no need to redesign the schema or to alter other parts of the graph.

Considering the intuitiveness and expressiveness of KGs, it is no surprise that the last two decades have seen an explosion of interest in them, both within academia and in industry [2]. Many different KGs were constructed, modeling anything from general knowledge to social networks to human languages. The most prominent among them are Freebase [3], DBPedia [4], WordNet [5], NELL [6], YAGO [7], and ConceptNet [8].

Today, knowledge graphs have a wide range of applications in natural language processing, medicine, law, biology, and other fields. However, KGs represent data

symbolically, and as a result, are difficult to manipulate [9]. Knowledge graph embedding (KGE) offers one solution to this problem.

KGE finds a mapping from entities and relations into a low-dimensional continuous vector space. Since 2010, many different embedding models were proposed. A lot of them aim to learn embeddings from the structure of the KG alone. However, a knowledge graph is more than the sum of its triples [10]. Entities often have textual descriptions associated with them, and relations have meanings beyond what is explicitly stated in the KG. That is, relations fall into certain *patterns*. Capturing these patterns in a KGE model has been shown to be crucial to its performance [10], [11].

Some models aim to learn to differentiate between types of relational patterns implicitly, while others focus instead on including relational patterns in the learning process explicitly, in the form of logical rules.

In this work, we focus on embedding models and their ability to infer relational patterns. We are motivated by two research questions:

Research question 1: Does combining different types of models affect their performance?

In addressing this question, we develop a flexible KGE suite MultEmbed. In it, we design and implement experimental embedding models based on RotatE [11] and QuatE [12]. We evaluate these models and compare them to the state-of-the-art embedding models. We further test their ability to capture different relational patterns.

Research question 2: Does injecting logical rules into a model that is already learning relational patterns implicitly improve its performance?

To answer this question, we implement a rule-injection method based on LogicENN [13]. We use it to empirically test performance of our experimental KGE models with and without the explicit rule injection.

The rest of this thesis is organized as follows: in Chapter 2, we give background information that is relevant to this work. We also review current state-of-the-art approaches in KGE.

In Chapter 3, we discuss the process we followed in answering our research questions. Our KGE suite MultEmbed is introduced here, and a high-level explanations of its constituent parts is given.

Chapter 4 elaborates on this and describes concrete implementation details of MultEmbed. It also details how to use MultEmbed.

Finally, in Chapter 5, we evaluate the models included in MultEmbed on four standard KGs. We answer our research questions here.

We close with Chapter 6, in which we draw conclusions and suggest directions for future research.

Chapter 2

Background and Related Work

In section 2.1, knowledge graph (KG), as well as relevant surrounding concepts (e.g. *embedding model*, *relational patterns*), are introduced and explained in detail. Notation to be used throughout this work is established, and a short review of relevant mathematical tools and concepts is given.

In section 2.2, we review previous work in KG embedding (KGE) models. We explain different KGE models and discuss their strengths and weaknesses. We aim to familiarize the reader with the field and to situate our work within a broader context.

2.1 Background

2.1.1 Knowledge graph

Representing knowledge in a form of a network has a long history in computer science. Semantic nets - a precursor to modern knowledge graphs - was first proposed by R. H. Richens in 1956. His goal was to facilitate automated translation by extracting information that is "invariant during translation", that is, the *meaning* of the sentence being translated [14]. This is achieved by representing sentences as graphs that link objects, relations, and qualities, as shown in Fig 2.1.

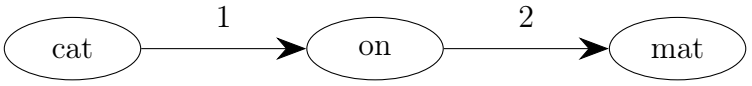


FIGURE 2.1: Semantic Net representation of the sentence "Cat is on the mat". Relations (*on*) and entities (*cat* and *mat*) both are represented as nodes.

In early 1990s, Stokman and de Vries [15] and van de Riet and Meersman [16] expanded on this work. Their aim was to encode scientific theories from medical and social sciences into a semantic net, or *knowledge graph*, as they called it. They saw structural dependencies between entities (i.e. relations) as "the core of scientific knowledge" and proposed to model them as edges [15]. In contrast to the previous semantic nets, the set of allowed relations was restricted.[1]

Since then, knowledge graphs have been steadily gaining in popularity, with many large-scale KGs constructed. Most notable examples include WordNet [5], ConceptNet [8], Freebase [3], DBpedia [4], YAGO [7], and NELL [6]. However, the term has really come into prominence in 2012 when Google came out with its own knowledge graph [18]. Google KG is used to improve their search engine by enabling it to "understand" the information it finds. This information is then presented to the user in a concise "knowledge panel" (see Fig 2.2), facilitating information gathering and further exploration.

But what exactly is a KG? Despite the popularity of KGs in recent decades, no formal, universally accepted definition exists [18]. Because of this, it is more informative to describe the structure and functionality of KGs.

[1]Since then, the terms *semantic network* and *knowledge graphs* became synonymous, and both are often used interchangeably [17].

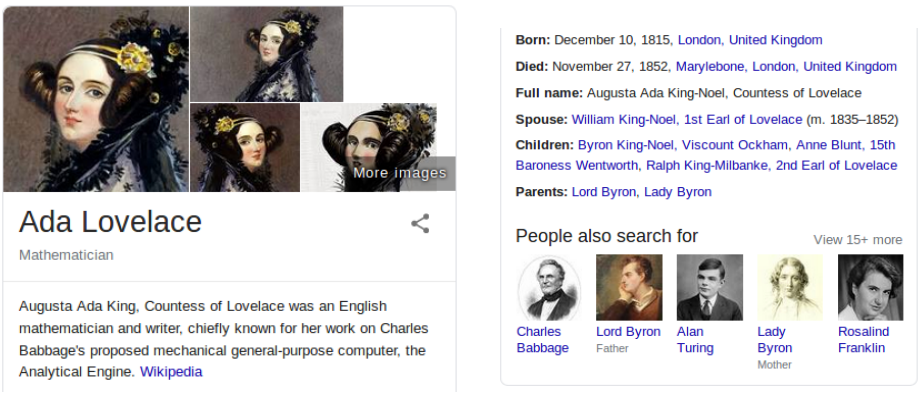


FIGURE 2.2: Google's knowledge panel, showing results for the search term *Ada Lovelace*. The search term is understood as an entity ("thing, not string" in Google parlance [19]). Relevant information about the entity is displayed and further exploration is facilitated by linking to similar entities in the *People also search for* part of the panel. Screenshot taken from Google page results for "Ada Lovelace" search; page accessed on Dec. 04, 2019.

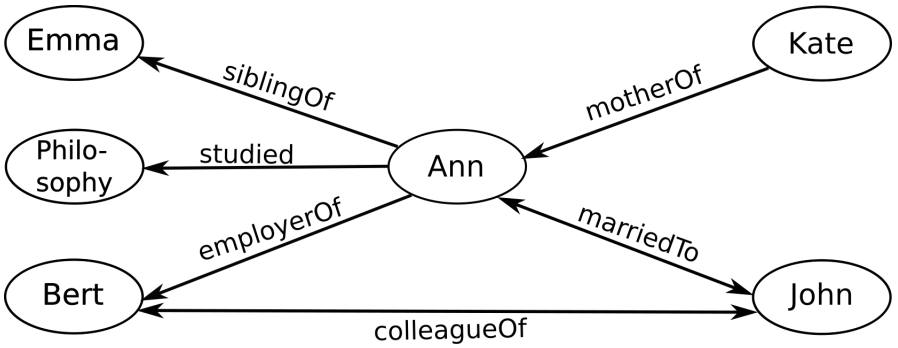


FIGURE 2.3: An example of a KG. Nodes represent entities and edges - relations between these entities. Bi-directional arrows indicate that the relationship applies in both directions: e.g. both (*Ann*, *marriedTo*, *John*) and (*John*, *marriedTo*, *Ann*) hold.

A KG is a multi-relational, labeled, directed graph that represents entities as nodes and relations between these entities as edges. An entity can be any physical object or an abstract concept (e.g. *Ann* and *Philosophy* respectively in Figure 2.3). KGs treat relations as first-class citizens. This makes them particularly well-suited for representing structured, highly interconnected data that has many different types of relations. Furthermore, the graph structure is one of the most intuitive ways to organize information and to provide human-readable, clear overview of how different parts and domains of the data connect to each other. Finally, a KG is self-descriptive: all the data and its meaning are contained in the KG.

The graph data is stored in the form of Resource Description Framework (RDF) triples. RDF is an abstract data model used to provide conceptual description of

information. It unambiguously models information as a set of subject-predicate-object triples (s, p, o). The subject s is the resource being described, the predicate p is a trait or aspect of the resource, and the object o can be either another resource or a *literal*, i.e. a data value, description, or another entity. For instance, statement "Speed of light is about $3.0{\times}10^5$ km/s" can be expressed by an RDF triple ("Light", "has speed", "3×10^5 km/s"), where "3×10^5 km/s" is a literal. RDF is an abstract model with many serialization formats available. In the context of KGs, we use notation (h, r, t) instead of (s, p, o) with h for *head* or subject of the triple, t - *tail* or object, and r - the predicate, or *relation* between h and t. Figure 2.4 shows how a fact can be represented as an RDF triple, and how that triple is represented as a graph.

$$(A.Merkel,\ chancellorOf,\ Germany)$$

FIGURE 2.4: An RDF triple and its graph representation of the fact: "Angela Merkel is the chancellor of Germany."

Another important aspect of a KG is its functionality. A KG must be able to generate new knowledge through a reasoning agent, and it must be capable of integrating information from one or more sources [18].

2.1.2 Notation and mathematics primer

Let $\mathcal{G} = (\mathcal{E}, \mathcal{R})$ be a KG where $\mathcal{E}$ is the set of its nodes (i.e. entities) and $\mathcal{R}$ is the set of its labeled edges (i.e. relations). The set of all possible triples in $\mathcal{G}$ is denoted by $\mathcal{T} = \mathcal{E} \times \mathcal{R} \times \mathcal{E}$. *Positive triples* or *facts* are denoted by Ω^+, and are a set of all triples explicitly contained in $\mathcal{G}$. We say that a triple t holds if $t \in \Omega^+$. Conversely, Ω^- represents the set of *negative triples*. We operate under the *closed world assumption* (CWA), which assumes that any triple not explicitly contained in $\mathcal{G}$ is false, i.e. it is in Ω^-. Under this assumption, it holds that $\mathcal{T} = \Omega^+ \cup \Omega^-$ and $\Omega^+ \cap \Omega^- = \emptyset$.

Entities and relations in a KG are denoted by italicized lower-case letters, and their embeddings (to be explained in the following section) - by lower-case bold letters. To keep the notation consistent, all vectors are represented by lower-case bold letters. For example, an embedding of $h \in \mathcal{E}$ is a vector $\mathbf{h}$. Matrices are

represented by boldface uppercase letters $\mathbf{M}$. An i'th element of a vector $\mathbf{v}$ or (i, j)'th element of a matrix $\mathbf{M}$ is represented by $\mathbf{v}_i$ and $\mathbf{M}_{ij}$ respectively.

Hadamard (or element-wise) product of two vectors $\mathbf{v}$ and $\mathbf{u}$ is represented by $\mathbf{v} \circ \mathbf{u}$ and is defined as: $(\mathbf{v} \circ \mathbf{u})_i = \mathbf{v}_i \mathbf{u}_i$. L$p$ norm is denoted by $\|\cdot\|_p$ for any $p \in \mathbb{N},\ p \geq 1$ and is computed for some vector $\mathbf{v}$ as follows: $\|\mathbf{v}\|_p \coloneqq (\sum_{i=1}^{n} |\mathbf{v}_i^p|)^{1/p}$, where n is the length of $\mathbf{v}$. Transpose of a vector $\mathbf{v}$ is denoted by $\mathbf{v}^\top$.

For any complex number $v \in \mathbb{C},\ v^\star$ or $\bar{v}$ denotes its complex conjugate, i.e.: if $v = x + y\mathbf{i}$, then $v^\star = x - y\mathbf{i}$. For any real number $r \in \mathbb{R}$, Euler's formula states that the following equality holds:

$$e^{\mathbf{i}r} = \cos r + \mathbf{i} \sin r \tag{2.1}$$

This implies that multiplying any complex number z by $e^{\mathbf{i}r}$ can be interpreted geometrically as counterclockwise rotation of z by angle r on the complex plane [11]. Quaternions are an extension of complex numbers, and are defined as follows: $v = a + b\mathbf{i} + c\mathbf{j} + d\mathbf{k}$ where a is its real part and $b,\ c$, and d are the magnitudes of the imaginary parts $\mathbf{i}$, $\mathbf{j}$, and $\mathbf{k}$ respectively. A complex conjugate of a quaternion number v is defined as $v^\star = a - b\mathbf{i} - c\mathbf{j} - d\mathbf{k}$. Like a multiplication by a complex number, multiplication by a unit quaternion - that is, quaternion whose norm is 1 - also represents rotation. However, quaternion rotation is a rotation in 4-dimensional space, and is, therefore, more expressive than rotation on the complex plane [12].

Hamilton product, or quaternion product, of two quaternion vectors $q_1 = a_1 + b_1\mathbf{i} + c_1\mathbf{j} + d_1\mathbf{k}$ and $q_2 = a_2 + b_2\mathbf{i} + c_2\mathbf{j} + d_2\mathbf{k}$ is defined as

$$\begin{aligned} q_1 \otimes q_2 = &(a_1 \circ a_2 - b_1 \circ b_2 - c_1 \circ c_2 - d_1 \circ d_2) + \\ &(a_1 \circ b_2 + b_1 \circ a_2 + c_1 \circ d_2 - d_1 \circ c_2)\,\mathbf{i} + \\ &(a_1 \circ c_2 - b_1 \circ d_2 + c_1 \circ a_2 + d_1 \circ b_2)\,\mathbf{j} + \\ &(a_1 \circ d_2 + b_1 \circ c_2 - c_1 \circ b_2 + d_1 \circ a_2)\,\mathbf{k} \end{aligned} \tag{2.2}$$

where $\circ$ is the Hadamard product of two vectors.

2.1.3 Embedding models

The very strength of KGs - their intuitive way of representing highly structured data - is also their main weakness. Symbolic representation of triples in KGs make them unsuitable for most machine learning tools. Recently, it has been proposed to solve this problem by embedding KGs. Knowledge graph embedding (KGE) is the task of finding vector representations of KG's entities and relations such that information about the structure of the KG is preserved. More formally, given a KG $\mathcal{G} = (\mathcal{E}, \mathcal{R})$, the goal is to find mappings $f_e : \mathcal{E} \mapsto \mathbb{R}^d$ (or $\mathbb{C}^d$) and $f_r : \mathcal{R} \mapsto \mathbb{R}^k$ (or $\mathbb{C}^k$), where d is referred to as *entity embedding dimension* and k - *relation embedding dimension*.

There are two main approaches to KGE: observed feature models and latent feature models. Observed feature models focus on symbolic representation of entities and relations and use first-order logic, graph-based features, and probabilistic graphical models to learn embeddings. Observed feature models include statistical relational learning approaches such as Relational Dependency Networks [20], Bayesian Logic Programs [21], and Probabilistic Relational Models [22]. Another approach to observed feature models, graph-based models, focuses on structural properties of KGs [9]. One of the most representative graph-based embedding models in this class is the Path Ranking Algorithm [23], [24].

In this work, we focus on latent feature models. A latent feature model consists of an *embedding representation*, a *scoring function* $\phi(h, r, t)$ and a *loss function*. Embedding representation refers to the form that embeddings take. The most common approach is to associate each entity and each relation with a unique vector in continuous real or complex space. Alternatively, some models associate a vector with entity-relation pairs instead, i.e. given
$\mathcal{P} = \{(r, e) | \forall r \in \mathcal{R}, e \in \mathcal{E}, \text{ s.t. } (e, r, \cdot) \in \mathcal{G} \text{ or } (\cdot, r, e) \in \mathcal{G}\}$ - the set of all entity-relation pairs present in the KG, these models find a mapping $f : \mathcal{P} \mapsto \mathbb{R}^d$ (or $\mathbb{C}^d$). The scoring function $\phi(h, r, t)$ takes a triple as an input and computes the probability that it holds. It is the most important part of a KGE model, at least for definitional purposes. That is, while many models share the same loss functions and the same embedding representations, each model is associated with a unique scoring function. Finally, the loss function guides the learning of the embeddings. Its objective is to maximize $\phi(\cdot)$ for positive triples, and to minimize it for the negative ones.

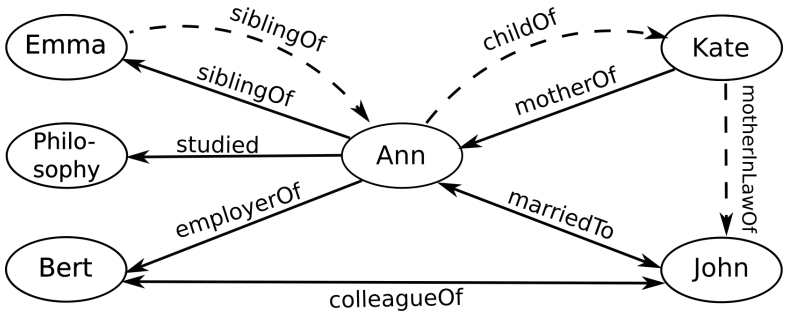


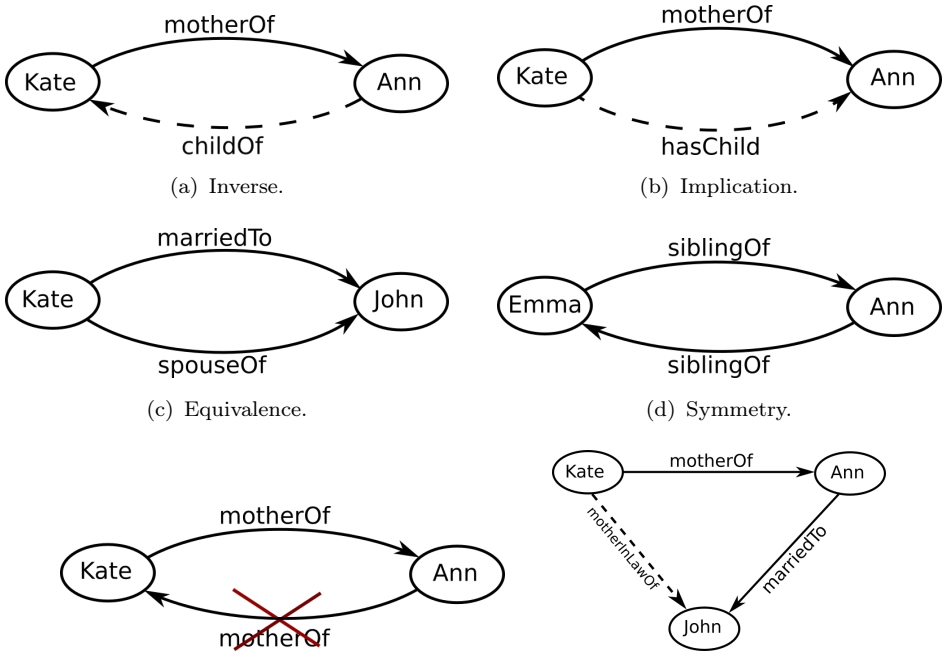
FIGURE 2.5: Graph from Figure 2.3. Relations represented by dashed lines can be directly inferred from 2.3.

2.1.4 Types of relations

A knowledge graph is more than the sum of its triples. Entities do not exist in a vacuum - they have textual descriptions and they have types - that is, categories to which they belong. Analogously, relations have meaning beyond what is explicitly stated in KG triples. Many relations exhibit a particular *relational pattern*. Understanding these relation types allows us to infer new knowledge not explicitly stated in the graph. Consider for instance the graph in Figure 2.3. Using our understanding of the relations involved, inferring new knowledge is trivial. We know automatically that Emma is Ann's sibling, that Kate is John's mother-in-law, and that Ann is Kate's child. These inferred facts are illustrated by dashed lines in Figure 2.5 and follow directly from the nature of the relations *siblingOf*, *motherOf*, and *motherOf*+*marriedTo*.

We say that a relation r exhibits a particular relational pattern: e.g. *marriedTo* is symmetric). Alternatively, relational patterns can be stated using first-order logic rules: e.g. $\forall\ h, t \in \mathcal{E}\ :\ (h,\ motherOf,\ t) \Rightarrow (t,\ childOf,\ h)$. When referring to patterns as rules, we call the premise part of the rule, e.g. $(h,\ motherOf,\ t)$ *antecedent*, and the triple that follows from it, e.g. $(t,\ childOf,\ h)$, *consequent*. To simplify notation, we omit explicit universal quantifier $\forall$ and use $?h$ to denote that h is an uninstantiated variable. In other words, $?h$ is a variable that can take on different values, and h indicates a concrete instantiation of it. Using this notation, the above rule can be rewritten as $(?h,\ motherOf,\ ?t) \Rightarrow (?t,\ childOf,\ ?h)$. We say a rule is *instantiated* (or *grounded*) if all its variables are substituted with concrete entities [25], [26], [13].

Here, we focus on 5 common relational patterns in a KG: inverse, implication,



(a) Inverse.

(b) Implication.

(c) Equivalence.

(d) Symmetry.

(e) Asymmetry: red X represents that the edge does not exist in the KG.

(f) Composition.

FIGURE 2.6: Examples of different types of relation patterns. Dashed arrows represent consequent triples, where applicable. For instance, 2.6(b) indicates that $(?h,\ motherOf,\ ?t) \Rightarrow (?h,\ hasChild,\ ?t)$ holds, while the converse need not necessarily follow.

Rule type	**Definition**	**Example**
Inverse	$(?h, r_1, ?t) \Rightarrow (?t, r_2, ?h)$	Fig. 2.6(a)
Implication	$(?h, r_1, ?t) \Rightarrow (?h, r_2, ?t)$	Fig. 2.6(b)
Equivalence	$(?h, r_1, ?t) \Leftrightarrow (?h, r_2, ?t)$	Fig. 2.6(c)
Symmetry	$(?h, r, ?t) \Leftrightarrow (?t, r, ?h)$	Fig. 2.6(d)
Asymmetry	$(?h, r, ?t) \Rightarrow \neg(?t, r, ?h)$	Fig. 2.6(e)
Composition	$(?h, r_1, ?t) \wedge (?t, r_2, ?s) \Rightarrow (?h, r_3, ?s)$	Fig. 2.6(f)

TABLE 2.1: Logical rules encoding relational patterns. Definitions should be understood as including implicit universal quantifier $\forall\ h, t, s \in \mathcal{E}$.

equivalence, symmetry/asymmetry, and composition. Table 2.1 defines these patterns and Figure 2.6 shows examples.

2.2 Related work

In the last decade, knowledge graph embedding has quickly gained in popularity, with numerous latent feature embedding models developed. These can be roughly

categorized into *translational*, *semantic matching*, and *rotational* models. Translational models, like the name suggests, view relation as a translation from the head to the tail entity. These models use distance-based scoring functions. Semantic matching types of models rely on similarity-based scoring. And rotational models conceptualize relation as a rotation from the head to the tail entity.

2.2.1 Translational models

One of the simplest embedding models to be proposed is the **TransE**. It was published in 2013 by Bordes et al. in **Trans**lating **E**mbeddings for Modeling Multi-relational Data [27]. Despite its simplicity and its early publication date, it remains a paradigmatic translational model. In this model, relations are interpreted as translations between entities, and the goal is to find embeddings such that for any true triple $(h,\ r,\ t)$, $\mathbf{h} + \mathbf{r} \approx \mathbf{t}$ holds. Consequently, the scoring function is defined as [27]:

$$\phi(h, r, t) = -\|\mathbf{h} + \mathbf{r} - \mathbf{t}\|_{1/2} \tag{2.3}$$

where $\|\cdot\|_{1/2}$ means either L1 or L2 norm.

The simplicity of TransE makes it highly scalable. However, as can be easily deduced from its scoring function, TransE has difficulties representing 1-MANY, MANY-1, and MANY-MANY relations. To see why, consider a set of n triples (h_i, r, t), for $i = 1, ..., n$. TransE requires that $\mathbf{h}_i + \mathbf{r} \approx \mathbf{t} \quad \forall i = 1 \ldots n$. As a result, it follows that $\mathbf{h}_i \approx \mathbf{h}_j$ for $i = 1, ..., n$, $j = 1, ..., n$. In words, TransE forces embeddings of entities on the MANY side of a relation to be very similar to each other [27], [9].

Bordes et al. estimate that only 26.2% of relations in FB15k are 1-to-1, with the remaining split roughly equally between 1-to-MANY, MANY-to-1, and MANY-to-MANY [27]. Because of this, much attention has been devoted to extending TransE to improve modeling of these types of relations. This resulted in several derivative models.

One such model is **Trans**lation on **H**yperplanes (**TransH**), developed in 2014 by Wang et al. [28]. Its scoring function is defined as:

$$\phi(h, r, t) = -\|\mathbf{h}_\perp + \mathbf{r} - \mathbf{t}_\perp\|_2^2 \tag{2.4}$$

where

$$\begin{aligned} \mathbf{h}_\perp &= \mathbf{h} - \mathbf{w}_r^\top \mathbf{h} \mathbf{w}_r, \\ \mathbf{t}_\perp &= \mathbf{t} - \mathbf{w}_r^\top \mathbf{t} \mathbf{w}_r, \end{aligned} \tag{2.5}$$

and $\mathbf{w}_r$ is a vector normal to the hyperplane on which $\mathbf{r}$ resides. In words, each relation r is associated with a hyperplane. The entities connected by r are first projected onto this hyperplane using equations 2.5 and are only then translated [28]. This allows the entities to have different representations when they are associated with different relations. For instance, embeddings of entities *Berlin*, *Munich*, *Bonn*, and *Cologne* might be similar when they appear in triples $(\cdot,\ locatedIn,\ Germany)$, but might be very different when they are parts of triples with a different relation [9], [28].

Closely after TransH, Lin et al. proposed an alternative extension of TransE. Their model, **TransR**, follows the same idea as TransH, but instead of associating relations with hyperplanes, it associates them with spaces in $\mathbb{R}^k$. Entities are projected into these space as follows:

$$\mathbf{h}_\perp = \mathbf{M}_r \mathbf{h}, \quad \mathbf{t}_\perp = \mathbf{M}_r \mathbf{t} \tag{2.6}$$

where $\mathbf{M}_r \in \mathbb{R}^{k \times d}$ is a projection matrix for the relation r [9], [29].

To avoid space-consuming matrices required by TransR, Ji et al. propose **TransD**. Instead of matrices, it associates each element of the KG (i.e. an entity or a relation) with two distinct vectors. The first vector aims to capture the representation of the element, while the second is used to dynamically compute the mapping matrix. Formally, to embed a triple (h, r, t), representation vectors $\mathbf{h}$,$\mathbf{t} \in \mathbb{R}^d$ and $\mathbf{r} \in \mathbb{R}^k$ and mapping vectors $\mathbf{w}_h, \mathbf{w}_t \in \mathbb{R}^d$ and $\mathbf{w}_r \in \mathbb{R}^k$ are defined. Using these vectors, projection matrices $\mathbf{M}_r^1$ and $\mathbf{M}_r^2$ are computed and then used to project the entities:

$$\begin{aligned} \mathbf{M}_r^1 &= \mathbf{w}_r \mathbf{w}_h^\top + \mathbf{I}, \\ \mathbf{M}_r^2 &= \mathbf{w}_r \mathbf{w}_t^\top + \mathbf{I} \\ \mathbf{h}_\perp = \mathbf{M}_r^1 \mathbf{h}, &\quad \mathbf{t}_\perp = \mathbf{M}_r^2 \mathbf{t}. \end{aligned} \tag{2.7}$$

where **I** is the identity matrix [30]. Both TransR and TransD use the TransH scoring function 2.4.

Fan et al. take another approach to improve modeling of the problematic, MANY-sided relations. Instead of using different representations of entities depending on a relation, their model **TransM** uses a weighted scoring function:

$$\phi(h, r, t) = -\theta_r \|\mathbf{h} + \mathbf{r} - \mathbf{t}\| \tag{2.8}$$

where θ_r is a relation-specific weight, with 1-to-MANY, MANY-to-1, and MANY-to-MANY relations assigned lower weights. This prevents the entities on the MANY side of these relations from collapsing into the same embedding by allowing greater distance between $\mathbf{h} + \mathbf{r}$ and $\mathbf{t}$ for these types of relations [9], [31].

2.2.2 Semantic matching models

Semantic matching models use similarity-based instead of distance-based scoring functions. **RESCAL** is a semantic matching model proposed in 2011 by Nickel et al. [32]. It associates entities with vectors in $\mathbb{R}^d$ and relations with matrices in $\mathbb{R}^{d \times d}$. The matrices model pairwise interactions between all the components of the head and tail embeddings (i.e. between latent features of the head and tail entities). Its scoring function is defined as:

$$\phi(h, r, t) = \mathbf{h}^\top \mathbf{M} \mathbf{t} = \sum_{i=0}^{d-1} \sum_{j=0}^{d-1} [\mathbf{M}]_{ij} [\mathbf{h}]_i [\mathbf{t}]_j \tag{2.9}$$

Although this is a very powerful model, it does not scale well because of the use of matrix multiplication. To simplify it, Yang et al. proposed **DistMult** [33]. Their approach models pairwise interactions between components of head and tail embeddings only in the same dimension. This allows them to use diagnonal matrices:

$$\phi(h, r, t) = \mathbf{h}^\top \mathrm{diag}(\mathbf{M}_r) \mathbf{t} = \sum_{i=0}^{d-1} [\mathbf{M}_r]_{ii} [\mathbf{h}]_i [\mathbf{t}]_i \tag{2.10}$$

2.2.3 Rotational models

Embedding models that rely on dot product, such as DistMult, are incapable of modeling asymmetric relations, since the dot product itself is commutative (see Appendix A.1 for a detailed proof). **CompEx** [34] was proposed as an elegant and scalable way to solve this problem. Their approach is to embed entities and relations into a complex space $\mathbb{C}^d$. Dot product in complex space, aka Hermitian (or sesquilinear), is defined as $\langle u, v\rangle \coloneqq \bar{u}^\top v$ and is not commutative. As a result, ComplEx can model asymmetric relations [9], [34]. Its scoring function is defined as:

$$\begin{aligned}\phi(h,r,t) = \mathrm{Re}(\langle \mathbf{r}, \mathbf{h}, \bar{\mathbf{t}}\rangle) &= \mathrm{Re}\left(\sum_d^{i-1} \mathbf{r}_i \mathbf{h}_i \bar{\mathbf{t}}_i\right) \\ &= \langle \mathrm{Re}(\mathbf{r}), \mathrm{Re}(\mathbf{h}), \mathrm{Re}(\mathbf{t})\rangle + \langle \mathrm{Re}(\mathbf{r}), \mathrm{Im}(\mathbf{h}), \mathrm{Im}(\mathbf{t})\rangle \\ &+ \langle \mathrm{Im}(\mathbf{r}), \mathrm{Re}(\mathbf{h}), \mathrm{Im}(\mathbf{t})\rangle - \langle \mathrm{Im}(\mathbf{r}), \mathrm{Im}(\mathbf{h}), \mathrm{Re}(\mathbf{t})\end{aligned} \tag{2.11}$$

where $\mathbf{r}, \mathbf{h}, \mathbf{t} \in \mathbb{C}^d$ and $\langle\cdot\rangle$ is dot product. Although this model captures symmetric, asymmetric, and inverse relations, it is not expressive enough to model composition relations [11].

RotatE [11] is model that is capable of inferring all four types of relations: inverse, symmetry, asymmetry, and composition. It embeds entities and relations into complex vector space and uses the following scoring function:

$$\phi(h,r,t) = -||\mathbf{h} \circ \mathbf{r} - \mathbf{t}|| \tag{2.12}$$

where $\mathbf{h}, \mathbf{r}, \mathbf{t} \in \mathbb{C}^d$, $\mathbf{r}$ is a unit vector, i.e. $|\mathbf{r}| = 1$, and $\circ$ is Hadamard (element-wise) product. Relation r is then interpreted as a rotation from h to t. Appendix A.1 gives detailed proofs of RotatE's expressivity.

QuatE [12] extends embedding models into quaternion space. Similarly to RotatE, QuatE represents relation as a rotation. However, rotation in quaternion space is more expressive than rotation in complex space: while rotation in complex space has only two planes of rotation, rotation in quaternion space has three [12]. QuatE finds a mapping $\mathcal{E} \to \mathbb{Q}^d$, where an entity h is represented by a quaternion vector $\mathbf{h} = a_h + b_h\mathbf{i} + c_h\mathbf{j} + d_h\mathbf{k}$, $a_h, b_h, c_h, d_h \in \mathbb{R}^d$.

The scoring function is computed as follows:

$$\phi(h, r, t) = \mathbf{h}' \cdot \mathbf{t} = \langle a'_h, a_t \rangle + \langle b'_h, b_t \rangle + \langle c'_h, c_t \rangle + \langle d'_h, d_t \rangle \tag{2.13}$$

where $\langle \cdot, \cdot \rangle$ is inner product. $\mathbf{h}'$ is computed by first normalizing relation embedding $\mathbf{r} = p_r + q_r\mathbf{i} + u_r\mathbf{j} + v_r\mathbf{k}$ to a unit quaternion:

$$\mathbf{r}^{(n)} = \frac{r}{|r|} = \frac{p_r + q_r\mathbf{i} + u_r\mathbf{j} + v_r\mathbf{k}}{\sqrt{p_r^2 + q_r^2 + u_r^2 + v_r^2}} \tag{2.14}$$

and then computing the Hamiltonian product between $\mathbf{r}^{(n)}$ and $\mathbf{h} = a_h + b_h\mathbf{i} + c_h\mathbf{j} + d_h\mathbf{k}$:

$$\begin{aligned} \mathbf{h}' = \mathbf{h} \otimes \mathbf{r}^{(n)} \coloneqq\ & (a_h \circ p - b_h \circ q - c_h \circ u - d_h \circ v) \\ & + (a_h \circ q + b_h \circ p + c_h \circ v - d_h \circ u)\mathbf{i} \\ & + (a_h \circ u - b_h \circ v + c_h \circ p + d_h \circ q)\mathbf{j} \\ & + (a_h \circ v + b_h \circ u - c_h \circ q + d_h \circ p)\mathbf{k} \end{aligned} \tag{2.15}$$

2.2.4 Loss functions

Although the scoring function is central to a KGE model, a loss function also has a significant effect on a model's performance [35], [36]. In this section, we discuss different loss functions.

RotatE adopts **negative sampling loss** (NS) function from [37] with slight modifications. Besides the scoring function, RotatE also contributes a novel negative sampling method: self-adversarial negative sampling. Instead of sampling negative triples uniformly, it samples them according to the following distribution:

$$p(h'_j, r, t'_j|\theta) = \frac{\exp(\alpha\phi(h'_j, r, t'_j))}{\sum_i \exp(\alpha\phi(h'_i, r, t'_i))} \tag{2.16}$$

where α is the temperature of sampling, θ is the current embedding model values, and (h'_i, r, t'_i) is the i-th negative triple [11]. Instead of sampling negative triples according to this distribution and then training the model on that set, they sample uniformly and use $p(\cdot)$ as weights. The loss is then defined as follows:

$$\mathcal{L} = -\log\sigma(\gamma - \phi(h, r, t)) - \sum_{i=1}^{N} p(h'_i, r, t'_i) \log\sigma(\phi(h'_i, r, t'_i) - \gamma) \tag{2.17}$$

σ is the sigmoid function $\sigma = \frac{\exp(x)}{\exp(x)+1}$, γ is a fixed margin, ϕ is the scoring function defined in 2.12, N is the number of negative samples per one positive triple and (h'_i, r, t'_i) is the i-th negative triple [11].

Nayyeri et al. in [35] propose **adaptive margin ranking loss** (AMR) loss function. They show that it significantly improves performance of even simple models like TransE. AMR loss is defined as follows:

$$\mathcal{L} = \lambda \exp(-\sigma\xi^2) + \lambda_+[\phi(h,r,t) - \gamma + \xi)]_+ + \lambda_-[-\phi(h',r,t') + \gamma + \xi]_+ \quad (2.18)$$

where $\lambda \geq 0,\ \lambda_+ \geq 0,\ \lambda_- \geq 0$ and $\sigma \geq 0$ are weight hyperparameters, $[x]_+ = \max(0, x)$, and ξ is the margin that is learned automatically. Besides optimizing the embeddings, this loss function also maximizes margin ξ that separates scores of positive and negative samples.

Another approach to the loss function is to formulate the problem as a classification task. Under this formulation, the positive triples are assigned the label of $+1$ and the sampled negative triples - -1. This is the approach adopted by QuatE. To learn the embeddings, they minimize **regularized logistic** (RL) loss, defined as:

$$\mathcal{L} = \sum_{(h,r,t)\in\Omega\cup\Omega^-} \log(1 + \exp(-Y_{hrt}\phi(h,r,t))) + \lambda_1 \mathrm{L2}_{ent} + \lambda_2 \mathrm{L2}_{rel} \quad (2.19)$$

where Y_{hrt} is the label of triple (h,r,t), $\mathrm{L2}_{ent}$ and $\mathrm{L2}_{rel}$ are L2 regularization terms for entities and relations respectively, and λ_1, λ_2 are their regularization weights.

2.2.5 Using other information

The embedding models that we described so far learn from the structure of the KG alone. However, this fails to take important background information encoded by relational patterns [26], [25]. Some models, such as RotatE and QuatE, learn to infer these patterns implicitly. Another way is to learn relational patterns explicitly, by injecting corresponding logical rules.

One model that takes this approach is **KALE**[26]. It focuses on two types of rules: implication and composition. To model these rules explicitly, they first ground

the rules. Grounding a rule means instantiating that rule with concrete entities. For instance, rule $(?h,\ isCapitalOf,\ ?t) \Rightarrow (?h,\ locatedIn,\ ?t)$ has grounding $(Helsinki,\ isCapitalOf,\ Finland) \Rightarrow (Helsinki,\ locatedIn,\ Finland)$. A grounding is considered valid only if one of its triples is present in the KG [26]. The set of valid groundings is then added to the training set.

KALE associates each triple and each grounding with a soft truth value. For a triple, the soft truth value is based on TransE scoring function and is computed as follows:

$$\pi(h, r, t) = 1 - \frac{1}{3\sqrt{d}}\|\mathbf{h} + \mathbf{r} - \mathbf{t}\|_1 \tag{2.20}$$

where d is the embedding dimension. Truth value of a grounding is computed by combining truth values of its constituent triples using t-norm fuzzy logics [38]:

$$\begin{aligned} \pi(f_1 \wedge f_2) &= \pi(f_1) \cdot \pi(f_2), \\ \pi(f_1 \vee f_2) &= \pi(f_1) + \pi(f_2) - \pi(f_1) \cdot \pi(f_2) \\ \pi(\neg f_1) &= 1 - \pi(f_1) \end{aligned} \tag{2.21}$$

where f_1, f_2 and can be either a triple or a combination of triples. In the latter case, the value is computed recursively and in the former - using equation 2.20. Finally, using these truth values, KALE minimizes a margin-based ranking loss:

$$\mathcal{L} = \sum_{f^+ \in \Omega} \sum_{f^- \in \Omega^-} [\gamma - \pi(f^+) + \pi(f^-)]_+ \tag{2.22}$$

where γ is the margin between positive and negative formulae, and f^- are negative samples. For a grounding, negative samples are generated by replacing relation in the consequent triple with another one at random.

RUGE [25] is another model that aims to model logical rules directly. One of their contributions is that they use soft rules, that is, rules that hold most of the time, but can sometimes be violated. An example of a soft rule is $(?h, bornIn, ?t) \Rightarrow (?h, hasNationality, ?t)$. Although it is not always the case that a person has the nationality of the country that she was born in, it is usually the case.

Each rule is associated with a confidence value, which expresses the probability of it holding. Similar to KALE, RUGE grounds the rules. Unlike KALE, a grounding is taken as valid only if the antecedent triples are observed, and the consequent is not [25]. The training set used by RUGE consists of two parts: 1) set $\Omega^{\mathcal{L}}$ of "labeled"

triples, containing all positive and negative samples, and 2) set of unlabeled triples $\Omega^{\mathcal{U}}$ containing the consequent triples of the grounded rules. Each triple in $\Omega^{\mathcal{U}}$ is associated with a soft label, which expresses the probability of it being true. The model alternates between predicting the soft labels and learning embeddings.

Rules can also be injected into models through regularization terms. This approach taken by Nayyeri et al. in [13]. Their work is based on the idea that a rule affects a triple's probability of being true. For instance, if we have an instantiated inverse rule $(h, r_1, t) \implies (t, r_2, h)$, we can deduce that the probability of the consequent triple (t, r_2, h) must be at least as high as that of the antecedent triple (h, r_1, t). The rule tells us that every time (h, r_1, t) holds, (t, r_2, h) must necessarily also be true. However, (t, r_2, h) can be true even if (h, r_1, t) is false. For each rule, the difference between the scores of the antecedent and the consequent triples are computed and added to the loss function as a regularization term.

Chapter 3

Approach

This work introduces MultEmbed, a new suite of KGE models. Our suite is flexible and allows the user a choice of an embedding model and a loss function with which to train it. Values of hyperparameters, such as embedding dimension, number of training steps, and so on, can be set at runtime. Supported models include state-of-the-art RotatE and QuatE and experimental variations of these.

This chapter describes how MultEmbed was developed. In Section 3.1, we discuss the motivation behind our work. Section 3.2,discusses the development process itself.

3.1 Motivation

The past decade has given us a wealth of different KGE models. More importantly, research has shown that a model's ability to learn relational patterns, such as symmetry/asymmetry, inversion, equivalence, and composition, as described in Section 2.1.4, is crucial to its performance [11], [34], [10], [25], [26]. Out of these, symmetry, asymmetry, inverse, and composition are the most frequent patterns [11], [10].

Despite the prevalence of these patterns, KGE models generally struggle representing all of them, unless they are learned explicitly. For instance, TransE, as well as its extensions, cannot capture symmetric relations [11]. DistMult, and all other models that rely on dot product in $\mathbb{R}^d$, cannot represent asymmetric relations. Table 3.1 summarizes this information, and Appendix A.1 gives detailed proofs.

Although a model's expressivity - and performance - can be improved by injecting rules explicitly, these improvements do not come cheap. Many ways of injecting logical rules require them to be grounded, i.e., to be instantiated using actual entities. But this produces a lot of grounded rules per one logical rule. As a result, these models do not scale well [39], [11].

In this work, we want to investigate how important explicit rule injection is, and whether models that already learn relational patterns implicitly still benefit from rule injection. We want to facilitate research into this question by developing an embedding suite that allows a user to inject rules explicitly.

RotatE is the first model to effectively capture all of the four relational patterns [11]. Although Sun et al. argue that RotatE's high expressivity is its advantage, Zhang et al. claim in [12] that it is actually a weakness. They argue that TransE's

	Symmetry	**Asymmetry**	**Composition**	**Inverse**
TransE	✗	✓	✓	✓
DistMult	✓	✗	✗	✗
ComplEx	✓	✓	✗	✓
RotatE	✓	✓	✓	✓
QuatE	✓	✓	✗	✓

TABLE 3.1: Expressivity of KGE models.

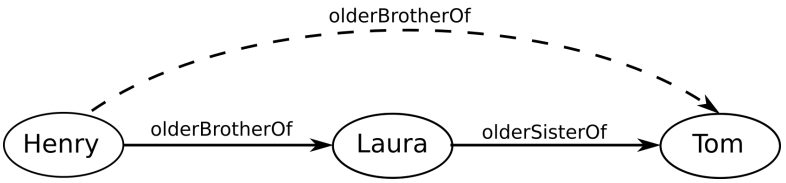


FIGURE 3.1: Composition relation that is problematic for RotatE. Solid black arrows represent relations that are explicitly given and the dashed - a relation that can be directly inferred. This example is taken from [12]

and RotatE's restriction on how composition patterns are modeled puts them them at a disadvantage. To see why, consider an example illustrated in Figure 3.1. In this example, we are given information that $(Henry, olderBrotherOf, Laura)$ and $(Laura, olderSisterOf, Tom)$. However, this implies that the relation between Henry and Tom is $olderBrother$, and not $olderBrother \circ olderSister$ or $olderBrother + olderSister$, as required by RotatE and TransE respectively [12]. This leads us to question if removing this restriction improves a model's performance.

Recently, it has been observed by both [35] and [36] that loss function has a significant impact on a model's performance. In light of this, we want to facilitate empirical exploration of KGE model performances. Importantly, we want to enable the user to train models with different loss functions and observe how that changes performance.

To this end, we develop MultEmbed - the flexible KGE suite that supports several embedding models, allows a user a choice of loss function, and optionally explicitly injects logical rules.

3.2 Methodology

Fig. 3.2 shows the entire process of development of MultEmbed. Before starting on implementation or analysis, an extensive literature review phase took place. Although the field of KGE is relatively new, it is very active, with numerous embedding models published every year. The literature exploration phase included study of different embedding models, loss functions, as well as methods of explicit rule injection. Although in Section 2.2 focuses primarily on latent feature models, we have also explored neural network approaches. Chief among them were the Neural Association Model (NAM) [40], ConvE [41], and LogicENN [13]. NAM and LogicENN both use deep neural network architectures, while ConvE relies on convolutional neural networks.

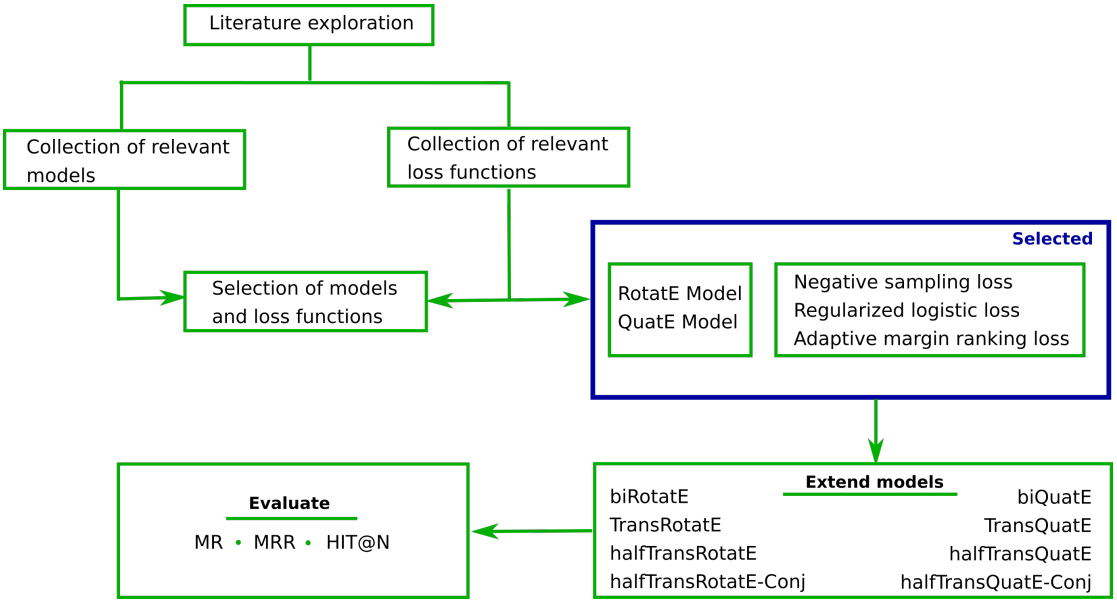


FIGURE 3.2: Workflow diagram: the work behind MultEmbed.

As promising as neural networks are in many other domains of artificial intelligence, we ultimately decided to use simpler, straightforward models. Neural networks result in black-box solutions, making analysis of expressivity of the model virtually impossible [11] Moreover, deep neural networks have considerably more parameters than simpler models such as RotatE or TransE. This makes training them a lengthier, more resource-heavy process, and makes them prone to overfitting [42]. Since simpler, non-neural network models often outperform neural them, we chose to concentrate on these simpler models.

During the literature exploration stage, two models stood out to us: RotatE 2.12 and QuatE 2.13. Both models are conceptually simple: RotatE uses one vector multiplication and QuatE is simply ComplEx 2.11 extended into quaternion space. Nonetheless, both have been shown to be highly effective, significantly outperforming several other state-of-the-art models.

Both RotatE and QuatE are rotational models. We want to investigate how combining rotation and translation affects a model's performance. To answer these question, we take RotatE and QuatE as starting point and constructed a number of experimental models based on them.

Since we are focusing on RotatE and QuatE, it is a natural choice to adopt their loss functions. Adaptive margin ranking loss (AMR) was shown to improve models' performances dramatically [35]. Because of this, we decided to include it as one of our loss functions. In summary, we have two core embedding models: RotatE 2.12 and QuatE 2.13 and three different loss functions: negative sampling used by RotatE 2.17, regularized logistic loss used by QuatE 2.19, and adaptive margin ranking loss 2.18. This is shown in Figure 3.2 in the dark blue box marked "Selected."

3.3 Extended models

As already discussed, one of our research questions is whether adding translation significantly improves a model's performance. To test this, we develop several extensions of the RotatE and QuatE models. These are listed in *Extend models* box in the workflow diagram. In total, we implement eight experimental models:

Complex	**Quaternion**
- TransRotatE	- TransQuatE
- halfTransRotatE	- halfTransQuatE
- halfTransRotatE-Conj	- halfTransQuatE-Conj
- biRotatE	- biQuatE

We divide these experimental models into two classes. The models in the left column are based on RotatE and embed a KG into the complex space. Models

in the right column are QuatE-based and embed into quaternion space. We refer to all the models except biRotatE and biQuatE as *hybrid*, since they use both translation and rotation. Models that start with "*half*" are also referred to as *simplified hybrid models*, or just *simplified models*.

All models represent entities as vectors in $\mathbb{C}^d$ for RotatE-based models and $\mathbb{Q}^d$ for QuatE-based models. The models differ, however, in how they represent relations. TransRotatE and TransQuatE embed each relation into three distinct vectors: *translation*, *head-rotator*, and *tail-rotator*; simplified models use two vectors per relation: *head-rotator* and *translation*; and biRotatE/biQuatE use two vectors per a relation: *head-rotator* and *tail-rotator*. For QuatE-based models, all of the relation embedding vectors are in quaternion space $\mathbb{Q}^d$. For RotatE-based models, however, only the translation vector is in complex space $\mathbb{C}^d$. Rotator vectors (*rotator-head* and *rotator-tail*) are vector in $\mathbb{R}^d$. We follow [11] and use Euler's formula 2.1 to construct a real and imaginary part of the rotator vectors.

TransRotatE and **TransQuatE** are defined using the following scoring function:

$$\phi(h, r, t) = -\|\mathbf{h} \circ \mathbf{r}_h + \mathbf{r}_{tr} - \bar{\mathbf{t}} \circ \mathbf{r}_t\| \tag{3.1}$$

where $\circ$ is the element-wise Hadamard product for complex vectors in TransRotatE or Hamilton product 2.2 for quaternion vectors in TransQuatE, $\bar{\mathbf{t}}$ is a complex conjugate of $\mathbf{t}$, $\mathbf{h}$ and $\mathbf{t}$ are embeddings of h and $t \in \mathcal{E}$, $\mathbf{r}_h$ is the head-rotator vector, $\mathbf{r}_t$ is the tail-rotator, and $\mathbf{r}_{tr}$ is the translation vector. Together, $\mathbf{r}_h, \mathbf{r}_t$, and $\mathbf{r}_{tr}$ are embeddings of $r \in \mathcal{R}$.

To test whether the extra rotation on the tail entity is useful, we propose two types simplified hybrid models: **halfTransRotatE** and **halfTransRotatE-Conj** for TransRotatE and **halfTransQuatE** and **halfTransQuatE-Conj** for TransQuatE. Both of these models rotate only the head entity before applying translation. Their scoring function is defined as:

$$\begin{gathered}\textbf{halfTransRotatE/halfTransQuatE}: \\ \phi(h, r, t) = -\|\mathbf{h} \circ \mathbf{r}_h + \mathbf{r}_{tr} - \mathbf{t}\|\end{gathered} \tag{3.2}$$

$$\begin{gathered}\textbf{halfTransRotatE} - \textbf{Conj/halfTransQuatE} - \textbf{Conj}: \\ \phi(h, r, t) = -\|\mathbf{h} \circ \mathbf{r}_h + \mathbf{r}_{tr} - \bar{\mathbf{t}}\|\end{gathered} \tag{3.3}$$

where $\bar{\mathbf{t}}$ is the complex conjugate of $\mathbf{t}$, $\mathbf{r}_h$ is the rotator vector for the head entity, and $\mathbf{r}_{tr}$ is the translation vector.

To test whether it is translation that really affects a model's performance, and not only the extra rotation on the tail entity, we further propose two purely rotational models **biRotatE** and **biQuatE**. The scoring function is as follows:

$$\phi(h, r, t) = -\|\mathbf{h} \circ \mathbf{r}_h - \bar{\mathbf{t}} \circ \mathbf{r}_t\| \tag{3.4}$$

where $\mathbf{r}_h$ is the head-rotator and $\mathbf{r}_t$ is the tail-rotator. This model rotates both the head and the tail entity, and compares the two, without using the translation.

3.4 Injecting rules

One of the biggest decisions in the scope of this work was whether to explicitly include logic rules and if so, which methods to use. Our first decision was in favor of including the rules. The original decision was to implement both RUGE method [25] and a method based on LogicENN [13]. However, since RUGE is already well-studied, we ultimately chose against it in favor of LogicENN-based approach.

As explained in Section 2.2.5, LogicENN is tailored to neural networks. We thus had to modify it to fit requirements of our models. We concentrate on equivalence, symmetry, implication, and inverse patterns and omit the more complex *transitivity* and *composition*.

Recall that LogicENN rule injection method is based on the fact that, given a logical rule, the probability of the consequent being true is always at least as high as the probability of the antecedent being true. For rules that use logical biconditional $\Leftrightarrow$, such as symmetry and equivalence, we require that both triples have identical scores. The regularization term used for these types of rules is computed as:

$$R_b = \|\phi(p) - \phi(q)\|_1 \tag{3.5}$$

where $\|\cdot\|_1$ is the L1 norm. This term is minimized when $\phi(p) \approx \phi(q)$, satisfying the condition that the triples in biconditional logical rules should have similar probability of being true.

For rules that use implication $\Rightarrow$, such as inverse and implication, we require the scores of the antecedent triple (i.e. the triple that implies) to be lower than the scores of the consequent triple (i.e., the implied triple.) For a rule $p \Rightarrow q$, the corresponding regularization term is then defined as:

$$R_c = \text{ReLU}(\phi(p) - \phi(q)) \tag{3.6}$$

where ReLU is the rectified linear unit function defined as $\text{ReLU}(x) = \max(0, x)$. Equation 3.6 is minimized when the score of the consequent q is higher than that of the antecedent p.

Injection of the rules into the learning process follows LogicENN: first, the rules are grounded; second, only the valid groundings are kept. A grounding is considered valid if the antecedent triple is observed in the KG, and the consequent one is not. The values for the valid groundings are then computed using equations 3.6 or 3.5 and added to the loss function as a regularization term [13].

Chapter 4

Implementation

This chapter describes architecture and implementation details of MultEmbed. In Section 4.1, we discuss the general architecture of the code: the control flow, settings, and so on. Then, in Section 4.2, we provide detailed instructions on how to run MultEmbed. In the following Section *Models*, we discuss the particularities of the models' implementation. Section 4.4 explains how rule injection is implemented and how it can be invoked. Lastly, we discuss available loss functions in Section 4.5. The goal of this chapter is to describe the implementation of the project in enough detail, so that it can be easily reproduced by an average masters student in computer science.

4.1 General architecture

Implementation of MultEmbed is based on that of RotatE.[1] The implementation is done in Python, using PyTorch module.[2] The architecture of our suite is illustrated in Fig. 4.1 and consists of an embedding model class `KGEModel`, the data processing classes `TrainDataset` and `TestDataset`, as well as the main `run.py` program that instantiates the classes and guides the training and testing processes.

At the runtime, the user can specify different hyperparameter settings. These are summarized in Table 4.2. Flags `-model`, `-data_path`, `-n`, `-d`, `-adv`, `-a`, `-b`, `-r`, and `-lr` are available in RotatE implementation. MultEmbed further allows the

[1] Original implementation of RotatE can be found at `https://github.com/DeepGraphLearning/KnowledgeGraphEmbedding`

[2] `https://pytorch.org/`

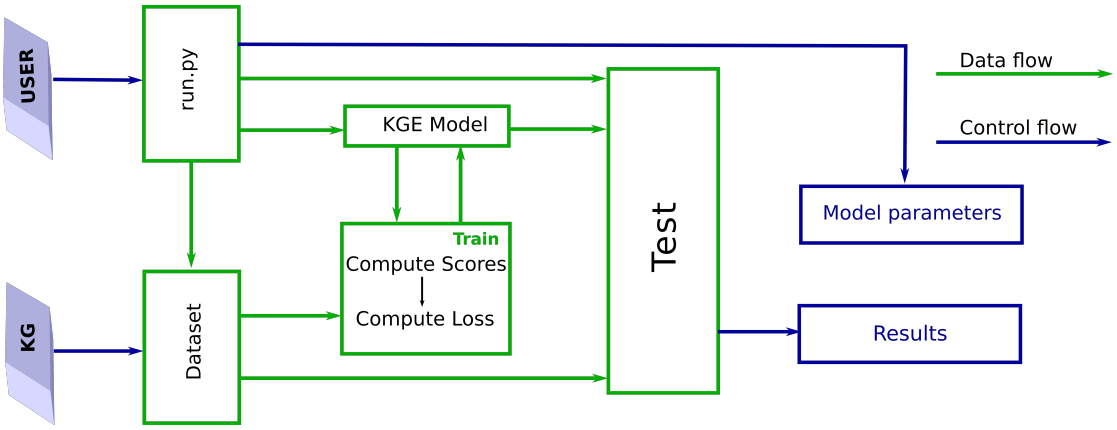


FIGURE 4.1: Architecture of KGE embedding suite **MultEmbed**. Green arrows represent the control flow in the software. Blue arrows represent data flow. Blue boxes represent either input or output of the program.

choice between different loss functions, additional L2 regularization, and a choice between Adam and AdaGrad optimizers.

The settings are fed to the `run.py` - the main control center of the suite. For clarity, we will refer to it as `Run`. Here, we adopt the framework of RotatE, with only slight modifications. Using these settings, `Run` first instantiates the data processing classes to create three datasets: *train*, *test*, and *validation*. The dataset classes are borrowed from RotatE and are responsible for generating negative samples and cycling through data batches. For each positive triple (h, r, t) in the batch, each dataset generates negative triples by iteratively corrupting either the head or the tail entity. In other words, dataset classes iterate between generating a batch of false triples $\{(h', r, t)|h' \in \mathcal{E},\ h' \neq h\}$ and $\{(h, r, t')|t' \in \mathcal{E},\ t' \neq t\}$. Entities h'/t' are chosen at random under the condition that the corrupted triple $(h', r, t)/(h, r, t') \notin \mathcal{G}$.

After `Run` calls `Dataset` class, it instantiates the `KGEModel`. Here, we follow in the footsteps of RotatE, and define one general embedding model `KGEModel` and implement different scoring functions on top of it. All of the models supported by RotatE share the same entity and relation representations: one vector per entity and one vector per each relation. The models that we implement, however, have differing representations. For instance, TransRotatE represents an entity with one vector, and a relation with three different ones: a head rotator, a tail rotator, and a translation vectors. Because of this, we modify the `KGEModel` as shown in Fig. 4.2. Namely, we add three instance variables: `use_translation`, `use_head_rotation`,

```
        if self.model_name in ['biRotatE', 'biQuatE']:
        self.use_translation = False
        else: self.use_translation = True

        if self.model_name in ['TransRotatE', 'TransQuatE', 'biRotatE', 'biQuatE']:
        self.use_tail_rotation = True
        else: self.use_tail_rotation = False

        if self.model_name not in ['QuatE', 'RotatE']:
        self.use_head_rotation = True
        else: self.use_head_rotation = False
```

FIGURE 4.2: General embedding model: different embedding models have different entity and relation representations.

```
        if self.use_translation:
        self.relation_embedding = nn.Parameter(
        torch.zeros(self.nrelations, self.relation_dim))
        self.initialize(self.relation_embedding, nrelations, hidden_dim)

        if self.use_head_rotation:
        self.rotator_head = nn.Parameter(
        torch.zeros(nrelations, hidden_dim))
        self.initialize(self.rotator_head, nrelations, hidden_dim)

        if self.use_tail_rotation:
        self.rotator_tail = nn.Parameter(
        torch.zeros(nrelations, hidden_dim))
        self.initialize(self.rotator_tail, nrelations, hidden_dim)
```

FIGURE 4.3: Model parameters are defined dynamically, depending on the representation used by the model.

and `use_tail_rotation`. Setting one to `True` creates a corresponding embedding model parameter, as shown in Fig. 4.3.

After the model and the dataset classes are instantiated, `Run` starts the training process. Instead of running the training for a specified number of epochs, the process runs for a specified number of training *steps* [11]. At each training step, `TrainDataset` is called to select a batch of training data, consisting of both positive and negative samples. These are then fed to the KGEModel class, which computes scores for the training batch, the loss, and finally, updates the model parameters.

When the training is complete, final model parameters are written to files and the testing process begins. Testing is adopted from RotatE and is discussed in detail in the following chapter.

Flag	Explanation	Default
`--model`	**KGE model**	—
`--data_path`	**KG to embed**	—
`-save`	**ID under which to save the model**	—
`--save_checkpoint_steps`	# steps after which to save a checkpoint	10,000
`--valid_steps`	# steps before testing on validation dataset	10,000
`--log_steps`	# steps before logging data	100
`--init`	Loads previously trained model	—
`--do_train`	Activates training mode	—
`--do_valid`	Activates testing on the validation dataset	—
`--do_test`	Activates testing on the test dataset	—
`--do_grid`	Activates grid search	—

TABLE 4.1: MultEmbed options and modes of operation. Options in bold must be specified at runtime. Additionally, at least one of the `--do`-options must be specified. If `--do_train` is not specified, then `-init` must be used, followed by the path to the model to be loaded.

4.2 Running MultEmbed

MultEmbed can be ran by simply running `python run.py` followed by appropriate flags. The user must choose at least one of the tasks for MultEmbed: `--do_train` for training a model, `--do_test` or `--do_valid` for testing the model on test or validation dataset respectively, or `--do_grid` for running grid search on hyperparameters. A new model can be trained from scratch or loaded by specifying `--init` followed by the path to it. A pre-trained model must be given, unless `--do_train` is used. The model itself, path to the dataset, as well as the ID under which to save the trained model (or ID of the model to be loaded) must be specified. Additionally, the user can set the following options:

- To save a partially trained model every N training steps: `--save_checkpoint_steps` N
- To output log information (e.g. current loss value) every L steps: `--log_steps` L.
- To test the model on a validation dataset every D training steps: `--do_valid --valid_steps` D.

Hyperparameters that can be specified directly by the user include:

- the loss function (choice between *adaptive* for adaptive margin ranking loss 2.18, *logistic* for regularized logistic loss 2.19, or *ns* for negative sampling loss 2.17);
- the number of negative samples per one positive triple in the batch;
- embedding dimension;
- fixed margin γ used in loss functions;
- training batch size;
- optimizer (choice between AdaGrad (`ada`) or `adam`);
- number of training steps to use.
- regularization and its coefficient (choice between L2 or L3). For L2, use `--l2-r` followed by the coefficient, and for L3: `-r` + the coefficient. If neither is specified, no regularization is used.
- adversarial sampling, as explained in Section 2.2.4. Flag `-adv` activates adversarial sampling and flag `-a` followed by a number $\in [0, 1]$ specifies the sampling temperature α in equation 2.16.

Tables 4.1 and 4.2 summarize these options and specify their default values, where applicable.

For instance, running the command

```
python codes/run.py --cuda --do_train --do_valid --valid_steps 10000
--do_test --max_steps 100000 --model TransRotatE --data_path data/FB15k
-n 20 -b 512 -d 200 -g 24.0 -adv -a 1.0 -lr 0.1 --opt ada -save 0
```

runs the code on a GPU device (`--cuda`), trains the model *TransRotatE* (`--model TransRotatE`) on *FB15k* (`--data_path data/FB15k`) for 100,000 training steps[3] (`--max_steps 100000`), tests it on the validation dataset every 10,000 steps (`--do_valid --valid_steps 10000`), and tests it on the test dataset after completing the training (`--do_test --test_batch_size 16`). It generates 20 negative triples per each positive triple (`-n 20`), uses batch size of 512 (`-b 512`), embedding dimension of

[3]Training steps are different from training epochs. One training step trains the model on one batch of data only.

Flag	Explanation	Default
`--loss`	loss function	adaptive
`-n`	# negative samples per one positive triple	50
`-d`	embedding dimension	500
`-adv`	activates adversarial sampling	—
`-a`	adversarial temperature for negative sampling	1.0
`-b`	batch size for training	1024
`-g`	gamma	12.0
`-r`	weight for L3 regularization	0
`--l2-r`	weight for L2 regularization	0
`-lr`	learning rate	.0001
`--opt`	optimizer	ada
`--max_steps`	# training steps	100,000
`--test_batch_size`	batch size for test dataset	4

TABLE 4.2: Hyperparameters that user can specify at runtime. When the flag is not used, the option in the **Default** column is used (— indicates that no default value is given.)

200 (`-d 200`), fixed margin $\gamma = 24.0$ (`-g 24.0`), adversarial sampling defined in equation 2.16 with temperature $\alpha = 1.0$ (`-adv -a 1.0`), and learns the model using AdaGrad (`--opt ada`) with the learning rate of 0.1 (`-lr 0.1`). The final model is saved under `models/TransRotatE_FB15k_0`, where "0" is specified by `-save 0`.

To simplify running of the code, the user can make use of the helper bash script `run.sh`. It offers four modes: `train`, `valid`, `test`, and `grid`. As the name suggest, `train` is used to train the model, `valid` and `test` test the model on validation and test datasets respectively, and `grid` is used to simplify grid search on hyperparameters. The first three are available through RotatE implementation, while the last one is our own addition.

To train the program using `run.sh` script, the user needs to use `train` option of run.sh and pass the following command-line arguments in this order:
2: model name, 3: dataset, 4: GPU ID, 5: save ID, 6: batch size, 7: number of negative samples, 8: embedding dimension, 9: value of γ, 10: adversarial temperature α, 11: learning rate, 12: number of training steps, and 13: test batch size, plus any other flags that the user wishes to specify. The first command line argument is used to specify the mode of `run.sh`.

For example, the following command

```
bash run.sh train TransRotatE FB15k 1 0 512 20 200 24 1.0 0.1 100000 16 --opt ada
```

	train	grid	valid	test
2		⟵ model name ⟶		
3		⟵ dataset ⟶		
4		⟵ GPU ID ⟶		
5		⟵ save ID ⟶		
6	⟵ batch size ⟶		—	—
7	# neg. samples	adv. temp. α	—	—
8	embedding dim.	learning rate	—	—
9	γ	test batch size	—	—
10	adv. temp. α	—	—	—
11	learning rate	—	—	—
12	# training steps	—	—	—
13	test batch size	—	—	—

TABLE 4.3: Command line argument position for different modes of the `run.sh` script. Argument at position 1 should be the mode itself (i.e. either `train`, `grid`, `test` or `valid`). Arguments 2 through 5 are the same for all 4 mode types. Argument at position 6 is the same for `train` and `grid` modes. — indicates that no argument at this position is necessary.

```
1    N_NEGS_LIST  = [20, 50]                # number of negative samples
2    N_STEPS_LIST = [150000, 200000]        # number of training steps
3    LOSSES       = ['adaptive', 'nsl']     # loss functions
4    DIMENSIONS   = [200, 500]              # embedding dimensions
5    OPTIMIZER    = ['ada']
```

FIGURE 4.4: Grid testing parameters. Fill in the appropriate lists to test on each value. For instance, line 4. DIMENSIONS = [200, 500] tests two models, one with dimension 200 and one with dimension 500.

is equivalent to the command at the start of this section.

For `valid` and `test`, only the arguments 2 through 5 are needed (i.e. model name, dataset, GPU device ID, and save ID). These two options do not train a model, and instead load the model saved under "MODEL"_"DATASET"_"SAVE_ID" and test it.

The `grid` mode of the run script requires slightly different command line argument order: 2 through 5 are the same as for testing (model, dataset, GPU device ID, and save ID), 6: training batch size, 7: adversarial temperature α, 8: learning rate, and 9: test batch size. Other command line arguments can be passed to MultEmbed by appending them at the end of the script command.

The ranges of hyperparameters over which to perform grid search are specified in the class `GridTesting` in the file `grid_testing.py` as shown in Fig. 4.4. All of the values must be specified as lists, even if it is a list of one element only, as in

line 5. If any of the values in the hyperparameter lists are left empty, then the command line argument (including default settings) is used. For instance,

```
bash run.sh grid TransRotatE FB15k 0 0 512 0.1 24 .1 16 --loss adaptive
```

runs grid search for TransRotatE on FB15k, on GPU device 0, saves the resulting models under ID 0, and uses model TransRotatE and adaptive margin ranking loss unless the hyperparameter list `LOSSES` is non-empty.

4.3 Models

In previous chapter, in Section 3.3, we discussed details of the models supported by MultEmbed. Our extended models embed KGs into either complex or quaternion space. We follow [34] as well as others (e.g. [11], [12]) and treat embedding dimension as dimension of each part of a complex number. For instance, if an embedding is a vector $\mathbf{v} \in \mathbb{C}^d$, then it is represented by two $d-$dimensional vectors: one for the real part of $\mathbf{v}$, and one for the imaginary. Similarly with quaternion numbers: for any embedding $\mathbf{v} = a+b\mathbf{i}+c\mathbf{j}+d\mathbf{k}$, a, b, c, and d are each represented by a distinct $d-$dimensional vector.

Not every model parameter (i.e. entity or relation embedding) is represented by a complex number. As explained in Section 3.3, RotatE-based models embed rotator vectors (i.e. head- and tail-rotators) in $\mathbb{R}^d$ and apply Euler's function 2.1 to construct the real and imaginary parts. Table 4.4 lists MultEmbed supported models, their score function, as well as relation representation.

Although RotatE- and QuatE-based models are conceptually similar, their implementations differ due to the variation in relation representation and complex vs quaternion product. Figure 4.5 shows implementation of TransRotatE. Before computing the score, embeddings of the head, tail, and translation must be split into their real and imaginary parts. This is done in lines 2, 3, and 10.

To split head-rotator $\mathbf{r}_h$ and tail-rotator $\mathbf{r}_t$ vectors into their real and imaginary parts, Euler's formula is applied:

$$\begin{aligned} Re(\mathbf{r}_h) &= \cos\left(\frac{\mathbf{r}_h}{c}\right) \\ Im(\mathbf{r}_h) &= \sin\left(\frac{\mathbf{r}_h}{c}\right) \end{aligned} \tag{4.1}$$

Model	Scoring function ϕ	Relation	# Parameters
RotatE	$-\|\mathbf{h} \circ \mathbf{r} - \mathbf{t}\|$	d	d
TransRotatE	$-\|\mathbf{h} \circ \mathbf{r}_h + \mathbf{r}_{tr} - \bar{\mathbf{t}} \circ \mathbf{r}_t\|$	head-rotator: d tail-rotator: d translation: $2d$	$4d$
halfTransRotatE	$-\|\mathbf{h} \circ \mathbf{r}_h + \mathbf{r}_{tr} - \mathbf{t}\|$	head-rotator: d translation: $2d$	$3d$
biRotatE	$-\|\mathbf{h} \circ \mathbf{r}_h - \mathbf{t} \circ \mathbf{r}_t\|$	head-rotator: d tail-rotator: d	$2d$
TransQuatE	$-\|\mathbf{h} \otimes \mathbf{r}_h + \mathbf{r}_{tr} - \bar{\mathbf{t}} \otimes \mathbf{r}_t\|$	head-rotator: $4d$ tail-rotator: $4d$ translation: $4d$	$12d$
halfTransQuatE	$-\|\mathbf{h} \otimes \mathbf{r}_h + \mathbf{r}_{tr} - \mathbf{t}\|$	head-rotator: $4d$ translation: $4d$	$8d$
biQuatE	$-\|\mathbf{h} \otimes \mathbf{r}_h - \mathbf{t} \otimes \mathbf{r}_t\|$	head-rotator: $4d$ translation: $4d$	$8d$

TABLE 4.4: Summary of MultEmbed supported models: scoring function, relation representation, and the number of parameters used to represent a relation. Hyperparameter d is the user-specified embedding dimension. For space and simplicity, we omit halfTransRotatE-Conj and halfTransQuatE-Conj: they have the same relation representation as halfTransRotatE and halfTransQuatE, respectively. The scoring function is, again, nearly identical, except for complex conjugate of $\mathbf{t}$ is used.

where $c = \frac{max}{\pi}$ is a normalization constant that ensures that $Re(\mathbf{r}_h), Im(\mathbf{r}_h) \in [-\pi, \pi]$ and max is the upper limit on the initial value of any element of $\mathbf{r}_h$. This is done by the function `self.extract_relations()`. This function, illustrated in Figure 4.6, takes a variable number of rotator vectors as input, applies Euler's formula 4.1 to each of them, and returns the real and imaginary parts.

Real and imaginary parts of the head and tail entities are then rotated in lines 11–15 to get $Re(\mathbf{h} \circ \mathbf{r}_h)$, $Im(\mathbf{h} \circ \mathbf{r}_h)$, $Re(\mathbf{t} \circ \mathbf{r}_t)$, and $Im(\mathbf{t} \circ \mathbf{r}_t)$. Note the addition and subtraction in lines 14 and 15 - it is reversed, in order to compute complex conjugate $\bar{\mathbf{t}} \circ \mathbf{r}_t$. The score is then computed using these rotated entities in lines 17–21.

For TransQuatE, however, we store each part of the embeddings explicitly. That is, all parameters share the same length of $4d$ and to get the real, $\mathbf{i}$, $\mathbf{j}$, and $\mathbf{k}$ parts,

```
1         def TransRotatE(self, head, rotator_head, rotator_tail, translation, tail):
2         re_head, im_head = torch.chunk(head, 2, dim=2)
3         re_tail, im_tail = torch.chunk(tail, 2, dim=2)
4
5         # multipliers
6         re_relation_head, im_relation_head, re_relation_tail, im_relation_tail =
7         self.extract_relations(rotator_head, rotator_tail)
8
9         re_translation, im_translation = torch.chunk(translation, 2, dim = 2)
10
11        re_rotated_head = re_relation_head * re_head - im_relation_head * im_head
12        im_rotated_head = re_relation_head * im_head + im_relation_head * re_head
13
14        re_rotated_tail = re_relation_tail * re_tail + im_relation_tail * im_tail
15        im_rotated_tail = re_relation_tail * im_tail - im_relation_tail * re_tail
16
17        re_score = re_rotated_head + re_translation - re_rotated_tail
18        im_score = im_rotated_head + im_translation - im_rotated_tail
19        score = torch.stack([re_score, im_score], dim = 0)
20        score = torch.norm(score, p = 1, dim = 0)
21        score = score.sum(dim = 2)
22        return score
```

FIGURE 4.5: TransRotatE implementation.

```
1         def extract_relations(self, *args):
2         pi = 3.14159265358979323846
3
4         split_relations = []
5         for relation in args:
6         phase_relation = relation/(self.embedding_range.item()/pi)
7         re_relation = torch.cos(phase_relation)
8         im_relation = torch.sin(phase_relation)
9         split_relations.extend([re_relation, im_relation])
10        return split_relations
```

FIGURE 4.6: Helper function for RotatE-based models: implementation of4.1.

we split the parameter into four parts. For the relation parameter, this is done by the call to `self.normalize_quaternion()`, illustrated in Figure 4.7.

Another important helper function for QuatE-based models is the `self.rotate_quaternion()`, shown in Figure 4.8. It takes two quaternion vectors as input: the embedding of the entity to be rotated and the rotator, and returns the Hamilton product 2.2 of the entity embedding and the normalized rotator embedding.

With these helper functions, TransQuatE implementation, shown in Figure 4.9, is very similar to that of TransRotatE. Note that in lines 13–15, we add the rotated

```python
def normalize_quaternion(self, tensor):
t, ti, tj, tk = torch.chunk(tensor, 4, dim = 2)
denom = torch.sqrt(t**2 + ti**2 + tj**2 + tk**2)
t = t/denom
ti = ti/denom
tj = tj/denom
tk = tk/denom
return t, ti, tj, tk
```

FIGURE 4.7: Function that normalizes and splits the rotator parameter into four parts for QuatE and TransQuatE.

```python
def rotate_quaternion(self, entity, rotator):
ent_re, ent_i, ent_j, ent_k = torch.chunk(entity, 4, dim = 2)
rot_re, rot_i, rot_j, rot_k = self.normalize_quaternion(rotator)

rotated_real = ent_re*rot_re  - ent_i*rot_i - ent_j*rot_j - ent_k*rot_k
rotated_i    = ent_re*rot_i + ent_i*rot_re + ent_j*rot_k - ent_k*rot_j
rotated_j    = ent_re*rot_j - ent_i*rot_k + ent_j*rot_re + ent_k*rot_i
rotated_k    = ent_re*rot_k + ent_i*rot_j - ent_j*rot_i + ent_k*rot_re

return rotated_real, rotated_i, rotated_j, rotated_k
```

FIGURE 4.8: Helper function. It takes an entity and a relation embedding as input and returns rotated entity.

```python
def TransQuatE(self, head, rotator_head, rotator_tail, translation, tail):
# rotate head
rotated_head_re, rotated_head_i, rotated_head_j, rotated_head_k = \
self.rotate_quaternion(head, rotator_head)
# rotate tail
rotated_tail_re, rotated_tail_i, rotated_tail_j, rotated_tail_k = \
self.rotate_quaternion(tail, rotator_tail)

# translation
tran_re, tran_i, tran_j, tran_k = torch.chunk(translation, 4, dim = 2)

score_r = rotated_head_re + tran_re - rotated_tail_re
score_i = rotated_head_i  + tran_i  + rotated_tail_i
score_j = rotated_head_j  + tran_j  + rotated_tail_j
score_k = rotated_head_k  + tran_k  + rotated_tail_k

score = torch.stack([score_re, score_i, score_j, score_k], dim = 0)
score = torch.norm(dim = 0)
return score.sum(dim = 2)
```

FIGURE 4.9: Implementation of TransQuatE scoring function.

tail entity instead of subtracting it as in line 12. Again, the sign is reversed because we want to take the conjugate of the tail embedding.

It should be easy to extrapolate implementation of the simplified models. All that needs to be done is removing irrelevant parts from TransRotatE or TransQuatE implementation. For instance, to implement biRotatE, one only needs to delete line 9 and remove `+ re_translation` and `+ im_translation` from lines 17 and 18 of Figure 4.5.

4.4 Rule injection

As covered in Section 3.4, MultEmbed allows explicit injection of rules. The following flags activate this functionality:

> `--inv`, `--impl`, `--sym`, `--eq` for inverse, implementation, symmetry, and equivalence rules respectively, and
>
> `--inject` to actually inject the rule.

To inject rules, at least one of the four flags in the upper row above must be specified. This tells MultEmbed which rule to inject. If any of these are specified, MultEmbed computes the corresponding regularization terms using Equations 3.6 for inverse and implication rules and 3.5 for symmetry and equivalence. Flag `--inject` tells MultEmbed to actually inject the rule.

Activating the flag `--inject` without specifying at least a single rule to inject leads MultEmbed to exit with an error message.

In order to make use of this option, the user must include a file with appropriate groundings in the dataset folder. Files containing groundings use the following naming convention: `grounding_`, followed by rule type, and `.txt`, for instance, `groundings_inverse.txt`. The groundings in the file must be listed one per line, listing entities first, followed by relations:

$$\begin{aligned}&\text{ent}_{a1}, \text{ent}_{a2}, \text{rel}_{a1}, \text{rel}_{a2}\\&\text{ent}_{b1}, \text{ent}_{b2}, \text{rel}_{b1}, \text{rel}_{b2}\\&\dots\\&\dots\end{aligned}$$

The first and the second entity always represent the head and tail of the antecedent and the first relation - the relation between them. Thus, the first line in the above example corresponds to the antecedent (ent_{a1}, rel_{a1}, ent_{a2}). The consequent is constructed automatically depending on what type of rule it is: for instance, if it is an inverse rule, we get the following grounding: (ent_{a1}, rel_{a1}, ent_{a2}) $\Rightarrow$ (ent_{a2}, rel_2, ent_{a1}).

4.5 Loss functions

We adapt the negative sampling technique from RotatE: as explained in Section 2.2.4, instead of sampling according to a distribution, RotatE samples uniformly and uses the distribution as weights. We keep the implementation of the negative sampling (NS) loss function used by RotatE. We further implement two additional loss functions.

The adaptive margin ranking (AMR) loss is implemented similarly to NS loss with four important differences. First, the logsigmoid in NS loss is changed to ReLU. Second, margin squared is added to both the positive and negative scores, e.g.:

```
negative_score = σ(negative_score * args.adv_temperature,dim = 1)
.detach()* ReLU(negative_score + self.margin**2)).sum(dim = 1)
```
(4.2)

where σ is the softmax function $\sigma(\mathbf{z})_i = \frac{\exp(\mathbf{z}_i)}{\sum_K^{j=1} \exp(\mathbf{z}_j)}$ [43] and that can be computed using `torch.nn.functional.softmax`, $args.adv\_tempterature$ is the user-specified adversarial temperature α from the negative sampling distribution equation 2.16, `.sum(·)` is the summation function used by Pytorch, and `.detach(·)` is a Pytorch function that prevents use of the detached part of an equation in the backpropagation.

The third difference from RotatE is how the final loss value is computed. While NS loss simply takes an average of both negative and positive losses, AMR uses equation 2.18. Finally, NS loss sets

```
positive_score = λ−positive_score
negative_score = negative_score−λ
```
(4.3)

while in AMR loss, we reverse the signs:

$$\begin{aligned} \texttt{positive\_score} &= \texttt{positive\_score} - \lambda \\ \texttt{negative\_score} &= \lambda - \texttt{negative\_score} \end{aligned} \tag{4.4}$$

Implementation of the third loss function, regularized logistic (RL) loss defined in equation 2.19, is as follows:

```
1 negative_loss = torch.mean(self.criterion(-negative_score))
2 positive_loss = torch.mean(self.criterion(positive_score))
3 loss = (positive_loss + negative_loss)/2
```

where `self.criterion` is the softplus function defined as $f(x) = \log(1 + e^x)$ [44]. We negate `negative_score`, because RL multiplies scores by a label Y_{hrt} of the corresponding triple, which for negative triples is -1.

Chapter 5

Evaluation

This chapter focuses on the evaluation of the models supported by MultEmbed. We start with describing the datasets in Section 5.1. Section 5.2 describes the metrics that we use in evaluating the datasets. Section 5.3 discusses results of our evaluations. We discuss how different models perform on each dataset and how well they learn relational patterns. Finally, we discuss injecting rules and how that changes model's performances.

5.1 Datasets

We evaluate our model on standard KG datasets: FB15k [27], WN18 [27], FB15k-237 [10], and WN18rr [41]. FB15k is a subset of Freebase [3], which is a large-scale KG of general knowledge. It includes facts about such diverse domains as persons, movies, sports, etc. In total, Freebase contains about 50 million entities and close to 3 billion triples [19]. FB15k limits the number of entities to just under 15,000.

WN18 is a subset of WordNet [5] - a large-scale KG that stores lexical information about the English language. It organizes words (nouns, verbs, adjectives, and adverbs) into sets of synonyms called synsets. Relations are defined either between these synsets or between individual words. In total, there are about 150,000 unique words in WordNet, but some of them appear in more than one synset (if the same word is used to express different concepts, for example, word *bat*). WN18 is its subset, with 18 unique relations.

It has been observed by Toutanova and Chen in [10] that nearly 81% of all test triples (h, r, t) in FB15k follow directly from a triple (h, r', t) or (t, r', h) in the training dataset. For WN18, this percentage is 94%. In other words, the two most important relational patterns in FB15k and WN18 are symmetry/asymmetry and inverse. As a result, evaluating a model on these datasets gives only limited information about its performance - a model that learns to recognize inverse relations, rather than learning the complete KG, can still score highly on these datasets [13], [41]. To mitigate this, two further datasets were proposed - FB15k-237 and WN18rr. They are subsets of FB15k and WN18, respectively, with inverse relations removed. The most important relational patterns for these datasets are symmetry/asymmetry and composition. Table 5.1 summarizes basic information about the datasets.

	FB15k	**WN18**	**FB15k-237**	**WN18rr**
# entities	14,951	40943	14,541	40,943
# relations	1,345	18	237	11
# train triples	483,142	141,442	272,115	86,835
# valid triples	50,000	5,000	17,535	3,034
# test triples	59,071	5,000	20,466	3,134

TABLE 5.1: Dataset statistics. It shows number of unique entities and relations for each dataset, as well as the sizes of training, validation, and test sets (in number of triples).

5.2 Evaluation metrics

We evaluate our models on the link prediction problem. The task is to predict whether a relation is likely to exist between two given entities, or, in other words, whether some previously unobserved triple is likely to be true. This task is particularly important in KGE research, because KGs are highly incomplete. The amount of knowledge is simply too large to capture in a KG. Therefore, KG completion, through predicting previously unobserved triples is one of the most important and common applications of KGE. To test our models on this task, we use the following procedure:

1. A set of candidate triples is constructed: for each positive triple (h, r, t) in the test set, the head or tail entity are corrupted at random, constructing either the set $\{(h', r, t)|h' \in \mathcal{E}, h' \neq h\}$ or $\{(h, r, t')|t' \in \mathcal{E}, t' \neq t\}$.

2. True triples are filtered out from the candidate set: a candidate triple is removed if it appears in either test, training, or validation sets. This is referred to as *filtered setting* and results in a set of only false candidate triples (under the closed world assumption) [27].

3. The remaining candidate triples are then ranked against the original test triple: the scores of all the test and candidate triples are computed and then sorted in ascending order.

We compute the standard evaluation metrics for embedding models: mean rank (MR), mean reciprocal rank (MRR), and Hit at top N (Hit@N) for N = 1, 3, and 10 [9]. A rank of a true triple (h, r, t) against its candidate set is its position in the sorted list obtained at step 3, or more formally, it is defined as:

$$\text{rank}(h, r, t) = 1 + \sum_{(h', r, t) \notin T} \boldsymbol{I}\left[\phi(h', r, t) > \phi(h, r, t)\right] \tag{5.1}$$

if candidate set was constructed by corrupting the head entity, or

$$\text{rank}(h, r, t) = 1 + \sum_{(h, r, t') \notin T} \boldsymbol{I}\left[\phi(h, r, t') > \phi(h, r, t)\right] \tag{5.2}$$

if the tail entity was corrupted. $\boldsymbol{I}$ is the indicator function that returns 1 when the condition holds and 0 otherwise and T is the set of all (true) test triples.

MR is the average rank of all the correct test triples and is computed as follows:

$$\text{MR} = \frac{1}{|T|} \sum_{(h,r,t) \in T} \text{rank}(h, r, t) \tag{5.3}$$

MRR is the average inverse rank of all correct triples, defined as:

$$\text{MRR} = \sum_{(h,r,t) \in T} \frac{1}{\text{rank}(h, r, t)} \tag{5.4}$$

where $(h, r, t) \in T$ is the set of all (positive) test triples. Lastly, Hit@N is the percentage of the positive test triples whose rank is N or less:

$$\text{Hit@N} = \frac{1}{|T|} \sum_{(h,r,t) \in T} \boldsymbol{I}(\text{rank}(h, r, t) \leq \text{N}) \tag{5.5}$$

where $\boldsymbol{I}$ is the indicator function. We want a model that scores highly on MRR and Hit@N and low on MR.

5.3 Results

Hyperparameters were tuned on the validation set, using grid search over the following ranges:
embedding dimension $d \in \{100, 200, 500\}$, batch size $b \in \{512, 1024\}$, self-adversarial sampling temperature $\alpha \in \{0.5, 1.0\}$, fixed margin $\gamma \in \{6, 8, 10, 12, 24\}$, loss functions $\mathcal{L} \in \{\text{NS, AMR, RL}\}$, optimizer $opt \in \{\text{Adam, AdaGrad}\}$, learning rate $lr \in \{0.0001,\ 0.001,\ 0.01,\ 0.1,\ 0.3\}$. Best hyperparameter settings are KG- and model-dependent, and are listed in Appendix A.2.

We compare our models to several state-of-the-art models such as RotatE, TransE, ComplEx, and so on. To get a quick overview of how models perform on different scores relative to each other, we refer the reader to Appendix A.3, which includes plots of Hit@N and MRR scores. For QuatE, we use implementation published in [12] and report scores using L2 regularization and without type constraints.

5.3.1 WN18

Results for WN18 dataset are summarized in Table 5.2. It is a dataset that is relatively easy for models to learn. This is explained by the test leakage described above. There is also not much variation between model's performances on this dataset. For instance, all of the models in the MultEmbed suite, as well as RotatE, TuckER, and QuatE, score very similarly on Hit@3 and Hit@10. TransQuatE, halfTransQuatE, and halfTransQuatE-Conj score identically or near-identically on MRR and Hit@N for all N. Nonetheless, TuckER appears to be the best performing model.

	MR	**MRR**	**Hit@1**	**Hit@3**	**Hit@10**
TransE*	–	.495	.113	.888	.943
DistMult*	655	.797	–	–	.946
ComplEx*	–	.941	.936	.945	.947
RotatE*	309	.949	.944	.952	.959
QuatE	349	.942	.927	.952	.960
TuckER	–	**.953**	**.949**	**.955**	.958
TransRotatE	181	.945	.933	.954	**.962**
halfTransRotatE	176	.946	.936	.954	.961
halfTransRotatE-Conj	170	.948	.943	.952	.958
biRotatE	224	.944	.933	.954	**.962**
TransQuatE	174	.950	.944	.953	.960
halfTransQuatE	**164**	.950	.944	.953	.959
halfTransQuatE-Conj	175	.950	.945	.953	.960
biQuatE	211	.947	.939	.954	**.962**

TABLE 5.2: **Evaluation 1.** Results of models evaluated on WN18. Results for models marked by $\star$ are taken from [11] and results for TuckER are from [45].

All of our models perform significantly better on MR scores than the state-of-the-art reference models. The models with the lowest mean rank are those that include translation as well as rotation on a head entity only. Models that rely on rotation only, such as RotatE, biRotatE, QuatE and biQuatE, score significantly worse. Surprisingly, QuatE, with its more complex 2-dimensional rotation scores particularly badly, with only DistMult and ConvE scoring worse than it.

TransRotatE scores almost identical to biRotatE (except on MR), indicating that translation in TransRotatE does not improve the model significantly. Moving the models into quaternion space leads to improvement in performance, albeit a

small one. This is particularly apparent on MR, MRR and Hit@1. TransQuatE, halfTransQuatE, and halfTransQuatE-Conj perform nearly identically, indicating that in quaternion space, the extra rotation on the tail entity adds little to a model's performance. Rotation in quaternion space is more complex than rotation in $\mathbb{C}$, which explains why for these models, it is sufficient to rotate one entity only. BiQuatE outperforms biRotatE (except for on Hit@3 and Hit@10, where the two models perform identically), indicating again that increasing complexity of rotation improves performance. This further corroborates our observation that rotation plays an important role on this dataset.

Surprisingly, RotatE does not fare well on this dataset - on none of the metrics is RotatE even the second best performing model. For instance, its performance is at least matched, and often exceeded, by TransQuatE, halfTransQuatE, and halfTransQuatE-Conj (albeit the difference is not very large.)

5.3.2 WN18rr

WN18rr, a subset of WN18 without the inverse relation [11], is much harder for models to capture. Table 5.3 gives an overview of the results on this dataset. Interestingly, RotatE's performance relative to other models improves - it has the highest score on MRR, and second highest scores on Hit@1 and 3. On MRR and Hit@10, it even significantly outperforms TuckER - the best performing model on WN18.

On this dataset, neither biRotatE nor biQuatE match performance of TransRotatE: they perform worse on all metrics but MR. In contrast to WN18, halfTransRotatE clearly outperforms TransRotatE. Overfitting of the TransRotatE appears to play some role in this. Thus, retraining TransRotatE on the embedding dimension of 350 instead of 500 produces the following results:
MR = 2900; MRR = 0.463; Hit@1 = 0.401, Hit@3 = 0.492; Hit@10 = 0.583
While this is an improvement, it still falls short of halfTransRotatE. Thus, the superior performance of halfTransRotatE cannot be explained by overfitting of TransRotatE alone. Instead, it indicates that the second rotation on the tail entity is superfluous and that translation is particularly important to capturing this dataset.

	MR	MRR	Hit@1	Hit@3	Hit@10
TransE*	3384	.226	–	–	.501
DistMult*	5110	.43	.39	.44	.49
ComplEx*	5261	.44	41	.46	.51
RotatE*	3340	**.476**	.428	.492	.571
QuatE	2272	.303	.179	.386	.530
TuckER	–	.470	**.443**	.482	.526
TransRotatE	2955	.461	.399	.487	.582
halfTransRotatE	2877	.469	.409	**.497**	**.584**
halfTransRotatE-Conj	2783	.469	.418	.484	.571
biRotatE	2940	.451	.385	.483	.575
TransQuatE	**1755**	.469	.416	.487	.577
halfTransQuatE	1862	.472	.422	.488	.575
halfTransQuatE-Conj	1806	.471	.419	.488	.575
biQuatE	2000	.454	.395	.479	.573

TABLE 5.3: **Evaluation 2.** Results of models evaluated on WN18rr. Results for models marked by $\star$ are taken from [11] and results for TuckER are from [45].

Moving the models into quaternion space does not lead to consistent improvements. While biQuatE performs better than biRotatE on all metrics except Hit@10, it fails to match the performance of halfTransRotatE or even TransRotatE. In other words, models that use simpler rotation (e.g. rotation in complex instead of quaternion space, rotation of the head entity only) and translation capture WN18rr better than the models with more complicated rotation.

It can be argued that TransQuatE outperforms biQuatE because it has $4d$ more parameters per relation. However, halfTransQuatE and halfTransQuatE-Conj also outperform biQuatE, and both of these models have the same number of parameters. Moreover, when dimension is set to 100 instead of 200, TransQuatE gets the following scores:
MR = 1869; MRR = 0.477; Hit@1 = 0.429; Hit@3 = 0.495; Hit@10 = 0.569.
Although this smaller model scores significantly lower on Hit@10, on all other metrics it still outperforms biQuatE. Notice also that halfTransRotatE outperforms RotatE on MR, Hit@3, and Hit@10 metrics. In other words, hybrid models that use translation as well as rotation, generally perform better on this dataset.

Surprisingly, QuatE shows a markedly poor performance. While its MR score is pretty average (for models in our suite), on all other metrics it is one of the worst performing models. It scores only 0.303 on MRR, which is over 35% worse than

MRR for halfTransRotatE (0.469). On Hit@1, QuatE scores 0.179, indicating that only under 18% of the correct triples get the rank of 1. Contrast this to Hit@1 of 0.409 for TransRotatE. This is more than twice the score of QuatE.

5.3.3 FB15k

In contrast to WN18, there is a lot more diversity in how models perform on FB15k, as can be seen in Table 5.4. Interestingly, DistMult performs better on this dataset than it does on others: DistMult has the highest MRR and second highest (by only 0.002) Hit@10. This is surprising, considering DistMult's inability to model asymmetric relations, as discussed in Section 2.2, and its low performance on other datasets. For example, in contrast to this, DistMult has the lowest Hit@10 on FB15k-237 and WN18rr, and second lowest on WN18.

	MR	**MRR**	**Hit@1**	**Hit@3**	**Hit@10**
TransE⋆	–	.463	.297	.578	.749
DistMult⋆	42	**.798**	–	–	.893
ComplEx⋆	–	.692	.599	.759	.840
RotatE⋆	40	.797	**.746**	.830	.884
QuatE	**35**	.742	.658	.805	.881
TuckER	-	.795	.741	.833	.892
TransRotatE	**35**	.764	.674	.837	**.895**
halfTransRotatE	37	.750	.656	.825	.886
halfTransRotatE-Conj	38	.767	.691	.824	.885
biRotatE	39	.764	.675	.837	.891
TransQuatE	47	.767	.681	.834	.892
halfTransQuatE	47	.735	.634	.818	.889
halfTransQuatE-Conj	44	.782	.708	**.838**	.892
biQuatE	54	.760	.668	.836	.890

TABLE 5.4: **Evaluation 1.** Results of models evaluated on FB15k. Results for models marked by ⋆ are taken from [11] and results for TuckER are from [45].

Out of the models in our suite, in complex space, TransRotatE generally performs well. It has the overall best Hit@10 and MR, and it is second best (by only 0.001) on Hit@3. Simplifications of TransRotatE do not perform as well: TransRotatE outperforms halfTransRotatE on all metrics, and halfTransRotatE-Conj on all but two. These differences cannot be explained by fewer number of parameters alone. Increasing embedding dimension to 800 instead of 500, we get the following results:

halfTransRotatE: MR = 35; MRR = 0.747; Hit@1 = 0.652; Hit@3 = 0.824; Hit@10 = 0.886
halfTransRotatE-Conj: MR = 36; MRR = 0.763; Hit@1 = 0.683; Hit@3 = 0.826; Hit@10 = 0.887

Except for MR, which is the same for halfTransRotatE–dim800 and TransRotatE–dim500, halfTransRotatE–dim800 still performs worse than TransRotatE–dim500. Analogously for halfTransRotatE-Conj with embedding dimension 800: except for Hit@1, it scores worse than TransRotatE. Hence, it is not merely the larger number of parameters (in the form of embedding dimension) that are responsible for an improved performance of TransRotatE - the rotation on the tail is the key. This claim is supported by performance of biRotatE, which is very similar to that of TransRotatE. This shows that removing translation from TransRotatE (to get biRotatE) hardly changes its performance, but removing rotation from the tail entity generally deteriorates the performance.

We do not observe the same trend in quaternion space. Instead, halfTransQuatE-Conj is the best performing model in this space. In other words, in quaternion space, translation starts to play a bigger role. In fact, removing rotation from the tail entity can improve a model's performance, while removing translation significantly worsens it (compare biQuatE and halfTransQuatE-Conj).

5.3.4 FB15k-237

As can be seen from Table 5.5, the best performing model on FB15k-237 is again TuckER. It significantly outperforms RotatE on all metrics (except for MR, for which we have no data for TuckER). It also significantly outperforms the models in MultEmbed suite.

In contrast to FB15k, biRotatE does not measure well against either TransRotatE or halfTransRotatE models. In fact, it underperforms significantly on all 5 metrics, indicating that for FB15k-237 in complex space, translation plays a more important role than rotation on both head and tail entity.

Making rotation more complex by moving it to quaternion space leads to poorer performance. This is true on all metrics, except for MR, on which TransQuatE and its simplifications outperform all other models. On all other metrics, however,

	MR	MRR	Hit@1	Hit@3	Hit@10
TransE⋆	357	.294	–	–	.465
DistMult⋆	254	.241	.155	.263	.419
ComplEx⋆	339	.247	.158	.275	.428
RotatE⋆	177	.338	.241	.375	.533
QuatE	170	.282	.178	.315	.501
TuckER	–	**.358**	**.266**	**.394**	**.544**
TransRotatE	167	.337	.238	.376	.539
halfTransRotatE	166	.341	.244	.378	.539
halfTransRotatE-Conj	166	.341	.245	.376	.534
biRotatE	161	.327	.230	.363	.527
TransQuatE	158	.332	.236	.366	.527
halfTransQuatE	156	.331	.235	.368	.525
halfTransQuatE-Conj	**154**	.333	.236	.370	.531
biQuatE	168	.324	.227	.359	.523

TABLE 5.5: **Evaluation 2.** Results of models evaluated on FB15k-237. Results for models marked by ⋆ are taken from [11] and results for TuckER are from [45].

halfTransRotatE outperforms all QuatE-based models. It appears that translation is very important on FB15k-237, and complicated rotation is not particularly useful. Increasing the complexity of rotation by moving it to quaternion space not only fails to improve the results, but actually harms them instead.

As is the case with WN18rr, QuatE performs significantly worse than other models, except on MR metric. On FB15k-237, QuatE scores only 0.282 on MRR - this is lower than all other models except for DistMult and ComplEx[1]. The same applies to Hit@1 measure - it is 0.178 for QuatE on FB15k-237, with only two models, DistMult and ComplEx, getting a lower score. In contrast, TransRotatE scores 0.337 on MRR and 0.238 on Hit@1 on this dataset.

In summary, for datasets in which inverse relation pattern plays an important role rotation is particularly important. Datasets that lack in inverse patterns, however, are best captured by hybrid models that use simpler rotation.

[1]QuatE is a generalization of these models, so, this improvement is to be expected.

	FB15k	FB15k-237	WN18
Inverse	69,104	–	7,254
Symmetry	7,740	7,737	–
Implication	3,259	578	–
Equivalence	543	861	–

TABLE 5.6: Size of test datasets for each type of relational pattern. Not all patterns are present in every dataset. By construction, FB15k-237 does not have inverse patterns. WN18 has only inverse patterns and WN18rr does not have any relational patterns, since it is a subset of WN18 with inverse patterns removed.

5.4 Relational patterns

Besides testing MultEmbed models on the standard test datasets, we also test them on test datasets specific for relational patterns. To do, we construct specialized test datasets using logical rules that describe relational patterns. We follow [13], we adopt FB15k rules from [25] and WN18 rules from [26]. FB15k rules were mined automatically by rule mining tools such as AMIE [46] and AMIE+ [47] [25]. And rules for WN18 were constructed manually [26]. For both datasets, we follow [13], and use rules with confidence of 0.8 or higher.[2]

Using these rules, we construct our test datasets in the following manner:

1. For each relational pattern represented by a rule, we find all training triples that appear in the antecedent. For example, given an inverse rule $(?h, r_1, ?t) \Rightarrow (?t, r_2, ?h)$, we first find all training triples that contain relation r_1.

2. From these training triples, we construct candidate test triples by applying the rule to get the consequent triple. Continuing with our example, let (a, r_1, b) be a concrete observed training triple. By applying inverse rule to it, we construct a candidate test triple (b, r_2, a).

3. We discard all candidate test triples if they appear in the training dataset.

Table 5.6 shows the sizes of the test datasets for each relational pattern. Not all datasets have all relational patterns present.

The results of tests on these test datasets are shown in Tables 5.7 – 5.10. Appendix A.3 provides graphs that visualize these scores.

[2]"Confidence" stands for probability of a rule being true.

	Inverse					Symmetry				
	MR	**MRR**	**Hit@1**	**Hit@3**	**Hit@10**	**MR**	**MRR**	**Hit@1**	**Hit@3**	**Hit@10**
TransE	9.9	0.883	0.837	0.92	0.953	8.1	0.337	0.0	0.625	0.853
RotatE	5.6	**0.901**	**0.87**	0.923	0.956	**1.4**	**0.867**	**0.755**	0.979	0.997
TransRotatE	**3.9**	0.9	0.855	0.935	0.971	2.1	0.509	0.052	0.976	0.999
halfTransRotatE	7.2	0.878	0.825	0.922	0.959	2.2	0.5	0.049	0.964	0.997
halfTransRotatE-Conj	7.5	0.88	0.832	0.918	0.957	1.7	0.696	0.419	0.981	**1.0**
biRotatE	5.1	0.899	0.857	0.933	0.966	2.1	0.5	0.047	0.966	0.999
TransQuatE	4.9	0.891	0.842	0.93	0.971	1.9	0.574	0.17	0.983	0.999
halfTransQuatE	5.2	0.848	0.775	0.911	0.967	2.1	0.547	0.154	0.945	0.997
halfTransQuatE-Conj	4.7	0.895	0.847	0.936	0.972	1.6	0.719	0.454	**0.992**	**1.0**
biQuatE	5.2	0.898	0.848	**0.941**	**0.973**	2.3	0.477	0.01	0.949	0.996

TABLE 5.7: **FB15k.** Results obtained on inverse and symmetry test datasets.

	Equivalence					Implication				
	MR	**MRR**	**Hit@1**	**Hit@3**	**Hit@10**	**MR**	**MRR**	**Hit@1**	**Hit@3**	**Hit@10**
TransE	4.0	0.773	0.703	0.809	0.92	1.6	0.926	0.895	0.946	0.987
RotatE	3.4	0.743	0.645	0.803	0.93	2.2	0.867	0.819	0.896	0.958
TransRotatE	2.8	0.763	0.658	0.847	0.961	2.3	0.88	0.839	0.907	0.967
halfTransRotatE	3.2	0.753	0.657	0.814	0.943	2.3	0.878	0.836	0.905	0.96
halfTransRotatE-Conj	3.0	0.755	0.659	0.816	0.962	2.2	0.894	0.856	0.917	0.963
biRotatE	3.4	0.744	0.637	0.828	0.943	3.4	0.844	0.796	0.883	0.928
TransQuatE	3.6	0.637	0.541	0.638	0.919	**1.4**	0.913	0.87	0.944	0.993
halfTransQuatE	3.5	0.657	0.571	0.641	0.924	**1.4**	**0.935**	**0.905**	**0.954**	0.991
halfTransQuatE-Conj	**1.4**	**0.913**	**0.87**	**0.944**	**0.994**	**1.4**	0.931	0.899	0.952	**0.994**
biQuatE	3.4	0.679	0.589	0.693	0.925	1.5	0.921	0.885	0.944	0.99

TABLE 5.8: **FB15k.** Results obtained on equivalence and implication test datasets.

	Implication					Symmetry				
	MR	**MRR**	**Hit@1**	**Hit@3**	**Hit@10**	**MR**	**MRR**	**Hit@1**	**Hit@3**	**Hit@10**
TransE	1.7	0.89	**0.839**	0.918	0.984	5.4	0.365	0.0	0.694	0.891
RotatE	1.7	0.869	0.804	0.911	0.993	**2.0**	**0.633**	**0.325**	**0.939**	**0.994**
TransRotatE	1.8	0.871	0.801	0.92	0.986	2.5	0.461	0.028	0.893	0.989
halfTransRotatE	1.6	0.867	0.796	0.918	0.991	2.6	0.466	0.037	0.894	0.988
halfTransRotatE-Conj	1.7	0.863	0.792	0.923	0.987	2.5	0.519	0.146	0.89	0.986
biRotatE	**1.5**	0.878	0.807	**0.941**	0.99	2.6	0.456	0.03	0.88	0.986
TransQuatE	1.6	0.873	0.807	0.926	0.99	2.5	0.481	0.07	0.893	0.987
halfTransQuatE	**1.5**	**0.891**	0.834	0.938	0.99	2.5	0.487	0.081	0.892	0.99
halfTransQuatE-Conj	1.6	0.871	0.798	0.923	**0.995**	2.6	0.479	0.07	0.885	0.987
biQuatE	1.6	0.86	0.782	0.919	**0.995**	2.4	0.524	0.152	0.894	0.988

TABLE 5.9: **FB15k-237.** Results obtained on implication and symmetry test datasets.

	FB15k-237: Equivalence					WN18: Inverse				
	MR	**MRR**	**Hit@1**	**Hit@3**	**Hit@10**	**MR**	**MRR**	**Hit@1**	**Hit@3**	**Hit@10**
TransE	**3.2**	**0.779**	**0.722**	**0.797**	**0.908**	**1.0**	0.989	0.981	0.998	**1.0**
RotatE	5.0	0.659	0.573	0.689	0.858	**1.0**	**1.0**	0.999	**1.0**	**1.0**
TransRotatE	7.5	0.608	0.54	0.614	0.757	**1.0**	**1.0**	0.999	**1.0**	**1.0**
halfTransRotatE	7.2	0.617	0.546	0.611	0.782	**1.0**	0.999	0.999	**1.0**	**1.0**
halfTransRotatE-Conj	7.2	0.615	0.542	0.628	0.776	**1.0**	0.998	0.997	**1.0**	**1.0**
biRotatE	6.5	0.633	0.556	0.65	0.8	**1.0**	0.999	0.999	**1.0**	**1.0**
TransQuatE	6.6	0.615	0.537	0.629	0.804	**1.0**	0.998	0.998	0.999	**1.0**
halfTransQuatE	6.3	0.628	0.547	0.652	0.808	**1.0**	0.999	0.999	**1.0**	**1.0**
halfTransQuatE-Conj	6.6	0.611	0.536	0.612	0.798	**1.0**	**1.0**	**1.0**	**1.0**	**1.0**
biQuatE	5.5	0.633	0.548	0.654	0.829	**1.0**	0.997	0.994	**1.0**	**1.0**

TABLE 5.10: **FB15k-237 and WN18.** Results for FB15k-237 on equivalence test dataset and for WN18 inverse test dataset.

Models differ in how well they capture different relational patterns. RotatE is best at capturing symmetry, at least on FB15k-237. On FB15k, it still performs

well, but falls short of halfTransQuatE-Conj on Hit@3 and Hit@10. On FB15k, equivalence is best inferred by halfTransQuatE-Conj. However, on FB15k-237, TransE clearly is the best model at inferring this pattern. Implication is best captured by halfTransQuatE and halfTransQuatE-Conj. For inverse pattern, there is no clear best performing model. RotatE does well, according to MRR and Hit@1, but performs significantly worse than TransRotatE, TransQuatE, halfTransQuatE-Conj, and biQuatE on Hit@3 and Hit@10.

In summary, we find that RotatE and halfTransQuatE(–Conj) perform particularly well at capturing different relational pattens. One surprising exception to this rule is TransE's superior performance in inferring equivalence pattern on FB15k-237.

5.5 Injecting Rules

For rule injection, we use the same rules that we used for testing in Section 5.4. Our rule injection method uses grounded rules. We follow [13] and [25] for rule injection, and consider a grounding valid only if its antecedent was observed and its consequent was not. Table 5.11 shows the number of groundings per each rule that we obtained. Not all rules are present in every dataset, and not all rules produce valid groundings. We inject the rules one type at a time, to test each rule's effect independently.

Tables 5.12 and 5.13 show results for rule injection on FB15k. For this dataset, injecting rules generally improves performance. One exception to this is biQuatE. Injecting any type of rules into this model worsens performance on all metrics but MR. Another interesting observation is that injecting symmetry into halfTransRotatE-Conj also fails to improve its performance.

HalfTransQuatE and halfTransQuatE-Conj are generally improved by rule injection, even though these models were already adept at inferring relational patterns, as shown in Section 5.4.

	Inverse	**Symmetry**	**Implication**	**Equivalence**
FB15k	67,757	7,740	3,259	8,771
FB15k-237	–	–	578	861
WN18	116,464	–	–	–

Table 5.11: Number of groundings per each rule.

	Inverse					Symmetry				
	MR	**MRR**	**HIT@1**	**HIT@3**	**HIT@10**	**MR**	**MRR**	**HIT@1**	**HIT@3**	**HIT@10**
TransRotatE	35	**0.774**	**0.686**	**0.845**	**0.898**	36	**0.768**	**0.680**	**0.838**	0.890
halfTransRotatE	**35**	**0.774**	**0.689**	**0.846**	**0.895**	37	**0.755**	**0.664**	**0.829**	**0.887**
halfTransRotatE-Conj	**36**	**0.794**	**0.729**	**0.844**	**0.893**	38	0.763	0.684	0.825	0.885
biRotatE	39	**0.770**	**0.682**	**0.842**	**0.895**	39	0.763	0.671	**0.838**	**0.892**
TransQuatE	**46**	**0.783**	**0.709**	**0.841**	**0.893**	**44**	**0.777**	**0.699**	**0.837**	0.892
halfTransQuatE	**46**	**0.772**	**0.691**	0.836	0.892	**46**	**0.743**	**0.648**	**0.820**	0.888
halfTransQuatE-Conj	44	**0.794**	**0.729**	**0.844**	**0.893**	45	**0.785**	**0.714**	**0.839**	**0.893**
biQuatE	**41**	**0.756**	0.668	0.826	0.887	**41**	0.747	0.649	0.826	0.887

TABLE 5.12: **FB15k.** Results with inverse and symmetry rules injected. We highlight results that are better with the rule injection in bold.

	Implication					Equivalence				
	MR	**MRR**	**HIT@1**	**HIT@3**	**HIT@10**	**MR**	**MRR**	**HIT@1**	**HIT@3**	**HIT@10**
TransRotatE	35	**.765**	.673	**.839**	**.896**	36	**.773**	**.687**	**.845**	**.896**
halfTransRotatE	41	**0.764**	**0.680**	**0.832**	0.886	37	**0.753**	**0.658**	**0.831**	**0.888**
halfTransRotatE-Conj	38	**0.768**	0.691	**0.827**	**0.887**	**37**	**0.772**	**0.697**	**0.832**	**0.887**
biRotatE	39	**0.765**	**0.676**	0.837	**0.893**	**39**	**0.768**	**0.680**	**0.840**	**0.895**
TransQuatE	**45**	**0.765**	**0.678**	**0.835**	0.892	47	**0.768**	**0.683**	**0.836**	0.892
halfTransQuatE	**46**	**0.737**	**0.638**	0.818	0.888	**45**	**0.737**	**0.638**	0.816	0.887
halfTransQuatE-Conj	46	0.782	**0.709**	0.838	**0.893**	45	**0.784**	**0.712**	**0.839**	0.892
biQuatE	**39**	0.745	0.647	0.826	0.887	**40**	0.741	0.642	0.821	0.887

TABLE 5.13: **FB15k.** Results with implication and equivalence rules injected.

	MR	**MRR**	**HIT@1**	**HIT@3**	**HIT@10**
TransRotatE	**141**	.939	.921	.953	.962
halfTransRotatE	**151**	0.946	**0.937**	0.954	0.961
halfTransRotatE-Conj	**163**	0.945	0.936	0.952	**0.959**
biRotatE	**208**	0.944	0.932	0.953	0.960
TransQuatE	198	0.950	**0.945**	**0.954**	0.960
halfTransQuatE	**159**	0.949	0.942	0.952	0.959
halfTransQuatE-Conj	194	0.950	0.945	0.953	0.960
biQuatE	289	**0.948**	**0.941**	0.952	0.960

TABLE 5.14: **WN18.** Results with inverse rule injected.

	Implication					Equivalence				
	MR	**MRR**	**HIT@1**	**HIT@3**	**HIT@10**	**MR**	**MRR**	**HIT@1**	**HIT@3**	**HIT@10**
TransRotatE	168	**0.338**	**0.240**	0.374	**0.540**	167	0.337	0.240	0.376	.538
halfTransRotatE	166	**0.342**	**0.245**	**0.380**	0.538	166	0.341	0.243	**0.379**	0.538
halfTransRotatE-Conj	166	0.339	0.242	0.374	**0.536**	167	0.339	0.242	0.376	**0.535**
biRotatE	176	0.325	0.227	0.361	0.526	163	0.328	0.23	0.364	0.526
TransQuatE	**154**	0.331	0.234	**0.367**	0.527	**157**	0.332	0.236	**0.368**	0.527
halfTransQuatE	**155**	**0.332**	0.235	0.368	**0.526**	158	0.331	**0.236**	0.365	**0.526**
halfTransQuatE-Conj	**153**	0.332	0.235	0.370	0.530	158	0.331	0.236	0.365	0.526
biQuatE	171	0.322	0.225	0.356	0.520	**167**	0.323	0.226	**0.360**	0.520

TABLE 5.15: **FB15k-237.** Results with implication and equivalence rules injected.

Tables 5.14 and 5.15 show results for rule injection on WN18 and FB15k-237 KGs. Injecting rules into models does not lead to significant improvements on these datasets. In many cases the models perform just as well, or even slightly worse after the rule injection. Where there are improvements, they are usually too small to be significant.

We hypothesize that in case of FB15k-237, the number of groundings per each rule is too small to significantly alter a model's performance. In case of WN18, there is an opposite problem. Inverse rule is extremely important to WN18 dataset. Therefore, for any model to perform well on it, it already has to infer inverse rule. In other words, the structure of WN18 already serves as implicit inverse rule "injection". As a result, injecting the rule explicitly does little to improve a model's performance (since there is limited room for improvement.)

Chapter 6

Conclusion and Future Work

In this final chapter of our work, we aim to give an overview of the main results of this work. In Section 6.1, we summarize our contributions and answer the research questions stated at the beginning of this work. In Section 6.2, we discuss possible directions for future research.

6.1 Conclusion

At the start of this work, we posed two research questions:

> Research question 1: Does combining different types of models affect their performance?

and

> Research question 2: Does injecting logical rules into a model that is already learning relational patterns implicitly improve its performance?

To investigate the first question, we start out with two state-of-the-art rotational models: RotatE and QuatE. We implement several experimental models that combine both rotation and translation and test them on the standard KGs.

We find that our models generally perform on par with other state-of-the-art embedding models. In particular, our models usually performed best on Hit@10 and MR metrics. Adding translation to RotatE generally improves its performance (at least on Hit@10 and MR), although admittedly, on other metrics, such as MRR, Hit@1, and Hit@3, our hybrid models failed to consistently outperform RotatE, and in some cases, even performed worse.

However, we find that QuatE-based models generally outperformed QuatE. In quaternion space, models that use single rotation only (i.e. rotation on the head entity only), performed better than other QuatE-based models.

We observe also that datasets which rely heavily on inverse relation are better modeled by rotational models, while datasets with few or no inverse patterns are best captured by models with simpler rotation combined with translation.

Testing on pattern-specific datasets in Section 5.4 indicates that RotatE and halfTransQuatE(–Conj) are particularly good at inferring relational patterns.

To investigate the second question, we injected rules into models and re-trained them. We find that injecting rules does indeed improve performance, but only when

- there are enough groundings to have an effect on a model's performance, and
- the dataset does not heavily rely on the rule to be injected. Otherwise, the models are already forced to learn that rule, and injecting it explicitly has limited effect on a model's performance.

6.2 Future Work

One obvious future direction for MultEmbed development is adding support for more embedding models. As new models are being developed, expanding MultEmbed to support them is essential to keeping it relevant.

Another direction for future research is expanding MultEmbed's rule injection capabilities. One promising approach is the adversarial method described in [48]. In this method, there are two distinct parts in the embedding learning process: the discriminator and the adversary. The discriminator attempts to learn embeddings from the data in the knowledge graph. The adversary, on the other hand, works against this. It takes a set of rules and relation embeddings as input and tries to find entity embeddings that violate these rules. The adversary loss measures the extend to which the rules are violated, and is added as a regularization term to the loss in the training process [48].

Another possible direction for future work is to add Bayesian optimization for hyperparameters [49], similar to [50]. This would avoid the costly grid search and likely find more optimal values for hyperparameters. This can be best done with Python module Hyperopt[1].

[1] `http://hyperopt.github.io/hyperopt/`

Bibliography

[1] Jeff Z Pan, Guido Vetere, Jose Manuel Gomez-Perez, and Honghan Wu. *Exploiting linked data and knowledge graphs in large organisations*. Springer, 2017.

[2] Jose Manuel Gomez-Perez, Jeff Z Pan, Guido Vetere, and Honghan Wu. Enterprise knowledge graph: An introduction. In *Exploiting linked data and knowledge graphs in large organisations*, pages 1–14. Springer, 2017.

[3] 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*, pages 1247–1250. AcM, 2008.

[4] Sören Auer, Christian Bizer, Georgi Kobilarov, Jens Lehmann, Richard Cyganiak, and Zachary Ives. Dbpedia: A nucleus for a web of open data. In *The semantic web*, pages 722–735. Springer, 2007.

[5] George A Miller. Wordnet: a lexical database for english. *Communications of the ACM*, 38(11):39–41, 1995.

[6] Andrew Carlson, Justin Betteridge, Bryan Kisiel, Burr Settles, Estevam R Hruschka, and Tom M Mitchell. Toward an architecture for never-ending language learning. In *Twenty-Fourth AAAI Conference on Artificial Intelligence*, 2010.

[7] Fabian M Suchanek, Gjergji Kasneci, and Gerhard Weikum. Yago: a core of semantic knowledge. In *Proceedings of the 16th international conference on World Wide Web*, pages 697–706. ACM, 2007.

[8] Hugo Liu and Push Singh. ConceptnetâĂŤa practical commonsense reasoning tool-kit. *BT technology journal*, 22(4):211–226, 2004.

[9] Quan Wang, Zhendong Mao, Bin Wang, and Li Guo. Knowledge graph embedding: A survey of approaches and applications. *IEEE Transactions on Knowledge and Data Engineering*, 29(12):2724–2743, 2017.

[10] Kristina Toutanova and Danqi Chen. Observed versus latent features for knowledge base and text inference. In *Proceedings of the 3rd Workshop on Continuous Vector Space Models and their Compositionality*, pages 57–66, 2015.

[11] Zhiqing Sun, Zhi-Hong Deng, Jian-Yun Nie, and Jian Tang. Rotate: Knowledge graph embedding by relational rotation in complex space. *arXiv preprint arXiv:1902.10197*, 2019.

[12] Shuai Zhang, Yi Tay, Lina Yao, and Qi Liu. Quaternion knowledge graph embedding. *arXiv preprint arXiv:1904.10281*, 2019.

[13] Mojtaba Nayyeri, Chengjin Xu, Jens Lehmann, and Hamed Shariat Yazdi. Logicenn: A neural based knowledge graphs embedding model with logical rules. *arXiv preprint arXiv:1908.07141*, 2019.

[14] Richard H Richens. Preprogramming for mechanical translation. *Mechanical Translation*, 3(1):20–25, 1956.

[15] Frans N Stokman and Pieter H de Vries. Structuring knowledge in a graph. In *Human-computer interaction*, pages 186–206. Springer, 1988.

[16] Reind P Van de Riet and Robert A Meersman. *Linguistic instruments in knowledge engineering*. Elsevier Science Inc., 1992.

[17] Ioana Hulpuş, Narumol Prangnawarat, and Conor Hayes. Path-based semantic relatedness on linked data and its use to word and entity disambiguation. In *International Semantic Web Conference*, pages 442–457. Springer, 2015.

[18] Lisa Ehrlinger and Wolfram Wöß. Towards a definition of knowledge graphs. *SEMANTiCS (Posters, Demos, SuCCESS)*, 48, 2016.

[19] Heiko Paulheim. Knowledge graph refinement: A survey of approaches and evaluation methods. *Semantic web*, 8(3):489–508, 2017.

[20] Jennifer Neville and David Jensen. Relational dependency networks. *Journal of Machine Learning Research*, 8(Mar):653–692, 2007.

[21] Kristian Kersting and Luc De Raedt. Bayesian logic programs. *arXiv preprint cs/0111058*, 2001.

[22] Daphne Koller. Probabilistic relational models. In *International Conference on Inductive Logic Programming*, pages 3–13. Springer, 1999.

[23] Ni Lao, Tom Mitchell, and William W Cohen. Random walk inference and learning in a large scale knowledge base. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*, pages 529–539. Association for Computational Linguistics, 2011.

[24] Ni Lao and William W Cohen. Relational retrieval using a combination of path-constrained random walks. *Machine learning*, 81(1):53–67, 2010.

[25] Shu Guo, Quan Wang, Lihong Wang, Bin Wang, and Li Guo. Knowledge graph embedding with iterative guidance from soft rules. In *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[26] 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, 2016.

[27] 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*, pages 2787–2795, 2013.

[28] Zhen Wang, Jianwen Zhang, Jianlin Feng, and Zheng Chen. Knowledge graph embedding by translating on hyperplanes. In *Twenty-Eighth AAAI conference on artificial intelligence*, 2014.

[29] Yankai Lin, Zhiyuan Liu, Maosong Sun, Yang Liu, and Xuan Zhu. Learning entity and relation embeddings for knowledge graph completion. In *Twenty-ninth AAAI conference on artificial intelligence*, 2015.

[30] Guoliang Ji, Shizhu He, Liheng Xu, Kang Liu, and Jun Zhao. Knowledge graph embedding via dynamic mapping matrix. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 687–696, 2015.

[31] Miao Fan, Qiang Zhou, Emily Chang, and Thomas Fang Zheng. Transition-based knowledge graph embedding with relational mapping properties. In *Proceedings of the 28th Pacific Asia Conference on Language, Information and Computing*, pages 328–337, 2014.

[32] Maximilian Nickel, Volker Tresp, and Hans-Peter Kriegel. A three-way model for collective learning on multi-relational data. In *ICML*, volume 11, pages 809–816, 2011.

[33] Bishan Yang, Wen-tau Yih, Xiaodong He, Jianfeng Gao, and Li Deng. Embedding entities and relations for learning and inference in knowledge bases. *arXiv preprint arXiv:1412.6575*, 2014.

[34] Théo Trouillon, Johannes Welbl, Sebastian Riedel, Éric Gaussier, and Guillaume Bouchard. Complex embeddings for simple link prediction. In *International Conference on Machine Learning*, pages 2071–2080, 2016.

[35] Mojtaba Nayyeri, Xiaotian Zhou, Sahar Vahdati, Hamed Shariat Yazdi, and Jens Lehmann. Adaptive margin ranking loss for knowledge graph embeddings via a correntropy objective function. *arXiv preprint arXiv:1907.05336*, 2019.

[36] Sameh K Mohamed, Vít Nováček, Pierre-Yves Vandenbussche, and Emir Muñoz. Loss functions in knowledge graph embedding models. In *DL4KG@ ESWC*, volume 2377, pages 1–10. CEUR-WS. org, 2019.

[37] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems*, pages 3111–3119, 2013.

[38] Petr Hájek. *Metamathematics of fuzzy logic*, volume 4. Springer Science & Business Media, 2013.

[39] Mengya Wang, Erhu Rong, Hankui Zhuo, and Huiling Zhu. Embedding knowledge graphs based on transitivity and asymmetry of rules. In *Pacific-Asia Conference on Knowledge Discovery and Data Mining*, pages 141–153. Springer, 2018.

[40] Quan Liu, Hui Jiang, Andrew Evdokimov, Zhen-Hua Ling, Xiaodan Zhu, Si Wei, and Yu Hu. Probabilistic reasoning via deep learning: Neural association models. *arXiv preprint arXiv:1603.07704*, 2016.

[41] Tim Dettmers, Pasquale Minervini, Pontus Stenetorp, and Sebastian Riedel. Convolutional 2d knowledge graph embeddings. In *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[42] Maximilian Nickel, Kevin Murphy, Volker Tresp, and Evgeniy Gabrilovich. A review of relational machine learning for knowledge graphs. *Proceedings of the IEEE*, 104(1):11–33, 2015.

[43] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[44] Xavier Glorot, Antoine Bordes, and Yoshua Bengio. Deep sparse rectifier neural networks. In *Proceedings of the fourteenth international conference on artificial intelligence and statistics*, pages 315–323, 2011.

[45] Ivana Balažević, Carl Allen, and Timothy M Hospedales. Tucker: Tensor factorization for knowledge graph completion. *arXiv preprint arXiv:1901.09590*, 2019.

[46] Luis Antonio Galárraga, Christina Teflioudi, Katja Hose, and Fabian Suchanek. Amie: association rule mining under incomplete evidence in ontological knowledge bases. In *Proceedings of the 22nd international conference on World Wide Web*, pages 413–422. ACM, 2013.

[47] Luis Galárraga, Christina Teflioudi, Katja Hose, and Fabian M Suchanek. Fast rule mining in ontological knowledge bases with AMIE+. *The VLDB Journal – The International Journal on Very Large Data Bases*, 24(6):707–730, 2015.

[48] Pasquale Minervini, Thomas Demeester, Tim Rocktäschel, and Sebastian Riedel. Adversarial sets for regularising neural link predictors. *arXiv preprint arXiv:1707.07596*, 2017.

[49] Bobak Shahriari, Kevin Swersky, Ziyu Wang, Ryan P Adams, and Nando De Freitas. Taking the human out of the loop: A review of bayesian optimization. *Proceedings of the IEEE*, 104(1):148–175, 2015.

[50] James Bergstra, Daniel Yamins, and David Daniel Cox. Making a science of model search: Hyperparameter optimization in hundreds of dimensions for vision architectures. 2013.

List of Figures

List of Tables

Appendix 1 - Model expressivity proofs

KGE models differ in their ability to infer relational patterns. Here, we provide proofs of models' (in)ability to infer certain patterns.

A.1 TransE

TransE can infer asymmetry, composition, and inverse patterns, but fails to infer symmetry.

Asymmetry:

Let r_a be an asymmetric relation such that triple $(h, r_a, t) \in \mathcal{G}$. Then, $(t, r_a, h) \notin \mathcal{G}$. Applying TransE scoring function 2.3, we get:

$$\mathbf{h} + \mathbf{r}_a = \mathbf{t} \;\&\; \mathbf{t} + \mathbf{r}_a \neq \mathbf{h} \quad \Rightarrow \quad \mathbf{h} + \mathbf{r}_a + \mathbf{r}_a \neq \mathbf{h} \quad \Rightarrow \quad \mathbf{r}_a \neq 0$$

Composition: Consider a grounded composition rule $(h, r_1, s)\&(s, r_2, t) \Rightarrow (h, r_3, t)$. By TransE scoring function, it holds that:

$$\begin{aligned} &\mathbf{h} + \mathbf{r}_1 = \mathbf{s} \;\&\; \mathbf{s} + \mathbf{r}_2 = \mathbf{t} \;\&\; \mathbf{h} + \mathbf{r}_3 = \mathbf{t} \quad \Rightarrow \\ &\mathbf{h} + \mathbf{r}_1 + \mathbf{r}_2 = \mathbf{t} = \mathbf{h} + \mathbf{r}_3 \quad \Rightarrow \quad \mathbf{r}_3 = \mathbf{r}_1 + \mathbf{r}_2 \end{aligned}$$

Inverse: Let $(h, r_1, t) \Rightarrow (t, r_2, h)$ be a grounded inverse rule. Then it holds that:

$$\mathbf{h} + \mathbf{r}_1 = \mathbf{t} \;\&\; \mathbf{t} + \mathbf{r}_2 = \mathbf{h} \quad \Rightarrow \quad \mathbf{t} + \mathbf{r}_1 + \mathbf{r}_2 = \mathbf{t} \quad \Rightarrow \quad \mathbf{r}_1 = -\mathbf{r}_2$$

Symmetry: Let r be a symmetric relation. Then it holds for some $h, t \in \mathcal{E}$ that

$$\mathbf{h} + \mathbf{r} = \mathbf{t} \;\; \& \;\; \mathbf{t} + \mathbf{r} = \mathbf{h} \;\; \Rightarrow \;\; \mathbf{t} + 2\mathbf{r} = \mathbf{t} \;\; \Rightarrow \;\; \mathbf{r} = 0$$

TransE collapses symmetric relations to 0 vectors, and as a result, it is unable to effectively embed these types of relations.

A.2 Dot product models

KGE models that rely on dot product in $\mathbb{R}^d$, such as DistMult, cannot model asymmetric relations, due to the dot product is commutative. To see this, consider an asymmetric relation r. By DistMult scoring function 2.10 we get:

$$\phi(h, r, t) = \sum_{i=0}^{d-1} [\mathbf{M}_r]_{ii} [\mathbf{h}]_i [\mathbf{t}]_i = \sum_{i=0}^{d-1} [\mathbf{M}_r]_{ii} [\mathbf{t}]_i [\mathbf{h}]_i = \phi(t, r, h)$$

But this implies that if the triple (h, r, t) is likely to be true, than so is the triple (t, r, h). Clearly, the very opposite should hold for asymmetric relations [9]. ComplEx avoids this, because dot product in $\mathbb{C}^d$ is not commutative [9], [11].

A.3 RotatE

Proofs for this section are taken from [11].
Symmetry & asymmetry: Let r be a symmetric relation. Then it holds for some $h, t \in \mathcal{G}$ that

$$\mathbf{h} \circ \mathbf{r} = \mathbf{t} \;\; \& \;\; \mathbf{t} \circ \mathbf{r} = \mathbf{h} \;\; \Rightarrow \;\; \mathbf{t} \circ \mathbf{r} \circ \mathbf{r} = \mathbf{t} \;\; \Rightarrow \;\; \mathbf{r} \circ \mathbf{r} = 1$$

Analogously for an asymmetric relation r:

$$\mathbf{h} \circ \mathbf{r} = \mathbf{t} \;\; \& \;\; \mathbf{t} \circ \mathbf{r}' \neq \mathbf{h} \;\; \Rightarrow \;\; \mathbf{t} \circ \mathbf{r} \circ \mathbf{r} \neq \mathbf{t} \;\; \Rightarrow \;\; \mathbf{r} \circ \mathbf{r} \neq 1$$

Composition: Consider a grounded composition rule $(h, r_1, s)\&(s, r_2, t) \implies (h, r_3, t)$. By RotatE scoring function 2.12, it holds that:

$$\mathbf{h} \circ \mathbf{r}_1 = \mathbf{s} \;\&\; \mathbf{s} \circ \mathbf{r}_2 = \mathbf{t} \;\&\; \mathbf{h} \circ \mathbf{r}_3 = \mathbf{t} \quad \Rightarrow$$
$$\mathbf{h} \circ \mathbf{r}_1 \circ \mathbf{r}_2 = \mathbf{t} = \mathbf{h} \circ \mathbf{r}_3 \quad \Rightarrow \quad \mathbf{r}_3 = \mathbf{r}_1 \circ \mathbf{r}_2$$

Inverse: Let $(h, r_1, t) \Rightarrow (t, r_2, h)$ be an grounded inverse rule. Then it holds that:

$$\mathbf{h} \circ \mathbf{r}_1 = \mathbf{t} \;\&\; \mathbf{t} \circ \mathbf{r}_2 = \mathbf{h} \quad \Rightarrow \quad \mathbf{t} \circ \mathbf{r}_1 \circ \mathbf{r}_2 = \mathbf{t} \quad \Rightarrow \quad \mathbf{r}_1 = \mathbf{r}_2^{\star}$$

All four of these conditions are satisfiable, indicating that RotatE is able to effectively capture them.

Appendix 2 - Optimal settings

Best optimal hyperparameter settings are:

learning rate lr: 0.1;
embedding dimension d: 500 for RotatE-based models and 200 for QuatE-based models;
regularizer: L3 with coefficient 0.00002 (only for QuatE-based models);
optimizer: AdaGrad;
negatives samples per one positive: 20;
adversarial sampling temperature α : 0.5.

Dataset specific hyperparameters are:

	batch	γ	loss	# steps
FB15k	1024	24	AMR	150k
WN18	512	12	AMR	150k
FB15k-237	1024	12	NS	450k
WN18rr	512	6	NS	300k

Table 1: KG dependent hyperparameters.

Appendix 3 - Graphs

The following are the plots of HIT@N and MRR scores for different models. To save space in Figures, we use the following abbreviations:

TR:	TransRotatE
hTR:	halfTransRotatE
hTR-C:	halfTransRotatE-Conj
biR:	biRotatE
TQ:	TransQuatE
hTQ:	halfTransQuatE
hTQ-C:	halfTransQuatE-Conj
biQ:	biQuatE

Figures 1 – 4 show plots of MRR and HIT@N scores on the original test dataset. Datasets differ in how much variation there is between models. However, there are clear trends: a model that performs well on one metric tends to perform well on others as well. This trend continues when we test on pattern-specific datasets, as shown in Figures 5 – 12. Inverse pattern on WN18, shown in Figure 12 is an exception to this trend.

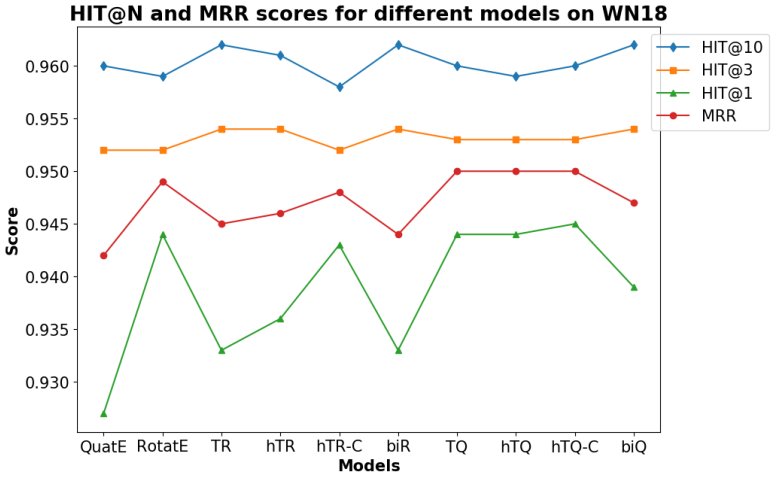


FIGURE 1: HIT@N and MRR scores per each model on WN18 dataset.

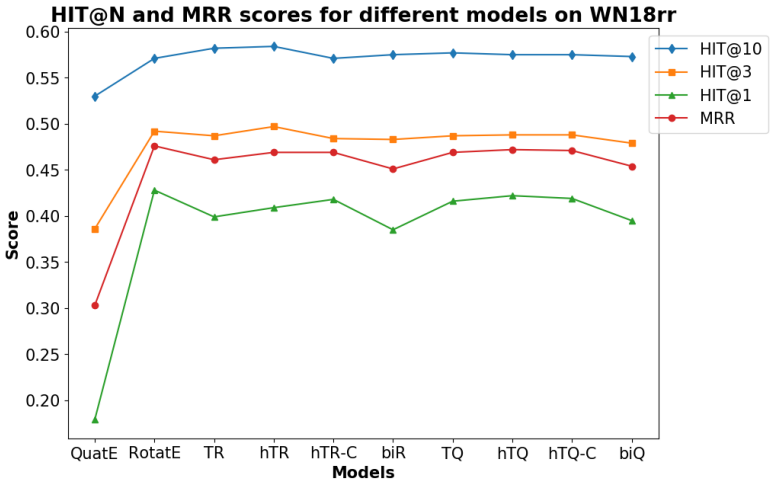


FIGURE 2: HIT@N and MRR scores per each model on WN18rr dataset.

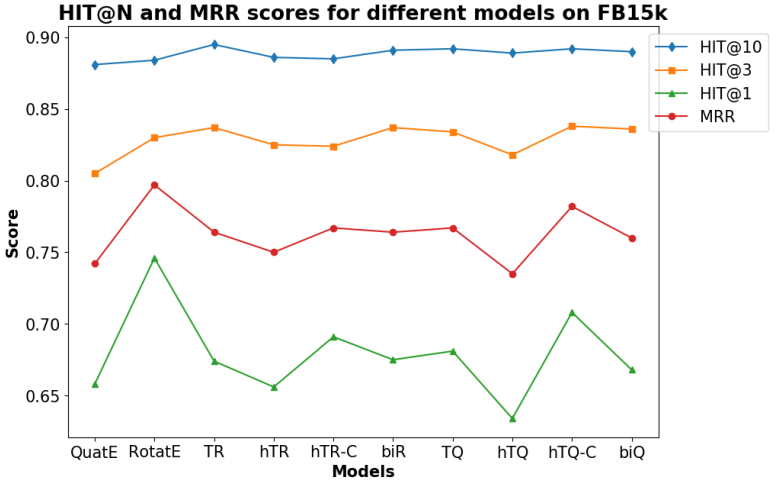


FIGURE 3: HIT@N and MRR scores per each model on FB15k dataset.

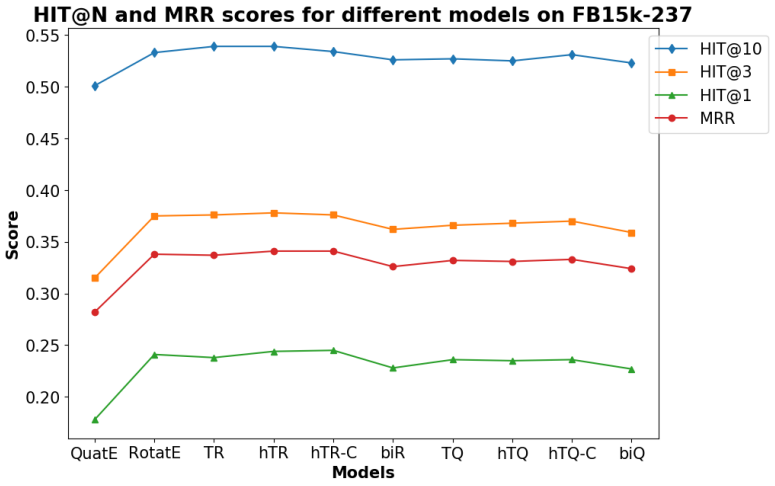


FIGURE 4: HIT@N and MRR scores per each model on FB15k-237 dataset.

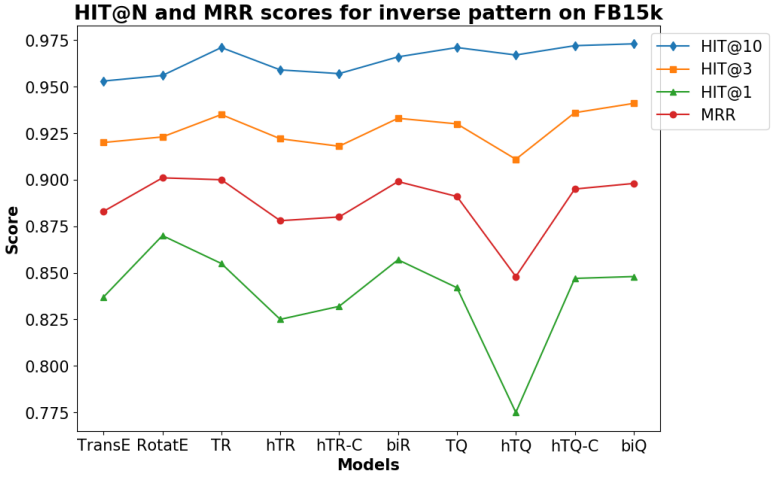


FIGURE 5: HIT@N and MRR scores for inverse pattern on FB15k.

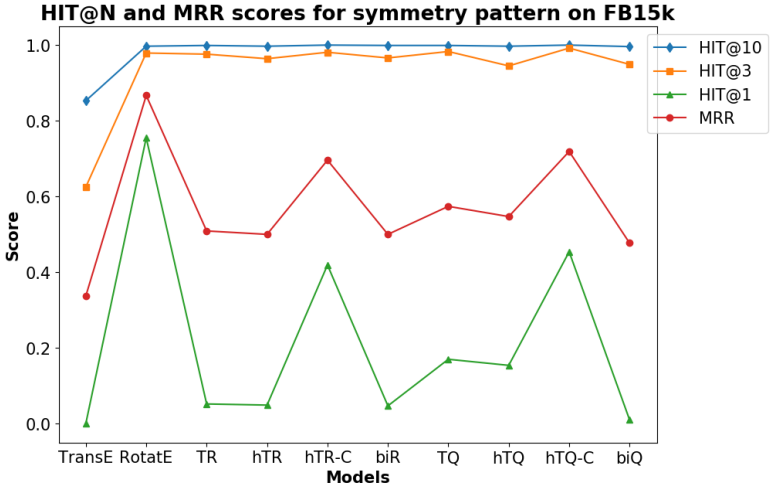


FIGURE 6: HIT@N and MRR scores for symmetry pattern on FB15k.

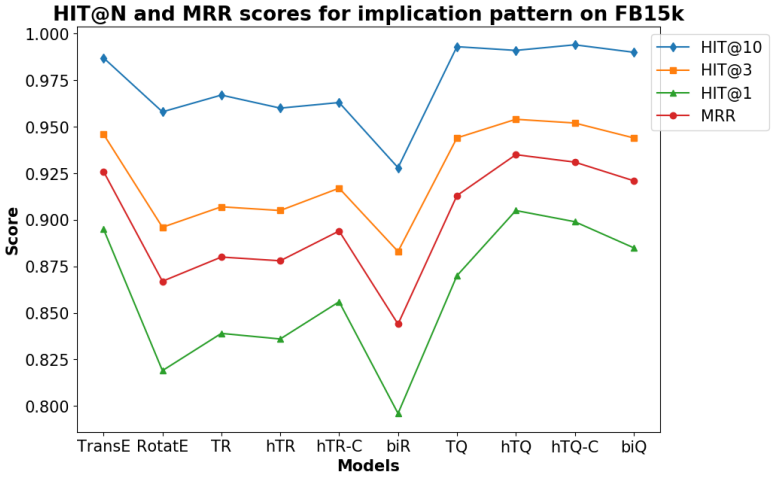


FIGURE 7: HIT@N and MRR scores for implication pattern on FB15k.

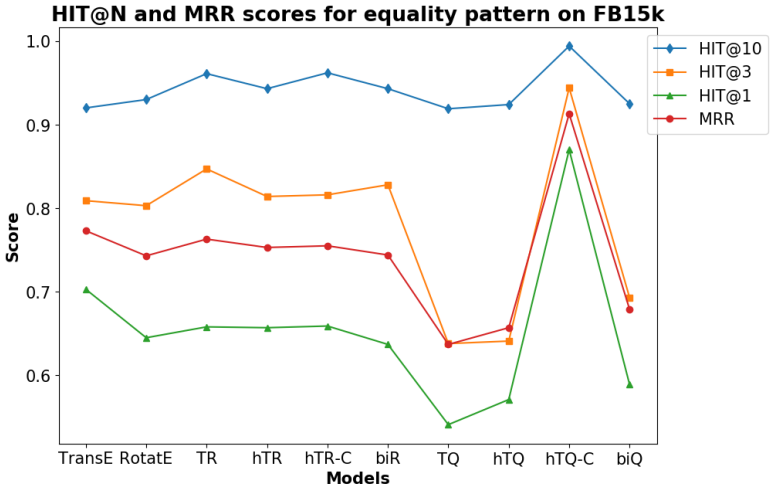


FIGURE 8: HIT@N and MRR scores for equality pattern on FB15k.

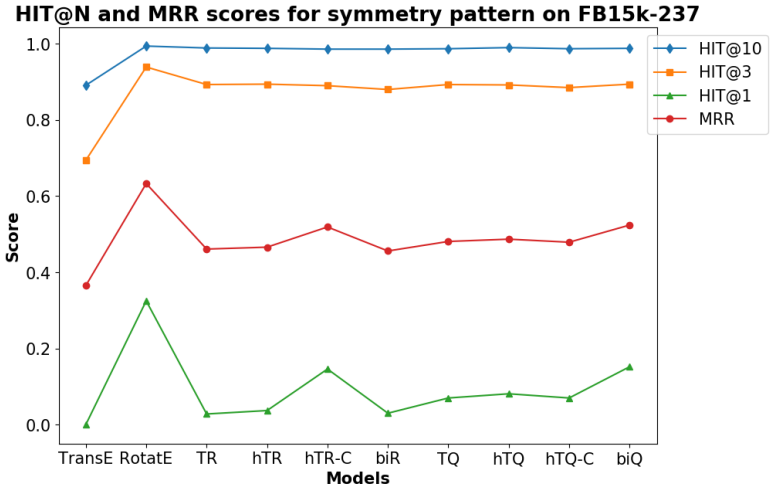


FIGURE 9: HIT@N and MRR scores for symmetry pattern on FB15k-237.

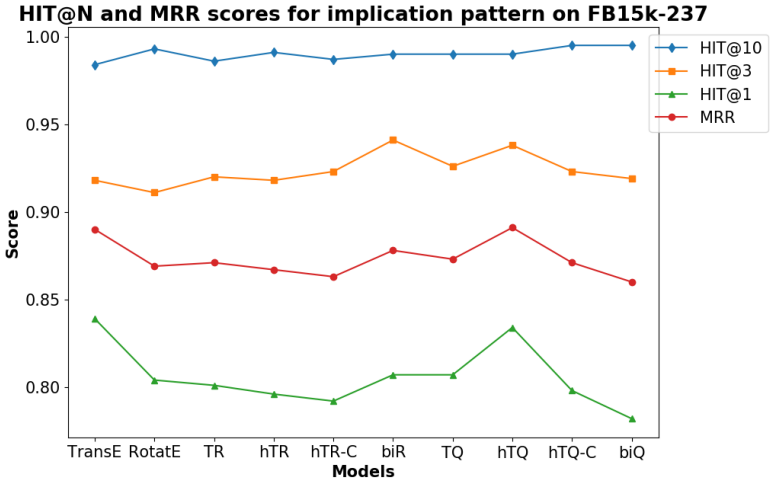


FIGURE 10: HIT@N and MRR scores for implication pattern on FB15k-237.

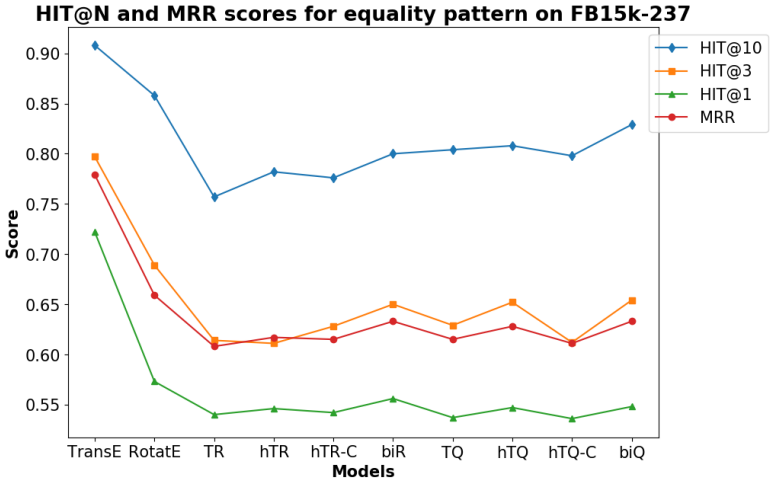


FIGURE 11: HIT@N and MRR scores for equality pattern on FB15k-237.

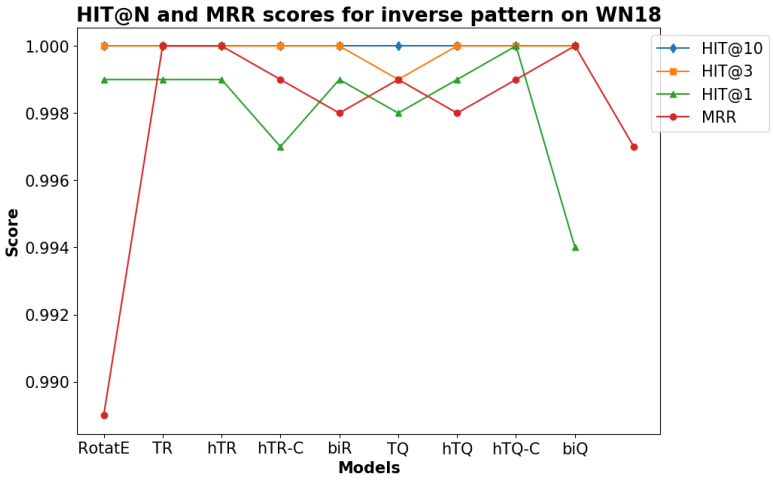


FIGURE 12: HIT@N and MRR scores for inverse pattern on WN18.