RHEINISCHE FRIEDRICH-WILHELMS-UNIVERSITÄT BONN
INSTITUT FÜR INFORMATIK III

Rulect: A System for Extraction, Representation and Learning of Relational Patterns on Knowledge Graph Embeddings

Master Thesis

Reviewer 1: Prof. Dr. Jens Lehmann
Reviewer 2: Prof. Dr. Andreas Behrend

Mirza Mohtashim Alam

June 2023

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 Any published (or unpublished) ideas or techniques from the work of others are fully acknowledged in accordance with the standard referencing practices.

Mirza Mohtashim Alam

Signed: _____

Date: 14.12.2020 _____

Acknowledgements

I praise the Almighty **Allah** for blessing me and providing me the strength to overcome all the hurdles by making everything easier.

I wholeheartedly thank my supervisors **Dr. Sahar Vahdati**, **Mr. Mojtaba Nayyeri** and **Mr. Chengjin Xu**. Without their technical and moral support, it would be tough to finish. I offer my heartfelt gratitude to my supervisor, **Dr. Sahar Vahdati** for her tremendous support, care, valuable guidance for shaping this thesis. I convey my firm gratitude to my supervisor, **Mr. Mojtaba Nayyeri** for his idea and knowledge for this research upon which this thesis is mainly based on. I convey my deepest gratitude toward my supervisor, **Mr. Chengjin Xu** for the codebase, upon which the whole thesis is built.

I want to thank wholeheartedly and convey my gratitude towards **Prof. Dr. Jens Lehmann** and **Prof. Dr. Andreas Behrend** for becoming my examiner for this thesis.

Moreover, I would like to show gratitude towards **Prof. Dr. Jens Lehmann** for giving me the possibility to work with his wonderful group **Smart Data Analytics (SDA)** during my whole masters period. During all this time, by working with many beautiful minds, I learned a lot. My heartfelt thank to **Dr. Hamed Shariat Yazdi** who conducted the **Knowledge Graph Analysis** course and supervised the lab from where I have learned many things about this domain. Special thanks to my **Knowledge Graph Analysis** group mate **Md Rashad Al Hasan Rony** who helped me to implement many things. I show my gratitude to **Afshin Sadeghi** who was co-supervisor in **Knowledge Graph Analysis** lab for his support.

I owe so much thanks to loving my father **Mr. Mirza Moksudul Alam** and my loving mother **Mrs. Taslima Shirin** for being my core strengths during all my life. Without my parent's love, care, and support, nothing would have been possible. I humbly thank my dearest wife **Karishma Mohiuddin** her mental and life support in every aspect of my life. I also thank her and **Md Tansen Khan** for the careful proofreading of this thesis.

My sincere gratitude towards all my family, friends, and well-wishers without whom it would have been difficult for me to be here.

Contents

Chapter 1

Introduction

Knowledge Graphs (KGs) are the hyped technology of recent decades which have changed the entire world of learning methods for Artificial Intelligence (AI). This technology provides a structured knowledge representation through connected nodes via edges. In other words, the nodes are representing entities, and the edges connecting these nodes are denoting the relations. In this way, each true statement from real-world (Obama was born in Hawaii.) is represented in the structure of triples such that (Obama(entity), wasBornIn(relation,Hawaii(entity))). Each of such triples in a KG consists of a head (entity), relation, and tail(entity). This simple data representation has revolutionized tasks based on modern AI such as question answering, link prediction, information retrieval and recommendation system. Despite this huge success, KGs suffer from incompleteness problem by nature. All of the missing information in a KG is considered as unknown under the open-world assumption. Several approaches have been proposed by the machine learning community in order to perform graph completion. One category of such models which got a lot of interest recently (due to their performance) is Knowledge Graph Embedding models (KGE models). KGEs are a set of learning models that are fundamentally used for link prediction. Such models preserve information of KGs (represented in symbolic way) in latent feature space (represented in tensors and vector).

One of the core challenges of KGEs is their capability level in capturing semantic similarity and learning relational patterns (i.e., implication, symmetric, inverse, asymmetric, transitivity) . Generally, relational patterns reside in a KG by nature and can be considered as one of the vital characteristic. However, the extent to

which KGE models are able to learn relational patterns is critical for their performance and accuracy level. This ability directly comes from the design of the embedding models both in the score function and loss function. Authors of [1] provide the name a KGE model where namely TransE [2]. According to [1], TransE is unable to capture one-to-one, many-to-many, one-to-many, and reflexive pattern [3]. Moreover, TransE is not able to encode symmetric patterns [4]. On the other hand, KGE models like RotatE [4] and ComplEx [5] have the ability to capture relational patterns. RotatE can encode and learn patterns such as symmetric, composition, inverse, and asymmetric [4]. ComplEx learns about asymmetric patterns but is unable to capture composition [4]. Thus, the external injection of such relational patterns can be helpful for KGE models [6]. A portion of this thesis is published in [6][1], and shows the effect of inclusion of relational patterns and leakages in RotatE and TransE.

Beside the model design of such KGEs, there are also other factors which can influence the encoding ability of relational patterns. One of the major factors is the injection of logical rules; we call it "rule injection" in order to facilitate KGEs in this regard. Although there are many of the individual KGE models that one evaluate in terms of relational patterns encoding by rule injection, a dedicated framework which can automate this process for evaluation of state-of-the-art KGEs is still missing. This thesis aims at building a system **RULECT**, where inclusion and learning of such relational patterns can be performed alongside of training KGE models. The implementation vastly relies on PyTorch [7], Numpy [8], Scikit-Learn [9], Pandas [10] [11], and Scipy [12]. This system is considered as an interactive module, where the required information can be provided (i.e., the hyperparameters, which model should be trained, whether to train the model with rule injection or not, etc.) before the training begins. In this case, both model parameters, namely embeddings and relational patterns, are learned by the KGE model. In this thesis, by saying about inclusion of rules or injection of rules, we mean inclusion of relational patterns in KGE models.

[1] The Author of this thesis is the first author of this paper. This paper is accepted, presented and in the proceedings of Workshop of Knowledge Representation and Representation Learning co-organized at ECAI Conference, 2020.

1.1 Problem Statement

A study related to the inclusion of rules in the popular KGE model needs to be done to identify the learning effect of the KGE models. The community does not cover until now is an extensive study of rule injection to aid the learning phase and the effect of leakages (covered in the upcoming chapters) among the test data considering the inference power of KGE models. Throughout this thesis, the inclusion of relational patterns refers to the injection of rules or simply rule injection.

Rule injection is a vast new domain for the research area of KGE models. This research aims to demonstrate the inclusion of relational patterns on famous KGE models such as Distmult [13], RotatE [4], ComplEx [5], TransE [2] ,TransComplEx [1] and LogicENN [14]. A part of this study [6][2] already shows the effect of rule injection in two popular KGE models: RotatE and TransE. A reinvestigation with other hyperparameters for these mentioned models with more datasets and further investigation on other embedding models such as TransComplEx, Distmult, ComplEx, and LogicENN is conducted further in this research. Firstly, a comprehensive study has been conducted to learn the effect of rule injection in KGE models.

Secondly, the development procedure is demonstrated for creating leakage-free test sets, which is briefly described and investigated for TransE and RotatE in [6][3]. A further comprehensive investigation has been done on the performance of the mentioned KGE models on the leakage-free test data. The notion of leakages and creation of leakage-free test files are discussed in chapter 3 and 4.

Thirdly, a further investigation has been performed on the same trained models, if only triples of specific relations (relation premise r which participates in the inverse, symmetric, implication, and reflexive rules in the training set) are kept in the test file. The generation process of these test files per pattern has been discussed in chapter 3 and 4.

[2]Please see footnote 1
[3]Please see footnote 1

1.2 Research Questions

Based on the problem statements, three research questions have been formulated. To answer these, specific evaluations need to be performed. The research questions that this thesis is targeting to answer are the following:

Research Question 1: Can we leverage injection of logical rules in the KGE models towards increasing their performance?

To achieve this, the rule injection should be done alongside training the KGE models. These rules can be transformed as groundings, which are to be obtained from those respective training data. These groundings need to be prepared so that they are injected and learned in parallel during the training process of KGE models.

Research Question 2: What can be the effect of rule injection in KGE models for leakage-free test set?

Removal of leakages from the test files can be achieved by searching for specific patterns from the test set's groundings. For an example the grounding pattern of implication rule is defined by $[h, t, r_1, r_2]$ for a particular premise (h, r_1, t) and conclusion (h, r_2, t), in the training set. Here, r_1 is considered as relation premise and r_2 is considered as relation conclusion for implication rule. Searching for the mentioned (h, r_2, t) in the test set would be considered a way to find an implication leakage. These leakages should be removed for all the four mentioned rules (inverse, implication, equivalence, and symmetric) from the test data to obtain a leakage-free test set.

Research Question 3: Can rule injection be helpful when only specific patterns appear in the test set?

To consider specific patterns in the test set, only those relations for which the rules (symmetric, implication, inverse, and reflexive) appear are considered in the test set. All the other triples which do not contain these relations should be removed from the test set. Finally, test sets with four different patterns (reflexive, implication, symmetric, and inverse) need to be obtained. Only focusing on the topmost of each relation type would yield a better test set considering those patterns.

1.3 Thesis Structure

In this thesis, the contribution can be generalized as follows.

1. **Overall system development of RULECT:** A generalized system of KGE models with rule injection module has been developed for easing injection of logical rule alongside the training of KGE models.

2. **Grounding Generation from relational patterns:** AMIE+ [15] has been used to mine rules from a particular KG. Using these mined rules groundings were fetched by putting a threshold based on their importance level.

3. **General loss function for groundings:** A general loss function for each mentioned type of relational pattern (groundings) has been developed, which can calculate the grounding (groundings are covered in chapter 3) losses.

4. **Utilize the grounding losses for each rule types in training:** Modification in the training phase of KGE models is done for rule injection. Thus, they are able to learn from the relational patterns alongside of their regular learning process. These models are considered as rule injected KGE models when the relational patterns are injected as form of groundings. The mentioned KGE models use a general rule loss function (described in chapter 3 in more depth) except LogicENN. During the training of LogicENN, grounding losses developed for each of the rule types is used from [14], and it is different from the general loss function from the other models (RotatE, Distmult, ComplEx, TransComplEx and TransE).

5. **Leakage identification and removal from the test set of popular datasets:** Famous datasets like FB15k [2] and WN18 [2] contain leakages, which posses the ability to make the inference task trivial, a further investigation has been concluded to figure out whether removal of leakage in the test set has any impact on performance for the rule injected trained model.

6. **Investigation on trained model performance on test sets per pattern** A further evaluation of test sets per pattern has been conducted. Generation of such patterns has been briefly discussed in chapter 4. This evaluation for the identification of the fact where the rule injected model performs well or not on these versions of test sets is another contribution of this research.

7. **Performance evaluation of RULECT:** Investigation has been performed to observe the effect of rule injection in KGs such as‘ FB15k [2], WN18 [2], FB15k-237 [16], Kinship [17] and UMLS [18]. From the github repository[4] of ConvE [19] many of the mentioned datasets are collected.

8. **Few additional modules:** Few additional modules: 1. an extended version of LogicENN [20][5], and a visualization module from trained embedding, which is described in [21][6] has been integrated in **RULECT**.

The whole thesis structure has been divided in a way that every chapter explains the core parts. Further, the subparts are explained cohesively. In chapter 2, an overview of the knowledge graph and embedding model has been briefly discussed. The technology and tools needed to build such a system and its internal modules are briefly discussed with examples. In chapter 3, the overview of this thesis's methodology is briefly discussed. This chapter provides the overall idea of this whole thesis (preparation, injection, leakage identification, extraction, etc.). The methodologies are described through both figures and algorithms in order to have clear idea. In chapter 4, the implementation details of the **Rule Extraction** module is discussed. Additionally, the implementation details of leakage detection and removal is also discussed in this chapter. Moreover, the creation of different versions of test sets based on rule patterns is demonstrated in this chapter as well. Chapter 5, provides the implementation details of **Training & Evaluation** module of **RULECT**. This chapter discusses the injection process while training the KGE models. Chapter 6, discusses about the obtained performance in order to answer each research questions. Finally, in chapter 7, a brief discussion about the research outcomes and the future direction has been generalized.

[4]https://github.com/TimDettmers/ConvE

[5]Author of this thesis is a co-author of this paper. This paper is currently under review at Transactions on Pattern Analysis and Machine Intelligence journal

[6]Author of this thesis is a co-author of this paper. This paper is published at IEEE Access.

Chapter 2

Background and Related Work

This chapter discusses about the background of the Knowledge Graph, Knowledge Graph Embedding models, related works, and the required technical backgrounds. In this chapter, at first a brief discussion about Knowledge Graphs with example has been done in section 2.1. A basic understanding of embedding models (KGEs) is demonstrated in section 2.2 since KGEs deal with Knowledge Graphs. This section can aid the preliminary learners about KGE models. Later, some required basics of Machine Learning are covered in section 2.3. A good review of KGs on Machine Learning can be obtained from [22]. In section 2.4, related works which deal with rules and embedding models are notified. Later, in section 2.5, some of the required technical background In section 2.5, a discussion about the technical part (mostly implementation tools, coding languages, and aiding libraries) are demonstrated. Many of the concepts of this chapter are inspired from [23] [24]. Structure of this chapter also follows [23].

2.1 Knowledge Graphs (KG)

A Knowledge Graph is a particular type of knowledge base where relations between the entities are preserved. More specifically, each instance of a KG consists of head (h), tail (t), and relation (r). According to the author of [24], an early version of the Knowledge graph was built by R. H. Richens, who did pre-programming for mechanical translation using such types of nodes linking with respective relations [25].

According to the authors of [26], a mathematical structure was initially given by [27], where the notion of connectivity with edges between the vertices as knowledge units is presented. Author of [24] also mentioned about another research [28] where a more structured way has been incorporated for representing graphs. Here, the authors represented an outline of scientific knowledge in a graph-like structure, which paved the way for elaborating the role of the textual analysis [28].

KG data is represented in a triple like format. A particular triple from KG contains a head (h), tail (t) and a relation (r). The whole KG is connected by such triples. This format is also known as subject (s), predicate (p) and object (o). Head/subjects or tail/predicates are also known as entities or nodes. Relations/predicates connecting entities/nodes can be denoted as edges between the nodes.

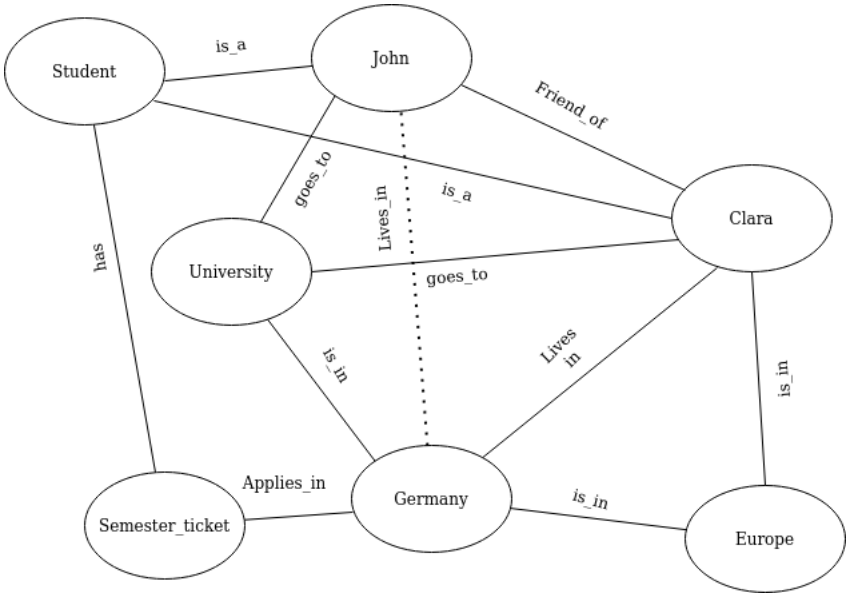


FIGURE 2.1: An generic example of a knowledge graph

An example knowledge graph is presented in figure 2.1. In this figure, we can see that entities (Student, John, University, Semester ticket, Germany, Clara, Europe) are represented as nodes and relations (has, is a, goes to, is in, Applies in, Lives in, Friend of) are stated as the connection between the nodes.

Author of [24] and [23] mentioned some of the important and most useful KGs which are: DBPedia [29], Freebase [30], WordNet [31], and YAGO [32]. Google announced their knowledge graph in 2012 in order to enhance the performance of their search engine [33]. Upon performing a search activity in the google search box, not only we get the particular entity that we want, but also other related things emerge as well. This can happen only due to google has a strong Knowledge Graph trained in their possession. This can be easily demonstrated in Figure 2.2.

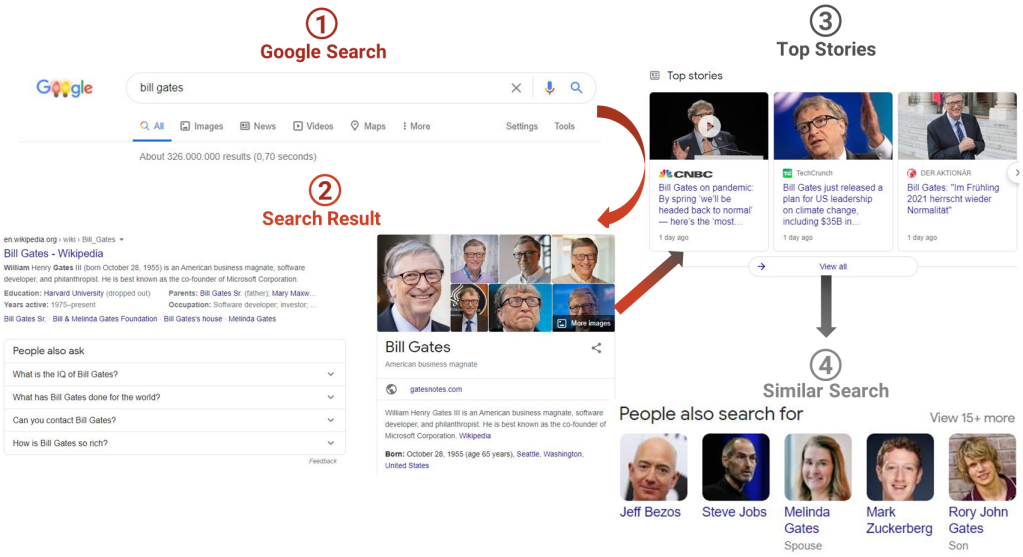


FIGURE 2.2: An example of Google search for a particular entity

.

It can be observed that all other similar or connected entities are coming at the bottom right corner of figure 2.2[1].

2.2 Knowledge Graph Embedding Models (KGE)

The aim of KGE models is to preserve the information of the knowledge graph in continuous latent feature space. RotatE [4] and TransE [2] are examples of two popular KGE models. These embedding models learn to capture semantic similarities of each node and the underlying patterns. The input data in the embedding models are RDF triples, which is, in fact, a KG. The algorithm learns to capture the semantics of the information. Thus, the model parameters are trained to predict unknown facts. The entities e and relations r are mapped to a continuous latent feature space (also called embedding space). Similar to machine learning and deep learning models, as they learn weight vector W, these embedding vectors are learned while training KGE models. A trained KGE model can later be used to predict unknown facts. Author of [23] demonstrates the three essential training steps for KGE models provided in [34]. Based on these, a general training of KGE

[1]This figure is generated by taking a screenshot from author's personal computer of an activity of searching in the Google search-box

models can be described in algorithm 1. In this algorithm, at first, initialization of the required hyper-parameters, and defining the score function (described in subsection 2.3.2) have been done. In the training loop, batches of positive and negative triples are generated. After calculating the loss based on the positive and negative triples using a suitable loss function (discussed in subsection 2.3.1), the optimization (described in subsection 2.3.3) has been done. Thus, the embedding vectors are updated by backpropagation. The training takes place until the loss has been minimized.

Algorithm 1: Learning procedure of a general embedding model

Map all the entities **e** from entity set **E** and relations **r** from relation set **R** to a unique embedding space. First it is assigned in a random fashion.
Initialize a suitable learning rate α
Define KGE model's score function $\mathbf{f}(\mathbf{h}, \mathbf{r}, \mathbf{t})$ to score the plausibility of each triple in the KG.
do
    Generate batch of positive triple $P = (h, r, t)$ and their corresponding negatives $N = (h', r, t')$ by either corrupting head or tail.
    Score positive triples $Pscore = \mathbf{f}(\mathbf{h}, \mathbf{r}, \mathbf{t})$
    Score negative triples $Nscore = \mathbf{f}(\mathbf{h}', \mathbf{r}, \mathbf{t}')$
    Calculate Loss based on $Pscore$ and $Nscore$
    Update the embedding vector by performing backpropagation. α.
while Minimum loss is achieved

One of the applications of the KGE models is predicting missing link. It can be observed from figure 2.1 again that John clearly lives in Germany because he is studying at a university which is German. The missing link is given as a dotted line between John and Germany. Moreover, it is also observable that John is a friend of Clara. Clara goes to University and also lives in Germany. Observing such a similar pattern can lead to an accurate prediction of the missing link between John and Germany. Other significant applications of KGE models include question answering, recommendation systems, and information retrieval.

2.3 A Brief Basic of Machine Learning

Since the KGE models heavily relies on Machine Learning, it is necessary to cover some basics of Machine Learning models first. In machine learning, a program is to learn task T, with experience E using some performance measure P [35]. To predict a function $f(X)$, where X can be considered as the feature vector and the

input of the model. The core task is to learn about the feature vectors in such a way that it can predict correctly on the unknown data or unknown feature vector. The dataset with which the model is trained is called the training set. After successful training, the model can predict unknown data or the test set. Machine Learning can fall into several categories: supervised learning, unsupervised learning, semi-supervised learning, reinforcement learning etc. In supervised learning, the model or the algorithm is aware of the labels or output of the feature vectors. However, these labels are not present in unsupervised learning (i.e., clustering problem). Semi-supervised learning has both labeled and unlabeled data. Most of the KGE models relies on supervised method of learning.

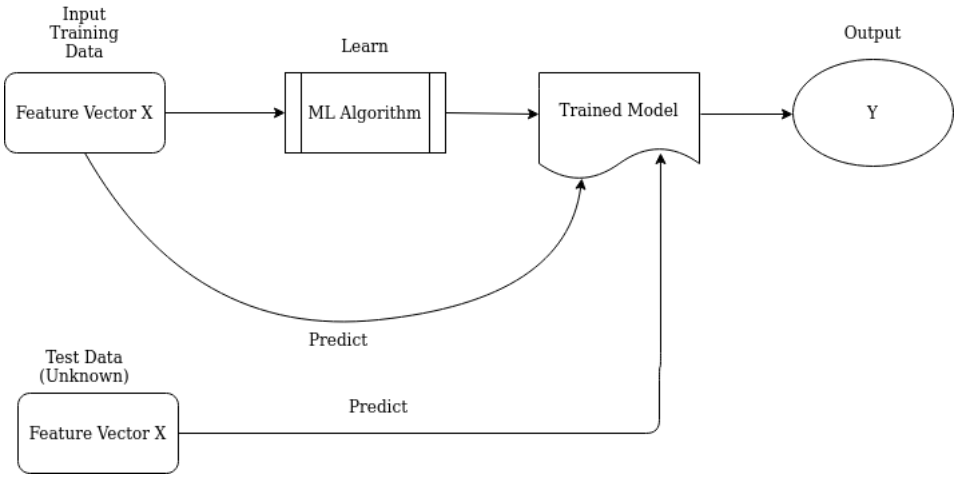


FIGURE 2.3: Core concepts of Machine Learning

.

2.3.1 Loss Function

In general, loss function is a way to determine the difference between the machine learning model's predicted output and the true label for the feature vectors. This difference is called error E or loss L. The error or loss needs to be minimized in order achieve a better prediction model, which means the prediction should be very close to the true labels, though we need to be careful with the term overfitting (model is not very well generalized on the feature vectors). Initially, the error tends to be very high while training. Eventually, it decreases with time as the model learns. An example of a common loss function is mean squared error (MSE)[2].

[2]https://en.wikipedia.org/wiki/Mean_squared_error

2.3.2 Score Function

The score function of an ML model refers to a function f which does the necessary calculation to predict the output the feature vectors X. Depending on the problem statement, this output can be a class (in case of classification problem) or a real value (in the regression problem). Higher accuracy or performance determines the closeness of the predicted score with the actual one. This is a key indicator of the wellness of the learning for ML models. The idea is to make the model generalized on the training set so that it can predict the unknown data with high accuracy. In KGE models, as depicted in algorithm 1 we have both true and corrupted triples. Generally, true triples tend to get higher scores than the corrupted ones. The positive scores and negative scores based on true and corrupted triples then provide a pathway to calculate the overall loss.

2.3.3 Loss Optimization

The main task of the optimization process is to minimize the error or loss during the training process. Some example of optimization algorithm which is commonly used in this domain are: gradient descent [36], adaptive gradient descent [37], RMS prop [38] [39], Adam [40] etc. In this section, a very brief general optimization technique named gradient descent will be discussed. Gradient descent is the most common and popular optimization techniques to optimize neural networks [41]. The core idea of gradient descent is to minimize the loss or objective function by updating the model parameters in the opposite direction of the gradient [41]. The model parameters or weights W can be treated as vectors, which is learned during the training phase. If the model's score function is defined by $f(X, W)$, where X is a feature vector and W is considered as weights or model parameters. Gradient descent provides a way to find optimal model parameters W which minimizes the loss function L by doing nearly correct predictions upon utilizing the feature vectors X. The loss function then can be stated as, $L(f(X, W), Y)$. As demonstrated in [41] and [23], the general steps of the gradient descent algorithm can be stated in algorithm 2. In this algorithm, the model parameters are first initialized, then the gradient is calculated based on a defined loss. Then the weights are updated using $W - \alpha * gradient$ formula until we reach minimum loss.

Algorithm 2: Gradient Descent Algorithm

Initialize learning rate α and the model parameters $\mathbf{W}$
do
  Use the score function, $\mathbf{f}$ to predict output using $\mathbf{X}$ and $\mathbf{W}$
  Do gradient calculation, $gradient = evaluate\_gradient(\mathbf{L}(\mathbf{f}(\mathbf{X}, \mathbf{W}), \mathbf{Y})$
  Do weight update, $\mathbf{W} = \mathbf{W} - \alpha * gradient$
while Loss is minimized

2.4 Related Works

This section provides a discussion about related works. For rule extraction, many frameworks have been established. Name of some rule extraction modules include ALEPH[3], AMIE [42], AMIE+ [15], WARMR [43] [44] [45], WARMER [46]. AMIE supports open world assumption and mines the rules from RDF dataset efficiently [42]. Additionally, AMIE+ is optimized to extract rules from bigger KGs [15]. WARMR uses inductive logic programming to find out frequent queries [43]. WARMER is an extention to WARMR and supports wider range of queries over arbitrary relational databases [46].

There are a handful of software packages which incorporate training and evaluation of KGE models. PyKeen [47], OpenKE [48] are two examples of such software packages. Currently, these software solutions do not support the injection of rules while training. This research aims to utilize the extracted rules from a rule extraction framework described, such as AMIE and perform rule injection in popular knowledge graph embedding models.

LogicENN [14] is a specialized neural network architecture based KGE model which can inject rules in the training phase. Though core idea for injecting rule while training has taken from LogicENN [14], the implementation of LogicENN does not inject rules in other state of the art KGE models. Rule injection process in state of the art KGE models is also done in [24]. Though the approach is similar but they are built on the entirely different codebase. The approach of [24] does not contain the leakage detection, removal and generation of test set per pattern, which is vital parts of this thesis. Additionally, it does not have a separate module which can generate those relational patterns or groundings to inject.

[3]http://www.cs.ox.ac.uk/activities/programinduction/Aleph/aleph_toc.html

2.5 Required Technical Background

For the technical part, it is required a vast domain of technical knowledge, especially one needs to gather knowledge regarding useful python libraries (i.e., core python[4], Numpy[5], Pandas[6], Pytorch[7] and Scipy[8]). Each of these libraries is discussed very briefly in this section to show the researcher a glimpse of the technologies needed to start such research. It is vital to know how to deal with text or tab-separated data in RDF [49] format (i.e., extraction, manipulation, etc.). To achieve this, knowledge of NumPy and Pandas is required. NumPy is a widespread and well-known python package for numerical computation. The benefit of using NumPy is huge since it enables numerical computing a lot easier than it should be.

2.5.1 Handling Data

For handling data, specifically, NumPy and pandas have been used throughout the whole research. Handling data is needed through the following processes: reading data, rule extraction, grounding generation, data pre-processing.

In figure 2.4, some basic operations are performed, such as creating matrices with random numbers, their slicing and element-wise multiplication, etc. Pandas have also been utilized in this research to make the data-related task a lot easier.

[4]https://www.python.org/
[5]https://numpy.org/
[6]https://pandas.pydata.org/
[7]https://pytorch.org/
[8]https://www.scipy.org/

#Code

```
import numpy as np

#create a 3x4 random matrix
x = np.random.rand(3,4)
y = np.random.rand(3,4)
print('x metrix: ', x)
print('######################')
print('y metrix: ', y)
print('######################')
'''
slice a part of these matrix,
so that we can take only first two
columns
'''
x = x[:,:2]
y = y[:,:2]
print('after slicing x: ', x)
print('########################')
print('after slicing y: ', y)
print('########################')
#elementwise multiply these two metrices
z = np.multiply(x,y)
print('final matrix: ', z)
```

#Output

```
x metrix:  [[0.32841674 0.3260079  0.6465202  0.24979736]
 [0.9255823  0.68058914 0.16322699 0.53775435]
 [0.22972041 0.03851051 0.95683608 0.47511077]]
######################
y metrix:  [[0.71248412 0.06340771 0.83709178 0.08276866]
 [0.26989058 0.1186479  0.53456867 0.23953884]
 [0.10146403 0.06282464 0.40245753 0.1195294 ]]
######################
after slicing x:  [[0.32841674 0.3260079 ]
 [0.9255823  0.68058914]
 [0.22972041 0.03851051]]
########################
after slicing y:  [[0.71248412 0.06340771]
 [0.26989058 0.1186479 ]
 [0.10146403 0.06282464]]
########################
final matrix:  [[0.23399171 0.02067141]
 [0.24980594 0.08075047]
 [0.02330836 0.00241941]]
```

FIGURE 2.4: An example of a basic operation of numpy

Pandas is a popular open-source library for python. Even for a beginner in Data Science, pandas can be regarded as a handy tool for Reading, writing, preprocessing, and manipulation of data as it makes everything easier. As an example, we can see that from figure 2.5, reading data becomes how much easier with pandas. The example also clarifies that querying from a pandas dataframe table is very simple, fast, and effective.

#Code

```
import pandas as pd

#Read the comma seperated dataset using pandas
data = pd.read_table('/home/mirza/PycharmProjects/frame_work/dataset/normal/demo_data', sep=',')

#Show the data
print(data)
print('################################################')
#Find the row whose value is greater than certain threshold of some particular column
print(data.loc[(data['A']>50) & (data['C']>100)])
```

#Output

```
    A    B    C    D    E
0   3    2    1    4    4
1   4    3    1    4    7
2  98  124  125  124  567
3  45   13   45   11   18
4  11   35    9    4    1
################################
    A    B    C    D    E
2  98  124  125  124  567
```

FIGURE 2.5: An example of a basic operation of Pandas

These advantages have been a huge motivator for choosing pandas for this research. With the usefulness of pandas, preprocessing and manipulation of data became a lot easier, making the research progress in a deliberate manner.

2.5.2 Utilizing ML-related Components

To deal with Machine Learning related stuff, two libraries are used throughout this research, are: Scikit-Learn [9][9] and PyTorch [7][10]. Scikit-Learn is Machine Learning library consisting of huge amount of machine learning algorithms, preprocessing techniques, model evaluation process to aid researchers and machine learning developers around the globe. A newbie can easily run and deploy machine learning algorithms on data and predict things right away, which makes this tool a norm for early-stage machine learning researchers. An example of fitting training data to a machine learning model is shown in figure 2.6.

```
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.datasets import load_boston
from sklearn.svm import SVR


#import boston dataset
X, y = load_boston(return_X_y=True)
#Split train test data
X_train, X_test, y_train, y_test = train_test_split(X,
                                                    y,
                                                    test_size=0.33,
                                                    random_state=42)

#Fit feature vectors on model
model = SVR().fit(X_train, y_train)
#Predict using the test data
y_pred = model.predict(X_test)
```

FIGURE 2.6: Fitting data to Machine learning model through Scikit Learn

Due to such rich prepared machine learning toolboxes being available in Scikit-Learn, any machine Learning related research or project can achieve greater heights. Another example which comes in handy every time is splitting up training and test set. It has been demonstrated in figure 2.7.

[9]https://scikit-learn.org/stable/
[10]https://pytorch.org/

#Code

```
from sklearn import datasets
from sklearn.model_selection import train_test_split

#Load iris dataset, where X is the feature vector and y is the label
X, y = datasets.load_iris(return_X_y=True)
X_reduced = X[:10,:-1]
y_reduced = y[:10]
print('Feature vector: X= ', X_reduced)
print('Their corresponding label: y= ', y_reduced)

#Divide the dataset into training and test set
X_train, X_test, y_train, y_test = train_test_split(X_reduced,
                                                    y_reduced,
                                                    test_size=0.33,
                                                    random_state=42)


print('Training Feature vector: X= ', X_train)
print('Training label: y= ', y_train)

print('Testing Feature vector: X= ', X_test)
print('Testing label: y= ', y_test)
```

#Output

```
Feature vector: X=  [[5.1 3.5 1.4]
 [4.9 3.  1.4]
 [4.7 3.2 1.3]
 [4.6 3.1 1.5]
 [5.  3.6 1.4]
 [5.4 3.9 1.7]
 [4.6 3.4 1.4]
 [5.  3.4 1.5]
 [4.4 2.9 1.4]
 [4.9 3.1 1.5]]
Their corresponding label: y=  [0 0 0 0 0 0 0 0 0 0]
Training Feature vector: X=  [[5.  3.4 1.5]
 [4.7 3.2 1.3]
 [4.9 3.1 1.5]
 [5.  3.6 1.4]
 [4.6 3.1 1.5]
 [4.6 3.4 1.4]]
Training label: y=  [0 0 0 0 0 0]
Testing Feature vector: X=  [[4.4 2.9 1.4]
 [4.9 3.  1.4]
 [5.4 3.9 1.7]
 [5.1 3.5 1.4]]
Testing label: y=  [0 0 0 0]
```

FIGURE 2.7: Splitting dataset into training and test set with scikit learn

Another significant library used throughout this research is PyTorch. PyTorch is a widely used machine learning and deep learning framework which is developed by Facebook. PyTorch can utilize Graphics Processing Unit (GPU) to fasten up the computing. PyTorch's libraries provide powerful support to building up customizable neural network models instantly.

2.5.2.1 Basic Operation on Tensor

The basic operation on tensor is very common in PyTorch, and it even utilizes GPU resources through CUDA technology. A basic example of multiplication and addition of two matrices is shown in figure 2.8.

2.5.2.2 Torch Autograd Module

Autograd module is responsible for automatic differentiation on tensors. Since it reduces the line of code to perform differentiation operation, many complex neural network architectures can be developed very easily. Declaration of tensors is needed for which the gradients to be computed. Currently, the Autograd module only has support for floating points tensor types [50][11]. An example of gradient

[11]https://pytorch.org/docs/stable/autograd.html

#Code

```
import torch

# Initialize random 3X4 matrices
x = torch.rand(3, 4)
y = torch.rand(4, 2)
z = torch.rand(3, 4)
# Operations
#add two matrices
k = torch.add(x,z)
#multiply two matrices
l = torch.matmul(x, y)

#Print both results
print(k)
print(l)
```

#Output

```
[[1.0280, 1.8038, 1.0671, 1.0256],
 [0.5980, 1.7636, 0.8166, 1.3570],
 [0.9892, 1.4756, 1.3937, 1.0392]])
[[1.3916, 0.9182],
 [0.7683, 0.8823],
 [1.0816, 0.5465]])
```

FIGURE 2.8: An Example of Basic Operation on tensor in PyTorch

calculation is provided in figure 2.9. By calling backward(), function the automatic differentiation can be performed.

#Code

```
118  import torch
119  vector = torch.tensor([0.5, 2.0 , 5.0, 1.0],
120                        dtype=torch.float,
121                        requires_grad=True)
122
123  function = torch.sum(vector ** 2)
124
125  function.backward()
126
127  print(vector.grad)
```

#Output

```
tensor([ 1.,  4., 10.,  2.])
```

FIGURE 2.9: An Example of Gradient Calculation in PyTorch

2.5.2.3 Torch Optim Module

The **optim** Module of PyTorch is responsible for optimization tasks. In general, this package comes with a bunch of optimization algorithms (i.e, Adam [40], Adagrad [37], Adadelta [51]). An Optimizer instance is required to be initialized. To initialize the optimizer object, we need to pass an iterable of dictionaries, which denotes different sets of parameter groups [52][12]. These parameter groups include model parameters, class parameters, etc. Some other parameters, including learning rate, weight decay, etc. Before performing a single optimization step, the

[12]https://pytorch.org/docs/stable/optim.html

zero_grad() function needs to be called to set the gradients to zero. One single step of optimization is done after calling the *step*() function on the initialized optimizer instance. The initialization of an optimizer, zeroing the gradients, and performing a single step of the optimization process is illustrated in figure 2.10.

```
#Initialize the model
solver = torch.optim.Adam(model.parameters(), model.learning_rate)

    #Training in each epoch
    #Use the forward function of the model on feature vectors
    #Calculate loss by some criterion
    #Empty the gradients
    solver.zero_grad()
    #Update the gradients based on loss
    total_loss.backward()
    #Perform a single step of optimization process
    solver.step()
```

FIGURE 2.10: An illustration of using optimizer object from Pytorch

2.5.2.4 Torch nn.Module

Torch **nn.Module** is responsible for implementing neural network structures in PyTorch framework. It provides a speedy and robust implementation of neural network architectures. All neural network classes that are to be implemented should be inherited from this base class **nn.Module** [53][13]. Since this thesis heavily relies on PyTorch libraries to implement the embedding models, **nn.Module** is one of the few important aspects. An example of **nn.Module** utilization is illustrated in image 2.11. in this example, a two-layer neural network has been initialized and its respective forward function has been defined.

```
71  import torch
72  import torch.nn.functional as F
73
74  class custom_two_layerNN(torch.nn.Module):
75      def __init__(self, input, hidden_dimension , output):
76          super(custom_two_layerNN, self).__init__()
77          self.first_layer = torch.nn.Linear(input, hidden_dimension)
78          self.second_layer = torch.nn.Linear(hidden_dimension, output)
79
80      def forward(self, features):
81          output_first_layer = F.relu(self.first_layer(features))
82          output_second_layer = F.relu(self.second_layer(output_first_layer))
83          return output_second_layer
84
```

FIGURE 2.11: Example of utilization of **nn.Module** in pytorch

[13]https://pytorch.org/docs/stable/generated/torch.nn.Module.html

Chapter 3

The *RULECT* System

In this chapter, the whole system's overview is described deliberately; namely, the core modules of **RULECT** are described across several sections which are: 3.1, 3.2 and 3.3. In section 3.2, while discussing about the **Rule Extraction** module, the whole grounding generation process (starting from the output patterns from AMIE+ to groundings per rule type) is demonstrated. Later, the ideology behind the generation of grounding losses for each rule (implication, inverse, symmetric, and equivalence) and how the groundings are injected while training the KGE models have been described and illustrated through both figures and algorithms in 3.3. A brief introduction about leakages, their removal process, and test sets per pattern generation have been discussed in sections 3.4 and 3.5. Lastly, two extra modules as new additions have been depicted at the end of this chapter in sections 3.6 and 3.7. The summary of the whole rule injection process and the notion about leakage in the test set and their removal process have been demonstrated in [6][1] (a portion of this thesis), in this chapter the process has been described in a more elaborate manner through both diagrams and algorithms.

RULECT uses three different modules altogether:

- **Rule extraction module** is responsible for the extraction of rules from KGs. Since groundings are required for obtaining grounding losses for each mentioned rule type, this module is equally responsible for generating groundings from the patterns (rules) that are initially extracted from AMIE+.

[1]The Author of this thesis is the first author of this paper. This paper is accepted, presented and in the proceedings of Workshop of Knowledge Representation and Representation Learning co-organized at ECAI Conference, 2020.

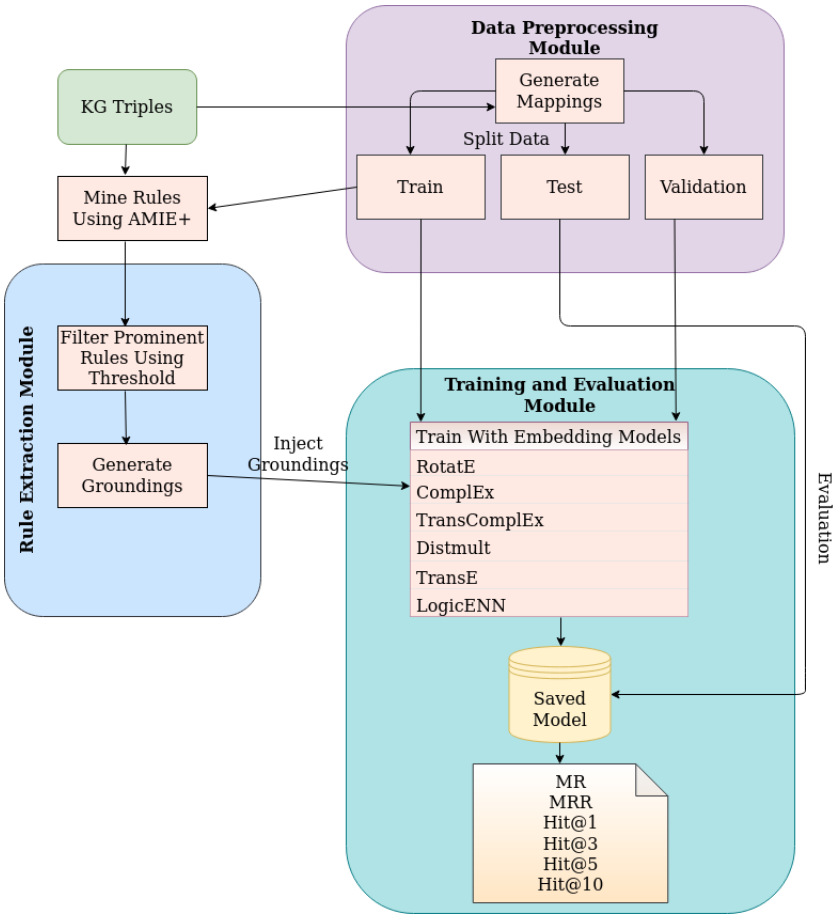


FIGURE 3.1: RULECT: A Rule Injection System for KGE models

- **Data preprocessing module** is responsible for generating the mappings based on the numerical dictionaries of entities and relations which is required in the **Training & Evaluation** module.
- **Training and evaluation module** is responsible for training the KGE models and injecting groundings for each rule type in parallel. One of the major tasks of this module is to save the trained model and evaluate the model based on five evaluation metrics, namely, mean rank, mean reciprocal rank, hit@1, hit@3, hit@5, and hit@10 (these metrics will be briefly discussed in chapter 6).

Figure 3.1, provides an illustration of the whole **RULECT** system. In following sections, these components are discussed elaborately.

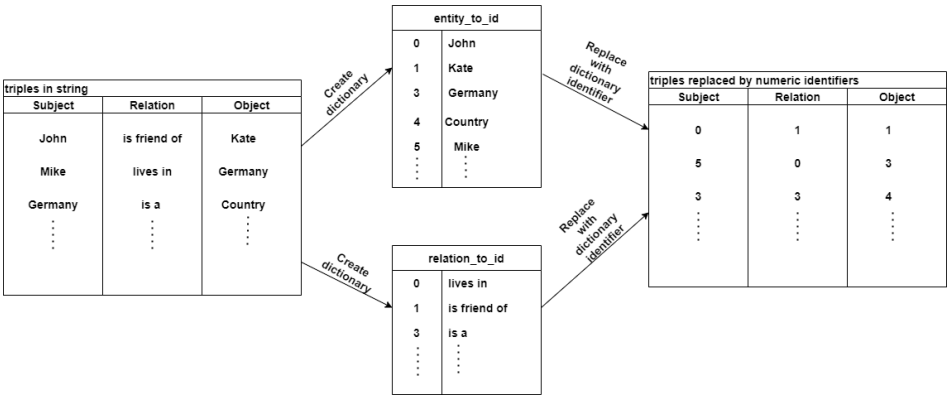


FIGURE 3.2: Mapping Generation

3.1 Data Preprocessing Module

The Data preprocessing module is responsible for the generation of mappings. The **Training & evaluation** module requires numeric values, but most of the KG are consisting of string type RDF triples. Thus, dictionaries are created for individual entities and relations existing in the Knowledge Graphs. The dictionaries contain numeric identifiers of each entity and relation. Eventually, the KGE triples are replaced with these numeric identifiers, which are required in the **Training & Evaluation** module.

3.1.1 Generation of Mappings

Since the **Training & Evaluation** module requires numeric identifiers for the individual entities and relations, that is why mappings are generated in such a way that individual strings of entities and relations existing in that KG are contained in a dictionary labeled by unique identifiers. Later, string triples (both entities and relations) of the particular KG are replaced with these identifiers of the dictionaries. These dictionaries are named as *entities.dict* and *relations.dict* and further saved in local storage. For some of the datasets obtained from several sources, the identifiers (dictionaries) were already given. That is why the generation of those files was explicitly not required for those KGs where the dictionaries of the identifiers were already present. Only replacing those string triples with the given identifiers was a requirement for training and evaluation phase. An illustration of mapping creation is provided in figure 3.2.

3.1.2 Splitting the Train, Test and Validation data

If the KG does not come up with separated train, test and validation data, **RULECT** can do the splitting. Otherwise, this subsection is not required. Small portions from the KGs are randomly taken out for testing and validation purpose. The obtained training data is used to train the KGE models. KGE models are trained on the training data. We need validation data for evaluating the model in training phase at defined intervals since early stopping is planned to be implemented in the system(which is not fully implemented at the moment). Early stopping is a criterion to stop the training process when the performance on the evaluation set starts decreasing.

3.2 Rule Extraction Module

The **Rule Extraction** module is responsible for the whole grounding generation process. This module prepares the rules which are extracted in such a way that they can be injected into the KGE models while training. This module use the output of AMIE+ [15] to generate groundings.

Each of the presented rule types in this research has a premise and a conclusion, which comes as an AMIE+ output (mined rules) (i.e., $?a\ isOwnerOf\ ?b => ?b\ isOwnedBy\ ?a$). These output patterns are not compatible for injection since we need particular head h, tail t, the premise relation r_1, and conclusion relation r_2 to inject. To satisfy such requirement by incorporating the components, groundings are generated by matching the patterns from the AMIE+'s output with the training triple's premise. The illustration for the process of generating the patterns from AMIE+ to the groundings per rule types are provided in figure 3.3. If we consider a practical example, the grounding for $\{?a\ isOwnerOf\ ?b => ?b\ isOwnedBy\ ?a\}$ is transformed into $\{BillGates\ isOwnerOf\ Microsoft => Microsoft\ isOwnedBy\ BillGates\}$ when the extracted patterns matches the training data, where both of these triples $\{BillGates\ isOwnerOf\ Microsoft\}$ and $\{Microsoft\ isOwnedBy\ BillGates\}$ are existing in training set.

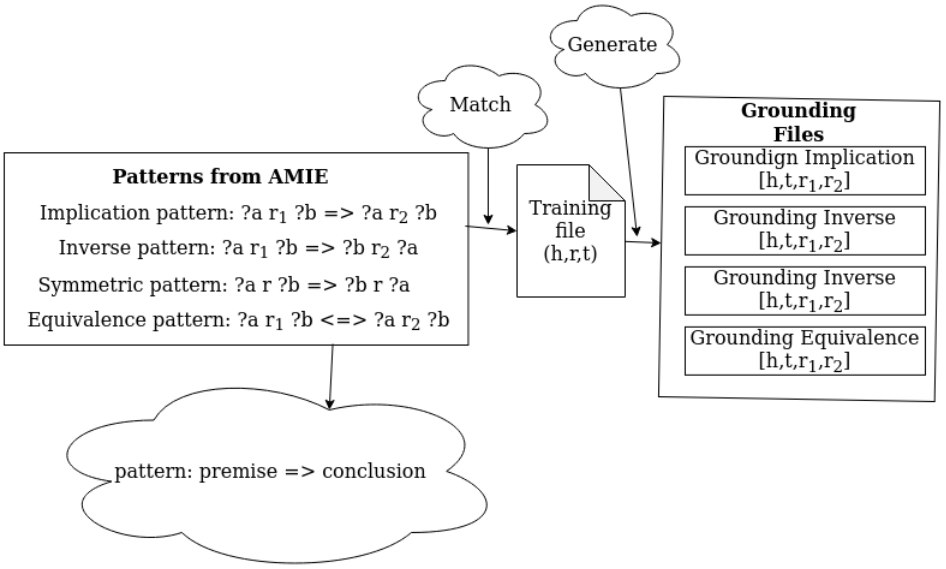


FIGURE 3.3: Illustration of the output patterns from AMIE and groundings

3.2.1 KG Triples

A format of a particular triple is (h, r, t), where h, r and t stand for head, relation and tail respectively. h and t are called entities or nodes. A relation r is connections between the nodes. Another variant for naming the {head, tail, relation} is {subject, predicate , object}.

3.2.2 Extract Rules using AMIE+

Primarily, as already stated, AMIE+ [15] has been used to mine the rules from a particular KG. AMIE+ is a rule extraction framework developed by the YAGO-NAGA team[2] and DIG team[3]. AMIE+ software can be obtained from the website[4] of AMIE [42]. The latest version of AMIE can be found at the Github page [5]. The AMIE+ software is licensed under the **Creative Commons Attribution-NonComercial license v3.0** [6]. AMIE+ uses Javatools which has been released under the terms of **Creative Commons Attribution license v3.0** [7]. In this

[2]https://www.mpi-inf.mpg.de/departments/databases-and-information-systems/research/yago-naga/amie/

[3]https://dig.telecom-paris.fr/blog/

[4]https://www.mpi-inf.mpg.de/departments/databases-and-information-systems/research/yago-naga/amie

[5]https://github.com/lajus/amie

[6]https://creativecommons.org/licenses/by-nc/3.0/

[7]https://creativecommons.org/licenses/by/3.0/

research the output of AMIE+ has been used to achieve the mined rules from a KG. No modification to the AMIE+ software has been done.

AMIE+ requires the training file of a particular KG in TSV (tab separated value[8]) format. In the output, it provides the extracted rules along with their statistical measurements (i.e., Standard Confidence, PCA confidence, Head Coverage etc). In chapter 4, rule generation is discussed more elaborately.

3.2.3 Filter Prominent Rules Using Threshold

Setting a threshold in the *Std Confidence* (i.e., 80%) for a particular rule, causes certain non-significant rules to be filtered out. This method been discussed briefly in chapter 4.

3.2.4 Grounding Generation

A part of this thesis [6][9] demonstrates the summary of grounding generation process, which is a vital step to this research. In this subsection this whole process is elaborated through both images and algorithms. As an example for a particular AMIE+ extracted rule $\{?a\ isOwnerOf\ ?b\}$=>$\{?b\ isOwnedBy\ ?a\}$, the grounding is produced as: $\{BillGates\ isOwnerOf\ Mircosoft\}$ => $\{Microsoft\ isOwnedBy\ BillGates\}$. The mentioned example provides a demonstration of inverse pattern which is producing inverse grounding. According to figure 3.3, the grounding format from this example is: [$BillGates$, $Microsoft$, $isOwnerOf$, $isOwnedBy$] and it is saved in inverse grounding file. Additionally, groundings for other rule types are also generated and saved in local storage in separate files. These grounding files are further used in the rule injection process.

To generate the groundings from extracted rules which are significantly important in the learning phase, specific steps needs to be taken. Algorithm 3, demonstrates the grounding generation process step by step. Figure 3.4 [6] also illustrates grounding generation process where we show a scenario of grounding generation from extracted rule pattern. In algorithm 3, initially, for each of the mentioned rule types, the premise components $\{h_1, r_1, t_1\}$ and conclusion components $\{h_2, r_2, t_2\}$

[8]https://en.wikipedia.org/wiki/Tab-separated_values
[9]please see footnote 1

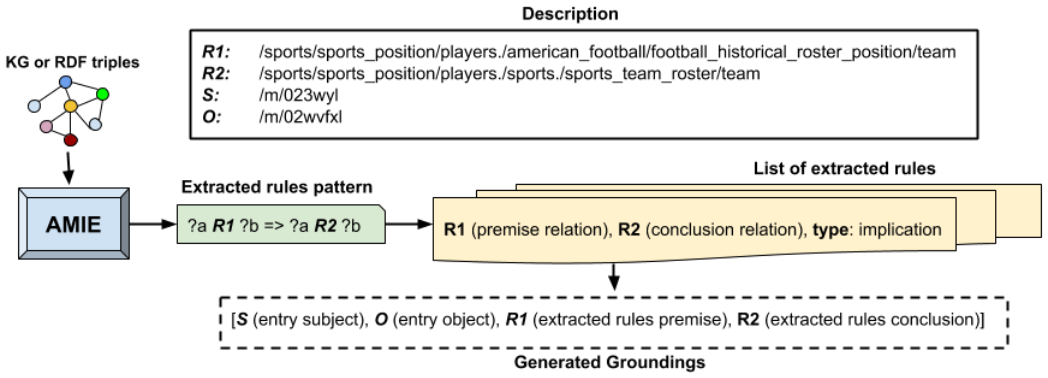


FIGURE 3.4: Preparation of the rule injection data [6]

are decomposed by $SplitRule()$ function. Then a checking has been done to classify these components (i.e., $h_1 = h_2$ and $t_1 = t_2$ and $r_1 \neq r_2$ is a condition for classifying implication rule). The relations from the respective rule's premise (r_1) and conclusion (r_2) are then taken with the corresponding rule type (label) into rule-bag T.

The entries from the rule-bag T is iterated, and the entry's premise (r_1) is matched with the training triple's relation (r). In cases where they match, the conclusion triples are formed by utilizing the rule-bag($\{h, r_2, t\}$ for implication, equivalence and $\{t, r_2, h\}$ for inverse, symmetric). If the conclusions which are formed does not exist in the training set, then, $[h, t, r_1, r_2]$ is taken in the corresponding grounding class (implication grounding, inverse grounding, symmetric grounding and equivalence grounding), where h, t, r_1 and r_2 stands for head, tail, premise relation and conclusion relation. Further these are written in the storage as separate files.

3.2.5 Development of Grounding Loss per Rule Ideology

The groundings obtained from **Rule Extraction** module is injected in order to train a particular KGE model with rule injection. Since the groundings are provided in the training phase, the development of loss function which is able to calculate losses for the groundings per relation type is essential. The core ideology for the development of such grounding loss per rule type has been taken from LogicENN [14] paper. The calculation of grounding loss is based on a constrain where, the score of the premise can be at most equal to the score of conclusion. The formal representation of rules can be seen in table 3.1, which demonstrates grounding loss formulation per rule type by providing scores for each rule's premise

Algorithm 3: Grounding generation

INPUT: *Rules* - Rules extracted from AMIE
OUTPUT: $G_{\text{implication}}$, G_{inverse}, $G_{\text{symmetric}}$,$G_{\text{equivalence}}$

Function `GroundingGeneration(`*Rules*`):`

$classtype = \{implication, inverse, symmetric, equivalence\}$
$T = \{\}$
for *all rule* $\in$*Rules* **do**
  $h_1, r_1, t_1, h_2, r_2, t_2 = SplitRule(rule)$
  if $h_1 == h_2$ *AND* $t_1 == t_2$ *AND* $r_1 \neq r_2$ **then**
    $T = T \cup [premise : r_1, conclusion : r_2, type : implication]$
  else if $h_1 == t_2$ *AND* $t_1 == h_2$ *AND* $r_1 \neq r_2$ **then**
    $T = T \cup [premise : r_1, conclusion : r_2, type : inverse]$
  else if $h_1 == t_2$ *AND* $t_1 == h_2$ *AND* $r_1 == r_2$ **then**
    $T = T \cup [premise : r_1, conclusion : r_2, type : symmetric]$
for *all entry* $i \in T$ **do**
  if $i.type == implication$ **then**
    for *all OtherEntry* $j \in T$ **do**
      if $j.type == implication$ *AND* $i.r_1 == j.r_2$ *AND* $i.r_2 == j.r_1$ **then**
        $i.type = equivalence$
        $T = T - \{j\}$
for *all triple* $t \in \tau$ **do**
  for *all entry* $e \in T$ **do**
    if $e.premise == t.relation$ *AND* $e.type == implication$ **then**
      if $\{t.subject, e.conclusion, t.object\} \notin \tau$ **then**
        $G_{implication} =$
        $G_{implication} \cup [t.subject, t.object, e.premise, e.conclusion]$
    if $e.premise == t.relation$ *AND* $e.type == inverse$ **then**
      if $\{t.object, e.conclusion, t.subject\} \notin \tau$ **then**
        $G_{inverse} =$
        $G_{inverse} \cup [t.subject, t.object, e.premise, e.conclusion]$
    if $e.premise == t.relation$ *AND* $e.type == symmetric$ **then**
      if $\{t.object, e.conclusion, t.subject\} \notin \tau$ **then**
        $G_{symmetric} =$
        $G_{symmetric} \cup [t.subject, t.object, e.premise, e.conclusion]$
    if $e.premise == t.relation$ *AND* $e.type == equivalence$ **then**
      if $\{t.subject, e.conclusion, t.object\} \notin \tau$ **then**
        $G_{equivalence} =$
        $G_{equivalence} \cup [t.subject, t.object, e.premise, e.conclusion]$
return $G_{implication}$, $G_{inverse}$, $G_{symmetric}$,$G_{equivalence}$

and conclusion for the mentioned KGE models (column **Formulation based on score function**) except LogicENN. The last two columns (**Formulation based on NN** and **Equivalent regularization form**) of table 3.1 demonstrates the formulation of represented rules for LogicENN. This table has been taken from [14] and acting as the core ideology for developing the grounding losses per rule type.

TABLE 3.1: Formulation and representation of rules [14]

Rule	Definition $\forall h,t,s \in \mathcal{E} : \ldots$		Formulation based on score function	Formulation based on NN	Equivalent regularization form
Equivalence	(h,r_1,t) (h,r_2,t)	$\Leftrightarrow$	$f^{r_1}_{h,t} = f^{r_2}_{h,t} + \xi_{h,t}$	$\Phi^T_{h,t}(\vec{\beta}^{r_1} - \vec{\beta}^{r_2}) = \xi_{h,t}$	$\max(\left\|\vec{\beta}^{r_1} - \vec{\beta}^{r_2}\right\|_1 - \xi_{\text{Eq}}, 0)$
Implication	(h,r_1,t) (h,r_2,t)	$\Rightarrow$	$f^{r_1}_{h,t} \leq f^{r_2}_{h,t}$	$\Phi^T_{h,t}(\vec{\beta}^{r_1} - \vec{\beta}^{r_2}) \leq 0$	$\max(\sum_i(\vec{\beta}^{r_1}_i - \vec{\beta}^{r_2}_i) + \xi_{\text{Im}}, 0)$
Inverse	(h,r_1,t) (t,r_2,h)	$\Rightarrow$	$f^{r_1}_{h,t} \leq f^{r_2}_{t,h}$	$\Phi^T_{h,t}\vec{\beta}^{r_1} - \Phi^T_{t,h}\vec{\beta}^{r_2} \leq 0$	$\max(\Phi^T_{h,t}\vec{\beta}^{r_1} - \Phi^T_{t,h}\vec{\beta}^{r_2} + \xi_{\text{In}}, 0)$
Symmetric	$(h,r,t) \Leftrightarrow (t,r,h)$		$f^r_{h,t} = f^r_{t,h} + \xi_{h,t}$	$(\Phi_{h,t} - \Phi_{t,h})^T\vec{\beta}^r = \xi_{h,t}$	$\max(\lvert(\Phi_{h,t} - \Phi_{t,h})^T\vec{\beta}^r\rvert - \xi_{\text{Sy}}, 0)$

There are two parts of the formulation: neural network-based formulation and a general formulation based on score function. Some of the specialized parameters in the tables are notified in section 6.1.2 in chapter 6 (mentioned parameters are also described in the paper [14] by the authors of LogicENN). In neural network-based formulation (applicable for the last two columns of the table 3.1), authors presented a new neural-based embedding model named LogicENN [14], where the model can learn the underlying groundings of a KG. The second formulation is more general, where the formulation is mostly based on the score function of KGE models. Here, a particular constrain has been defined for each of these mentioned rule type in a way that, the score of the premise ($f^{r_1}_{h,t}$) can not be bigger than the score of the conclusion ($f^{r_2}_{h,t}$). For each of the rule types, there are different rule loss constants ξ. We can have a closer look at the definition of each rule type in table 3.2. In table 3.2, the formal definition of these four rules are given, where h represents head, t represents tail, r_1 represents relation of the premise (left hand side of the definition) and r_2 represents the relation of the conclusion(right hand side of the definition).

In this research, four of the logical rules discussed in 3.2 are only considered for learning: implication, inverse, symmetric, and equivalence. All the other logical rules remain outside this thesis's scope and can be considered as a future direction.

TABLE 3.2: Representation of Rules

Rule	**Definition**
Implication	$(h, r_1, t) \Rightarrow (h, r_2, t)$
Symmetric	$(h, r, t) \Longleftrightarrow (t, r, h)$
Inverse	$(h, r_1, t) \Rightarrow (t, r_2, h)$
Equivalence	$(h, r_1, t) \Longleftrightarrow (h, r_2, t)$

3.3 Training & Evaluation Module

In this module, the stored groundings for each of the rule type are ready to be injected while KGE models are in the training phase. The aim of the rule injected KGE models is to train on the data with an additional term called grounding loss. The development of grounding loss is based on the ideology provided in 3.2.4, has been illustrated further in algorithm 4. Several KGE models are integrated in **RULECT** for rule injection, and they are: DistMult [13], ComplEx [5], TransComplEx [1], RotatE [4]), TransE [2] and LogicENN [14]. The training phase of these KGE models is modified in a way that the models are trained with an additional term namely, the grounding loss. Afterward, during the evaluation phase, this trained KGE model is evaluated with standard evaluation metrics (discussed in chapter 6). One thing is to notify that, for the training of LogicENN (with the injection of rules) only, the grounding loss functions developed particularly for LogicENN as in [14] have been used. The other mentioned KGE models use a general loss function to calculate grounding loss (described in algorithm 4).

3.3.1 Selected Embedding Models

In order to train and evaluate the effect of rule injection, a handful of popular KGE models are chosen. These selected KGE models for rule injection in **RULECT** are described as follows:

LogicENN A neural-based Embedding model [14] named LogicENN, which has been developed to capture underlying logical rules existing in a particular KG. This model acts as a baseline and motivator for this research. Authors of LogicENN [14] describes their score function is stated in 3.1 [14].

$$f_{h,t}^{r} = \sum_{i=1}^{L} \phi(\langle \vec{w}_i, [\vec{h}, \vec{t}] + b_i \rangle) \beta_i^r = \sum_{i=1}^{L} \phi_{h,t}(\vec{w}_i, b_i) \beta_i^r \\ = \Phi_{h,t}^T \vec{\beta}^r \tag{3.1}$$

In equation 3.1, h,t, and r is denoting the head, tail, and relation of the triple. vector $\vec{w}$ and b denotes the weights and bias of the neural network. L denotes the number of nodes of a particular layer. ϕ denotes the score function of the KGE model. $f_{h,t}^r$ is the score of LogicENN model. The following optimization is done by the authors which can be seen from equation 3.2 [14].

$$\min_{\theta} \sum_{(h,r,t)\in\mathcal{S}} \alpha_{h,t}^r log(1 + exp(-\mathcal{Y}_{h,t}^r f_{h,t}^r)) + \lambda \sum_{i=1} \frac{\mathcal{R}_i}{\mathcal{N}_i} \tag{3.2}$$

$\mathcal{N}$ denotes the total number of groundings, $\mathcal{R}$ represents the rules, and λ is a constant value for rule loss. The label of the triples is represented by $Y_{h,t}^r$. $f_{h,t}^r$ denotes the score for the triples as described in 3.1.

TransE TransE [2] translates the head h towards tail t via the relation r. TransE has a distance function namely, d which calculates the distance between them .

The loss function for TransE has been stated in equation 3.3 [2], where the margin is γ, and h' and t' represents the randomly perturbed head or tail for a particular h and t of a triple.

$$L = \sum_{(h,r,t)\in S} \sum_{(h',r,t')\in S_(h,r,t)} [\gamma + d(h + r, t) - d(h' + r, t')]_+ \tag{3.3}$$

DistMult The Distmult score function is calculated by the multiplication of its head h and tail t with their corresponding relational matrix M_r [13]. Here, M_r is a diagonal matrix. The following equation [13], represents the score function of Distmult.

$$f_{\mathrm{r}} = h^T M_r t \tag{3.4}$$

Similar to TransE, for loss minimization margin ranking loss is used by the authors of Distmult in [13]. The head and tail are corrupted in random fashion to generate negative triple.

ComplEx ComplEx has real Re and imaginary Im parts for entity and relation embeddings. The score function of ComplEx $\phi(h, r, t; \theta)$ is written in the equation 3.5 [5].

$$\begin{aligned}\phi(h, r, t; \theta) =& < Re(w_r), Re(e_h), Re(e_t) > \\ &+ < Re(w_r), Im(e_h), Im(e_t) > \\ &+ < Im(w_r), Re(e_h), Im(e_t) > \\ &- < Im(w_r), Im(e_h), Re(e_t) >\end{aligned} \tag{3.5}$$

here, h, r and t denotes the head, relation and tail. θ is the model parameter. The entity embedding space is consists of both h and t and denoted as e_h and e_t. The relation embedding space is denoted by w_r.

Authors used log likelihood on the model parameter θ. $L2$ regularization has been used in this case which is shown in equation 3.6 [5].

$$\min_{\theta} \sum_{r(h,t)\in\Omega} log(1 + exp(Y_{rht}\phi(h, r, t; \theta))) + \lambda \left\|\theta\right\|_2^2 \tag{3.6}$$

TransComplEx The score function of TransComplEx is obtained by the translation of the head h toward the conjugate of tail $\bar{t}$ via relation r [1]. The score function of TransComplEx is shown in equation 3.7 [1].

$$f_r = \|h + r - \bar{t}\| \tag{3.7}$$

Similar to ComplEx, entity (h and t) and relation (r) vectors are using complex embedding space, which means they have both real and imaginary parts. Margin ranking loss has been used by the authors of [1] to minimize the overall loss similarly as TransE and Distmult.

RotatE RotatE rotates the head h towards tail t via phase relation r. For such operation complex embedding space has been chosen. In equation 3.8 [4], the score is demonstrated, where h is head, t is tail and r is relation.

$$d_r = \|h \circ r - t\| \tag{3.8}$$

The loss is demonstrated in the equation 3.9 [4].

$$L = -log\sigma(\gamma - d_r(h,t)) - \sum_{i=1}^{n} \frac{1}{k} log\sigma(d_r(h'_i, t'_i) - \gamma) \tag{3.9}$$

In this equation 3.9 [4], margin is denoted as γ and the score is $d_r(h,t)$. This score is a distance base between h and t for a particular relation r.

3.3.2 Injection of Groundings

Injection of groundings per rule type are done from the stored grounding files the training phase of selected KGE models. For training of the RotatE, TransComplEx and TransE, loss function described in [4] has been used. For Distmult, ComplEx, and LogicENN, the self-adversarial logistic loss (a combination of binary logistic loss function [5] and self adversarial sampling [4]) has been used.

The table 3.1, shows the ideology behind developing the grounding losses for each mentioned rule types. In this table, the score function of a particular embedding model's premise and conclusion is stated as $(f^{r_1}_{h,t})$ and $(f^{r_2}_{h,t})$ accordingly, where h,t,r_1 and r_2 denotes head, tail, premise's relation and conclusion's relation respectively.

Depending on the design of score function, either score of premise $f^{r_1}_{h,t}$ or score of conclusion $f^{r_2}_{h,t}$ can be bigger. It is possible because some models provide higher scores for positive triples, on the other hand, some provide higher scores for negative ones. In the cases of Distmult, ComplEx and LogicENN, score of premise $f^{r_1}_{h,t}$ is always bigger or equal than score of conclusion $f^{r_2}_{h,t}$. On the other hand, for TransComplEx, RotatE and TransE, score of conclusion $f^{r_2}_{h,t}$ is always bigger or equal than score of premise $f^{r_1}_{h,t}$. This constrain is the basis of the development for the grounding loss per rule type. Obtaining the grounding losses per rule is provided in algorithm 4.

Algorithm 4: Grounding Loss Generation

INPUT: *model* - RotatE, TransE, Distmult, TransComplEx, ComplEx,
groundings - the ground truths,
mode - it contains the type of the groundings (i.e, implication, inverse, symmetric, equivalence),
ξ - the grounding constant, slack variable
OUTPUT: *loss* - The loss of the groundings

Function `grounding_loss`(*model, groundings, mode,* ξ`):`
- **Extract** parts of groundings
- $h, t, r_1, r_2 = groundings[h], groundings[t], groundings[r_1], groundings[r_2]$
- **Check** the mode of groundings
- **if** $mode = inverse\ OR\ mode = symmetric$ **then**
  - $output_{\text{premise}} = f(h, r_1, t)_{\text{model}}$
  - $output_{\text{conclusion}} = f(t, r_2, h)_{\text{model}}$
- **else if** $mode = implication\ OR\ mode = equivalence$ **then**
  - $output_{\text{premise}} = f(h, r_1, t)_{\text{model}}$
  - $output_{\text{conclusion}} = f(h, r_2, t)_{\text{model}}$
- **else**
  - **return** other modes are not available
- **Check** model name
- **if** $(name_{model} = Distmult\ OR\ name_{model} = ComplEx)$ **then**
  - **if** $mode = symmtric$ **then**
    - **return** $relu(|(output_{\text{premise}} - output_{\text{conclusion}})| + \xi)$
  - **else**
    - **return** $relu(output_{\text{premise}} - output_{\text{conclusion}} + \xi)$
- **else**
  - **if** $mode = symmtric$ **then**
    - **return** $relu(|(output_{\text{conclusion}} - output_{\text{premise}})| + \xi)$
  - **else**
    - **return** $relu(output_{\text{conclusion}} - output_{\text{premise}} + \xi)$

In algorithm 4, the input is the embedding model which is to be trained, since the score function f of the embedding model is used to generate the score of premise ($output_{\text{premise}}$) and conclusion ($output_{\text{conclusion}}$). Other inputs are groundings, *mode* (*mode* indicates which type of grounding is obtained as input to the algorithm), and ξ. ξ is a constant value (it may vary per grounding-type). The function's output is a particular grounding loss defined by the constraints described in table 3.1.

In algorithm 5, the inclusion of grounding loss in the training phase is demonstrated. Here, it uses the *grounding_loss*() function defined in algorithm 4. In this algorithm, first, all the required hyperparameters have been initialized, including rule loss constant λ and all the individual constants ξ which are used for each

of the grounding type. A KGE model is needed to be initialized by defining its score function f beforehand. This score function $f(h, r, t)$ is required for scoring the triples (both true and corrupted ones).

Algorithm 5: Injection of ground truths while training embedding models

Map each entity **e** and relation **r** to a unique vector. First it is assigned in a random fashion.
Initialize a suitable learning rate α
Initialize a rule loss constant λ
Initialize an implication loss constant ξ_{im}
Initialize an inverse loss constant ξ_{inv}
Initialize an symmetric loss constant ξ_{sym}
Initialize an equivalence loss constant ξ_{eq}
Initialize other hyperparameters such as batch size β, embedding dimension **d**
Define a score function $\mathbf{f}(\mathbf{h}, \mathbf{r}, \mathbf{t})$ to score the plausibility of each triple in knowledge graph.
do
  Generate batch of positive triple $\mathbf{P} = (\mathbf{h}, \mathbf{r}, \mathbf{t})$.
  Generate batch of C negative triple $\mathbf{N} = (\mathbf{h}', \mathbf{r}, \mathbf{t}')$ per positive.
  Generate batch of Implication groundings $\mathbf{G}_{\text{imp}}$.
  Generate batch of Inverse groundings $\mathbf{G}_{\text{inv}}$.
  Generate batch of Symmetric groundings $\mathbf{G}_{\text{sym}}$.
  Generate batch of Equivalence groundings $\mathbf{G}_{\text{eq}}$.

  Score positive triples $\mathbf{Pscore} = \mathbf{f}(\mathbf{h}, \mathbf{r}, \mathbf{t})$
  Score negative triples $\mathbf{Nscore} = \mathbf{f}(\mathbf{h}', \mathbf{r}, \mathbf{t}')$
  Calculate base loss $\mathbf{L} = \mathbf{loss}(\mathbf{Pscore}, \mathbf{Nscore})$

  Calculate implication rule loss $\mathbf{L}_{\text{im}} = \mathbf{grounding\_loss}(\mathbf{f}, \mathbf{G}_{\text{im}}, \xi_{\text{im}})$
  Calculate inverse rule loss $\mathbf{L}_{\text{in}} = \mathbf{grounding\_loss}(\mathbf{f}, \mathbf{G}_{\text{inv}}, \xi_{\text{inv}})$
  Calculate symmetric rule loss $\mathbf{L}_{\text{sym}} = \mathbf{grounding\_loss}(\mathbf{f}, \mathbf{G}_{\text{sym}}, \xi_{\text{sym}})$
  Calculate equivalence rule loss $\mathbf{L}_{\text{eq}} = \mathbf{grounding\_loss}(\mathbf{f}, \mathbf{G}_{\text{eq}}, \xi_{\text{eq}})$

  Calculate average grounding loss $\mathbf{L_g} = \mathbf{Average}(\mathbf{L}_{\text{im}}, \mathbf{L}_{\text{inv}}, \mathbf{L}_{\text{inv}}, \mathbf{L}_{\text{eq}})$
  Calculate total loss $\mathbf{TL} = \mathbf{L} + \lambda * \mathbf{L_g}$

  Update the embedding vector by performing backpropagation on Total Loss **TL** with learning rate α

while Minimum loss is achieved

In algorithm 5, it can be seen that, in each epoch, both the training set and the groundings are to be fetched in mini-batches. For each true triple (h, r, t), corrupted triple (h', r, t') has been be generated by either corrupting head h or tail t. For each positive (true) triple, one or multiple negative (corrupted) triple is

generated. Here the number of the negative triple is denoted by C. By forwarding the positive batch of triples P and the negative batch of triples N to the score function f of the embedding model, positive score $Pscore$ and negative score $Nscore$ are calculated. Based on the $Pscore$ and $Nscore$, base loss L is generated. Next, the losses for each individual groundings (implication (L_{im}), inverse (L_{inv}), symmetric (L_{sym}) and equivalence(L_{eq})) are calculated by passing those groundings as parameters to the function $grounding\_loss()$, which has been described in algorithm 4. All of these grounding losses are then averaged and named as average grounding loss, L_g. L_g is further added with the base loss L, multiplied with a rule loss constant λ. Finally, this epoch ends by updating the parameters on the basis of total loss. The whole process repeats until a minimum loss is achieved. The process was originally created for rule injection of LogicENN model [14] but now this is adapted to inject rule in other KGE models as well.

3.4 Leakage Detection and Removal

To investigate the second research question, detection and leakage removal from the KG test sets are required. As mentioned previously, in this research, leakages are the conclusions in the test set whenever the respective premise resides among the training set. An illustration of leakage can be seen in figure 3.5.

To evaluate our trained models on these leakage-free test sets, removing leakage from the test set is necessary. Other researchers removed leakages from the training

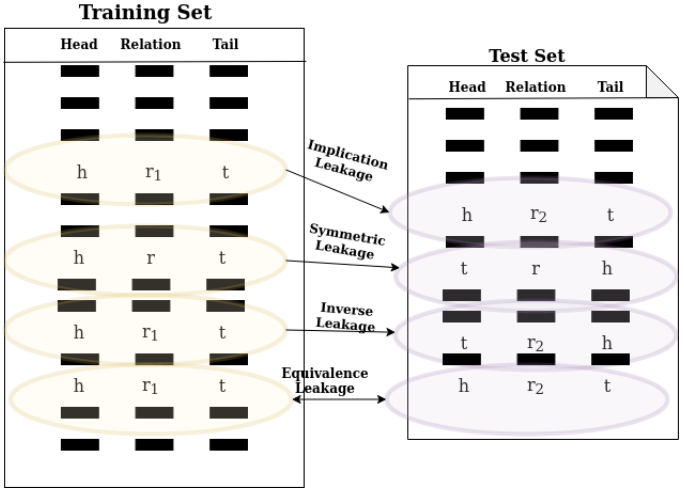


FIGURE 3.5: Illustration of leakages in the test set

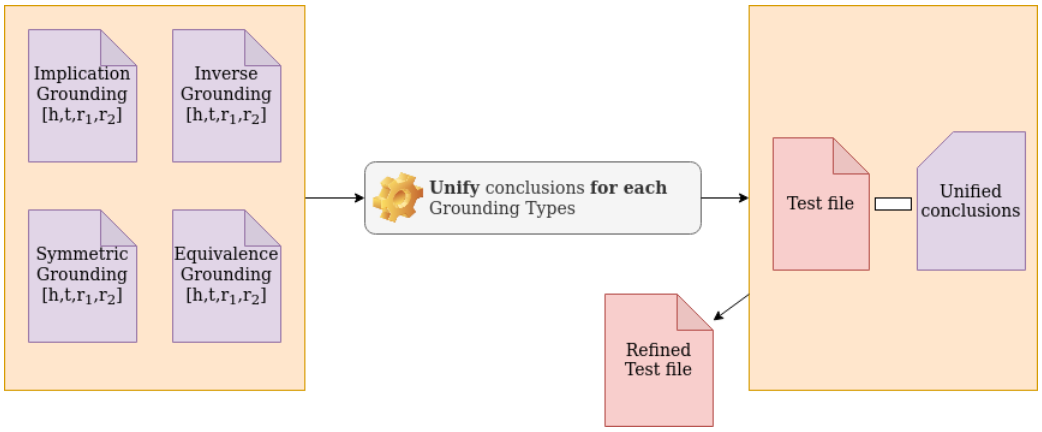


FIGURE 3.6: Illustration of leakage removal from test set

set (i.e., FB15k-237 removed the leakages from the training set by removing those relations for which leakages exist). Since this particular thesis requires the training file to be rich in patterns, leakages from the test sets have been removed. The leakage removal process from test sets requires those grounding files since those contain both premise and conclusions for a particular training data, and it is in the form of $[h, t, r_1, r_2]$, where r_1 and r_2 is premise's relation and conclusion's relation respectively. To remove the leakages, the following steps need to be taken:

- **Fetch** the conclusions from respective grounding files.
- All the fetched conclusions needs to be **unified**.
- **Remove** the unified conclusions from the KG test files.

The removal of leakages is illustrated in figure 3.6.

3.5 Generation of Test Set per Pattern

To evaluate the third research question, test sets per pattern has been generated. For this reason, a separate functionality is added to generate these versions of test file. This requires the KG's training file and the corresponding test file. The main idea is to search for specific relations for which four patterns (implication, inverse, symmetric, and reflexive) exist in the training file. The reflexive patterns are the triples where the head h and tail t are same for a particular relation r. The other

patterns are already discussed in previous sections. Based on the pattern frequency per relation, only top relations are considered. Finally, only the triples containing those relations are kept in the test set. The details of the implementation of such versions of the test set are provided in chapter 4.

3.6 Extra Module: Extended version of LogicENN

An extended version of LogicENN from [20][10] has been integrated into the pipeline of **RULECT**. The extension for LogicENN [20] has several parts: removal of groundings, making the first layer fixed not to learn anything and using a double LogicENN model. All of these extensions are described in the following.

3.6.1 Conversion of grounding

According to [20], We train the models with integration of reverse triples (for (h, r, t) we include (t, r^{-1}, h) in the training data by adding a new relation) with the training set. This inverse and symmetric groundings can be avoided by including an additional relation r^{-1} which corresponds to the particular relation r.

Inverse grounding removal [20] demonstrates that, for each inverse rule $(h, r_1, t) \implies (t, r_2, h)$, if we consider a reverse triple $(h, {r_2}^{-1}, t)$ corresponding to the conclusion (t, r_2, h), we get $(h, r_1, t) \implies (h, {r_2}^{-1}, t)$. Thus, inverse grounding is removed by converting it to implication.

Symmetric grounding removal [20] states the procedure of symmetric grounding removal. For each symmetric rule $(h, r, t) \implies (t, r, h)$, we can consider a new triple with a new relation corresponding to the conclusion of symmetric rule. Thus, (t, r, h) can be a correspondence of (h, r^{-1}, t). Hence, the symmetric grounding can be converted to equivalence by $(h, r, t) \Leftrightarrow (h, r^{-1}, t)$.

[10] Author of this thesis is a co-author of this paper which is currently submitted to Transactions on Pattern Analysis and Machine Intelligence journal.

3.6.2 Fixing first layer

According to [20], to make the model not too much burdened with model parameters θ, the functionality has been added for fixing the first layer of the neural network of LogicENN. This approach might even cause the model to learn faster than a regular LogicENN.

3.6.3 Training with dual LogicENN model

According to [20], in order to make the learning robust, two LogicENN has been combined. First LogicENN learns by injection of original grounding $[h, t, r_1, r_2]$. Another LogicENN model learns by injecting reverse grounding $[t, h, {r_1}^{-1}, {r_2}^{-1}]$ which corresponds to the original grounding. The combined output of both LogicENN provides ultimate output.

3.7 Extra Module: Visualization of Trained Embedding

The visualization system for the trained embedding described in [21][11] has also integrated into the pipeline. The initial entity types and the idea for such visualization process for trained embedding have taken from [54]. The same authors also developed one of the other models, which learns contextual embeddings [55]. The data for each entity type has been obtained from the author's Github repository[12]. For the visualization, two function has been illustrated which are demontrated in algorithm 6 and 7. Algorithm 6 demonstrates the grouping of entities per type in a listwise manner. The input for the algorithm 6 is in the form of $[entity, type]$ and the function provides an output ($data\_out$) of grouped entities per type as $[type, frequency, list(entities)]$. The output of the algorithm 6 is provided as the input of algorithm 7 for visualization in two-dimensional projection using T-SNE [56]. Algorithm 7, takes additional inputs as the indicated types (the selected entities of this type are considered in two-dimensional projection) and the trained embedding vectors in a matrix form where each row corresponds to an

[11] Author of this thesis is a co-author of this paper. This paper is published at IEEE Access.
[12] https://github.com/cmoon2/knowledge_graph

entity. This algorithm's output is a visualization window, where d dimensional embedding vectors are projected in two dimensions for each indicated type using separated color code. Some visualizations of trained embedding are illustrated at the end of chapter 6.

Algorithm 6: Type creation for visualization

INPUT: $data\_in$ - In the format $[entity\_name, type]$
OUTPUT: $data\_out$ - In the format
$[type, frequency, list(matched\_entity\_name)]$

Function `GroupEntityTypes(`$data\_in$`):`
- **Get unique types** from the $data\_in$ as $unique\_types = data\_in['type']$
- **Initialize** an empty list $data\_out$
- **for** $type \in types$ **do**
  - **Group** all the entities which matches $type$
  - **Count** how many entities matches with current $type$
  - **Form** array $temp = [type, frequency, list(matched\_entity\_name)]$
  - **Append** $temp$ with $data\_out$

return $data\_out$

Algorithm 7: Visualizing trained entity embedding

INPUT: $data\_out$ - In the format
$[type, frequency, list(matched\_entity\_name)]$,
$indicated\_types$ - To visualize embedding of those types,
$trained\_embedding$ - matrix of trained embedding vector where each row corresponds to each entity
OUTPUT: $visualization\_window$ - Embedding visualization for each given category

Function `GenerateVisualization(`$data\_out$`):`
- **Initialize** an empty list $temp$
- **for** $type \in indicated\_types$ **do**
  - **Fetch** the indexes of entities of current $type$
  - **Form** vectors containing $[indexes, type]$
  - **Stack** all the vectors with $temp$
- **Retrive** those embedding vectors for the indexes residing in $temp$
- **Provide** each type as separate color code
- **Project** those embedding vectors of n dimension to 2 dimension using TSNE using hue parameters as respective types

return $visualization\_window$

Chapter 4

Preparation and Extraction

This chapter provides a practical level idea about the implementations of the rule extraction module, detection and generation of leakage and generation of test set per pattern discussed in chapter 3. To grasp an idea about how the structure of the whole directory of **RULECT** looks like, we may observe the figure 4.1.

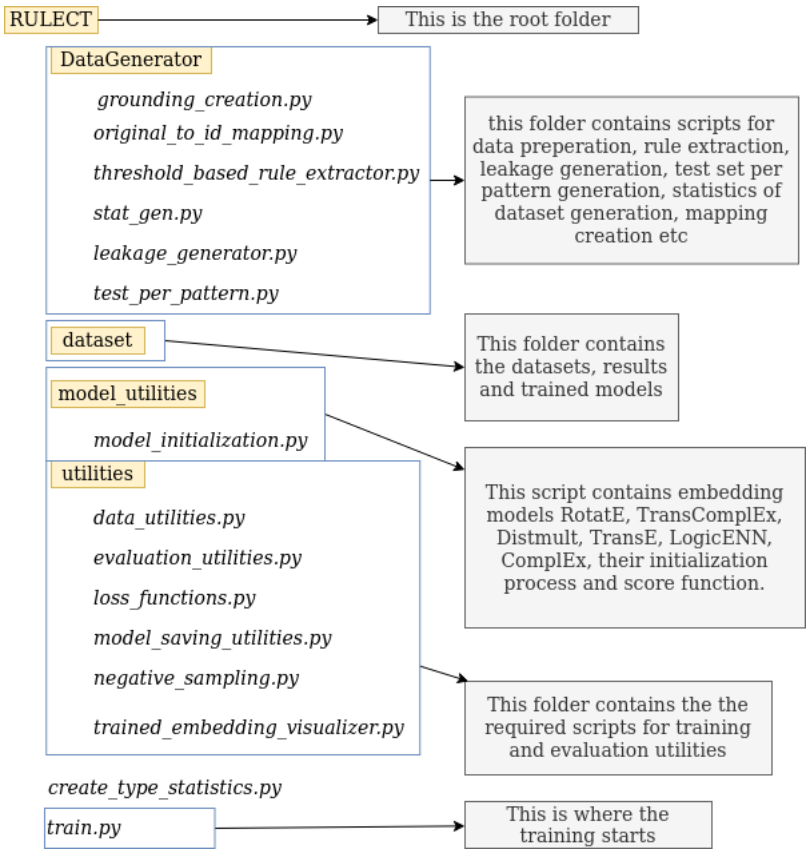


FIGURE 4.1: Folder Structure of the **RULECT** repository

4.1 Pattern Extraction from AMIE+

For obtaining the groundings AMIE+ [15] has been used as discussed in the chapter 3. AMIE+ requires triples as input which consists of (h, r, t). A training file of knowledge graph which contains training triples in (h, r, t) format has been provided as an input to AMIE. To have a glimpse of how an example of training triple looks like can be visualized in the figure 4.2.

```
/m/027rn    /location/country/form_of_government    /m/06cx9
/m/017dcd   /tv/tv_program/regular_cast./tv/regular_tv_appearance/actor /m/06v8s0
/m/07s9rl0  /media_common/netflix_genre/titles  /m/0170z3
/m/01sl1q   /award/award_winner/awards_won./award/award_honor/award_winner  /m/044mz_
/m/0cnk2q   /soccer/football_team/current_roster./sports/sports_team_roster/position    /m/02nzb8
/m/04nrcg   /soccer/football_team/current_roster./soccer/football_roster_position/position  /m/02sdk9v
/m/07nznf   /film/actor/film./film/performance/film /m/014lc_
/m/02qyp19  /award/award_category/nominees./award/award_nomination/nominated_for    /m/02d413
/m/0q9kd    /award/award_nominee/award_nominations./award/award_nomination/award_nominee    /m/0184jc
/m/03q5t    /music/performance_role/regular_performances./music/group_membership/role   /m/07y_7
/m/0gqng    /award/award_category/winners./award/award_honor/ceremony   /m/073hkh
/m/0b76d_m  /film/film/release_date_s./film/film_regional_release_date/film_release_distribution_medium /m/029j_
/m/014_x2   /award/award_winning_work/awards_won./award/award_honor/award_winner    /m/012ljv
/m/0ds35l9  /film/film/release_date_s./film/film_regional_release_date/film_release_region  /m/05r4w
/m/015qsq   /film/film/language /m/02bjrlw
/m/04bdxl   /award/award_nominee/award_nominations./award/award_nomination/award_nominee    /m/02s2ft
/m/09c7w0   /location/location/contains /m/0rs6x
/m/079vf    /film/actor/film./film/performance/film /m/0d90m
/m/015zyd   /organization/organization/headquarters./location/mailing_address/country   /m/09c7w0
/m/05vsxz   /award/award_nominee/award_nominations./award/award_nomination/award_nominee    /m/06qgvf
/m/04ljl_l  /award/award_category/nominees./award/award_nomination/nominated_for    /m/03qcfvw
/m/0grwj    /people/person/profession   /m/05sxg2
/m/0rh6k    /location/statistical_region/religions./location/religion_percentage/religion   /m/01lp8
/m/05d7rk   /award/award_nominee/award_nominations./award/award_nomination/award    /m/027dtxw
/m/05hs4r   /music/genre/artists    /m/01pbxb
/m/03qcq    /influence/influence_node/influenced_by /m/084w8
/m/01lxd4   /music/genre/artists    /m/0f0y8
/m/08815    /education/educational_institution/students_graduates./education/education/student  /m/05bnp0
/m/0g56t9t  /film/film/release_date_s./film/film_regional_release_date/film_release_region  /m/05r4w
/m/0160w    /organization/organization_member/member_of./organization/organization_membership/organization  /m/02vk52z
/m/0dbpyd   /people/person/place_of_birth   /m/09c7w0
```

FIGURE 4.2: Part of Freebase15k training triple as an input to AMIE

According to the instruction from the AMIE+'s website [1], the following command has to be issued in the command line as provided in figure 4.3.

```
Run this command in CommandLine to generate rules based on TSV

nohup java -XX:-UseGCOverheadLimit -Xmx8G -jar amie_plus.jar tsv_filename.tsv > file_to_save_output.txt

1. TSV file is needed to generate rules
2. Rule will be generated in a txt file which one shall have to filter using threshold_based_feature_extractor


Example command:
nohup java -XX:-UseGCOverheadLimit -Xmx8G -jar amie_plus.jar train.txt > wn18_all_rules.txt
```

FIGURE 4.3: AMIE instruction

[1] https://www.mpi-inf.mpg.de/departments/databases-and-information-systems/research/yago-naga/amie

The output of AMIE is a file which contains columns such as Rule, Head Coverage, Std Confidence, PCA Confidence, Positive Examples, Body size, PCA Body size, Functional variable, Std. Lower Bound, PCA Lower Bound, PCA Conf estimation.

4.2 Filtering Out Significant Rules

For filtering out the less significant rule, the *Std Confidence* level column has been considered. To achieve this, *threshold_based_rule_extractor.py* script from the *DataGenerator* folder has been utilized. This script contains a function where it takes the output of AMIE+ as an input and filters out less significant rules given a threshold (i.e., 0.80 or 80%). The function has been provided in figure 4.4. The function takes the input path, extracts rules with a given confidence range, and saves the rule file with only columns containing *Rule* and *Std Confidence*.

```
import numpy as np
import pandas as pd

def build_new_data(path):
    data=pd.read_table(path)
    data = data[(data['Std Confidence'] >= 0.8) & (data['Std Confidence']<=1.0)]
    rules = data.loc[:, data.columns.isin(['Rule', 'Std Confidence'])]
    rules = rules['Rule'].astype(str) + '    ' + rules['Std Confidence'].astype(str)
    #print(rules)
    #print(len(data))
    return rules

data_dir = '/media/mirza/Samsung_T5/PycharmProjects/frame_work/dataset/fb15k_new/mapped'
file_name = '/fb15k_train_rules_all_thresholded_'
data_new_data =build_new_data(data_dir + '/fb15k_all_rules.txt').reset_index()
data_new_data = data_new_data.drop(['index'], axis=1)
data_new_data.to_csv(data_dir+ file_name+'rule.tsv',index=False,sep='\t', header=None)

print('now go to grounding_creation.py')
```

This function takes **Rule** and **Std Confidence** column and filters significant rules based on threshold

FIGURE 4.4: Threshold based rule extraction script

These thresholded rules are used for further generation of the groundings. Rules of four patterns defined in equations 4.1, 4.2, 4.3, 4.4 are only considered further for implication, inverse, symmetric and equivalence grounding generation.

$$?a\ r_1\ ?b \Rightarrow ?a\ r_2\ ?b \tag{4.1}$$

$$?a\ r_1\ ?b \Rightarrow ?b\ r_2\ ?a \tag{4.2}$$

$$?a\ r\ ?b \Rightarrow ?b\ r\ ?a \tag{4.3}$$

?a r_1 ?b <=> ?a r_2 ?b (4.4)

4.3 Grounding Generation

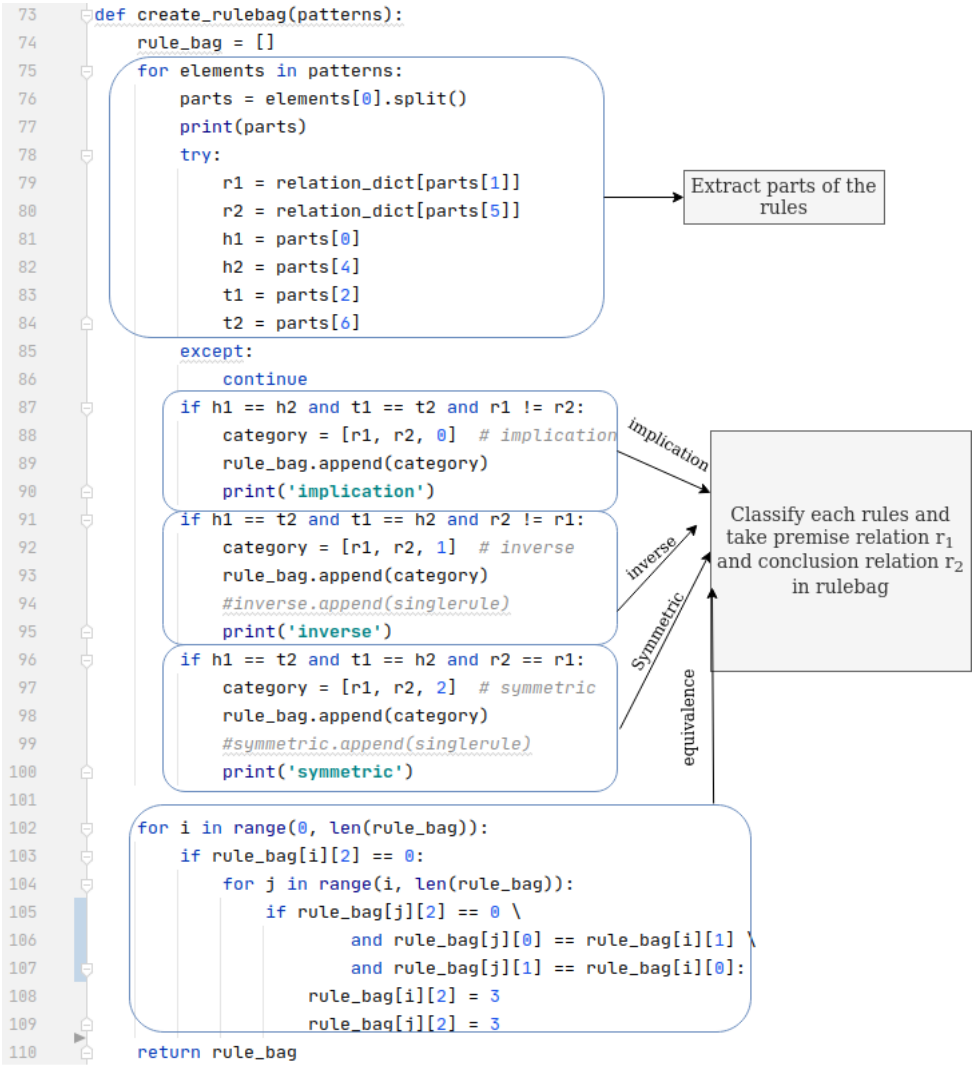
grounding_creation.py script from *DataGenerator* folder is responsible for generating the groundings from the thresholded rule file. Based on the algorithm provided in algorithm 3 of chapter 3, required groundings (implication, inverse, symmetric, and equivalence groundings) are generated. Three inputs are required for the grounding generation script: the thresholded rule file, training triples (from which the groundings are be generated), and the relation dictionary *relations.dict.* (which contains a unique numerical identifier for each relation).

The grounding generation algorithm (algorithm 3) has two parts which is separated as two different functions in the script: *create_rule_bag*(), and *ground_truth_generation*(). The *create_rule_bag*() function is responsible for the classifying and labeling the rules by looking at the conditional patterns. From the thresholded rule file, a rule-bag is generated where each of the row for the rule bag looks like [*premise* r_1, *conclusion* r_2, *rule type*]. The *create_rulebag*() function is illustrated in figure 4.5.

If we look into the details, this function takes the rule patterns (equations 4.1, 4.2, 4.3, 4.4) as input and splits them into their individual parts namely $(h_1, t_1, r_1, h_2, t_2, r_2)$. Eventually, the patterns are checked and the corresponding relations (r_1, r_2) (relation premise and relation conclusion) are labeled with the matched rule type. For labeling those rules (r_1, r_2) in the input patterns, numerical values of 0 to 4 (0-implication, 1-inverse, 2-symmetric, 3-equivalence) has been considered. Finally, the function returns the rule-bag as in the format $[r_1, r_2, label]$.

After obtaining the rule-bag (the output of *create_rulebag*() function), the function *ground_truth_generation*() has been called in order to generate the groundings. The calling of *ground_truth_generation*() function has been illustrated in figure 4.6.

The *ground_truth_generation*() function takes the rule-bag as input and the training triple; it eventually returns groundings of the four mentioned rules (implication, inverse, symmetric, and equivalence). After obtaining the required inputs, each of the training triples is traversed and checked whether the relation of the




```
def create_rulebag(patterns):
    rule_bag = []
    for elements in patterns:
        parts = elements[0].split()
        print(parts)
        try:
            r1 = relation_dict[parts[1]]
            r2 = relation_dict[parts[5]]
            h1 = parts[0]
            h2 = parts[4]
            t1 = parts[2]
            t2 = parts[6]
        except:
            continue
        if h1 == h2 and t1 == t2 and r1 != r2:
            category = [r1, r2, 0]  # implication
            rule_bag.append(category)
            print('implication')
        if h1 == t2 and t1 == h2 and r2 != r1:
            category = [r1, r2, 1]  # inverse
            rule_bag.append(category)
            #inverse.append(singlerule)
            print('inverse')
        if h1 == t2 and t1 == h2 and r2 == r1:
            category = [r1, r2, 2]  # symmetric
            rule_bag.append(category)
            #symmetric.append(singlerule)
            print('symmetric')

    for i in range(0, len(rule_bag)):
        if rule_bag[i][2] == 0:
            for j in range(i, len(rule_bag)):
                if rule_bag[j][2] == 0 \
                        and rule_bag[j][0] == rule_bag[i][1] \
                        and rule_bag[j][1] == rule_bag[i][0]:
                    rule_bag[i][2] = 3
                    rule_bag[j][2] = 3
    return rule_bag
```

FIGURE 4.5: Function for creating rule-bag

training triple matches with every premise (r_1) of the rule-bag. If it matches, then it further checks the label of the corresponding rule-bag and a condition. The condition is, whether $\{triple_{\text{head}}, triple_{\text{tail}}, r_2(conclusion's\ relation)\}$ already exists in the training triple or not (for implication and equivalence). For symmetric and inverse, the condition has been checked as $\{triple_{\text{tail}}, triple_{\text{head}}, r_2(conclusion's\ relation)\}$ exists or not among the training triple. Finally, if they do not reside in the training triple, then these are formed into a specific format, which is: [$triple_{\text{head}}$, $triple_{\text{tail}}$, $r_1(premise)$, $r_2(conclusion)$].

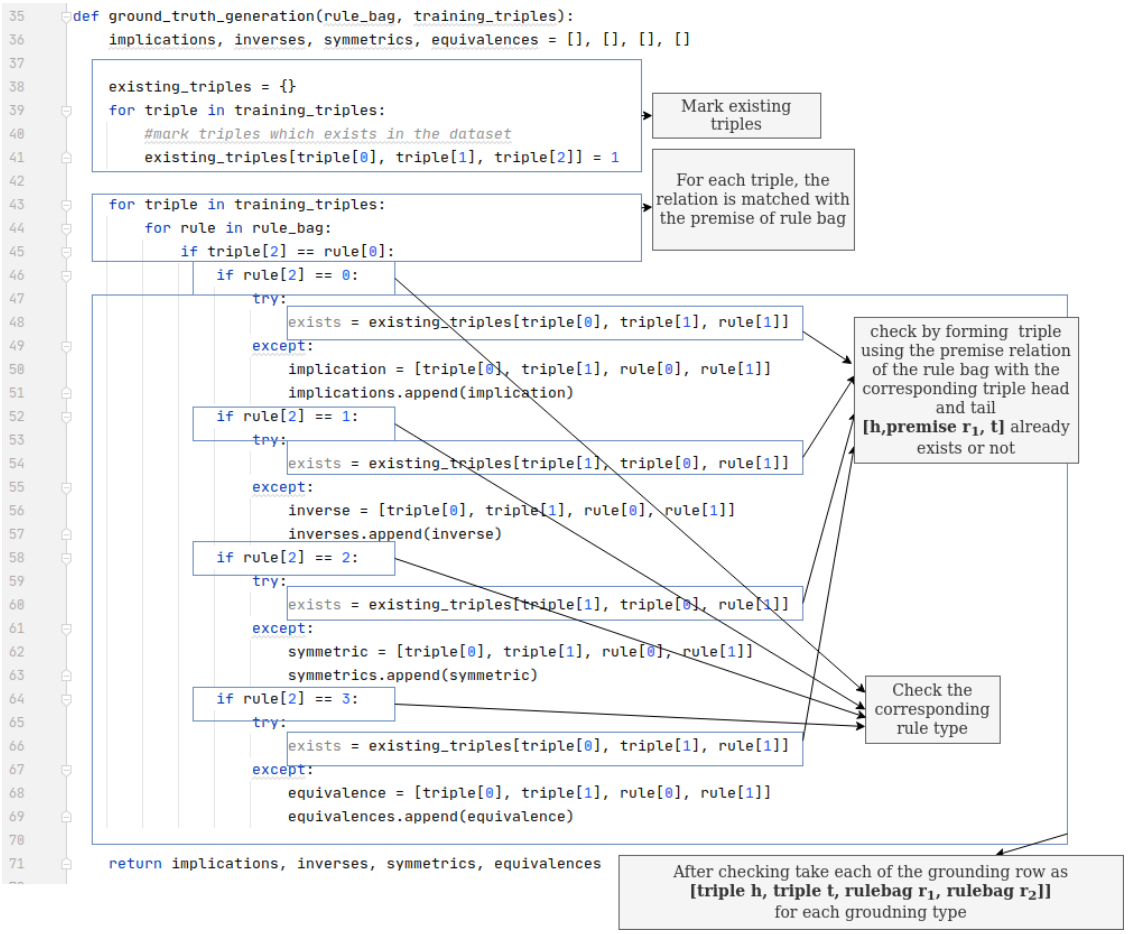


FIGURE 4.6: The function *grounding_generation*()

4.4 Removal of Leakage From Knowledge Graph Test Set

One of the core tasks of the **RULECT** system is to check whether rule injection helps if leakages are removed from the test set since creation of leakage-free version of test set is necessary for the second research question. To identify potential leakages in the test set, certain strategies are taken.

- Firstly, the conclusions are fetched from the respective grounding files as conclusion triples.
- The fetched conclusions are matched with each triple of the test data. If the conclusion matches then, it is considered as a potential leakage in the test set.

- For each of the grounding types, the matched conclusions with the test data are combined or unified.
- Finally, those unified conclusions are removed from the test data in order to obtain a refined test set.

For example, if any conclusion (t, r, h) in the test set exists for a particular premise (h, r, t) in the training set, it can be considered a symmetric leakage. These leakages are removed from the test set, which makes the link prediction task more challenging. This research has a vital part and it is to explore whether rule injection helps or not if we consider leakage-free test sets. Previously in other researches, from the training set leakages are removed. In this case, a discovery of the embedding model's inference power is to be identified on leakage-free test sets.

To be more specific, from grounding files, for a particular grounding pattern $[h, t, r_1, r_2]$, the conclusions are fetched. Depending on the grounding-type, the conclusion can be of two types. For implication and equivalence the conclusion is in the format of $\{h, r_2, t\}$. On the other hand, the inverse and symmetric conclusion is $\{t, r_2, h\}$. To detect potential leakages from the test set, *detect_leakage*() function from *leakage_detector.py* script has been called which is illustrated in figure 4.7.

This function takes a particular grounding data, the test data (which is to be refined), and the corresponding grounding type (each type of grounding files are saved separately) as input. It provides the particular leakages which are existing in the test set in return. After getting the individual leakages per rule type (implication, inverse, symmetric, and equivalence) with the help of the respective grounding files, they are unified. The unification can be seen in figure 4.8. Finally, those leakages were removed from the test set.

```python
def detect_leakage(grounding, test_triples, type = None):
    leakages = []
    conclusions = []
    #Take each triple from the groundings
    for triple in grounding:
        #Fetch the conclusions
        if type == 'implication' or type == 'equivalence':
            conclusion_head = triple[0]
            conclusion_relation = triple[3]
            conclusion_tail = triple[1]
        elif type == 'symmetric' or type == 'inverse':
            conclusion_head = triple[1]
            conclusion_relation = triple[3]
            conclusion_tail = triple[0]
        else:
            print('other groundings are not implemented')
            exit()
        #Form the conclusion
        conclusion = np.array([conclusion_head,conclusion_relation,conclusion_tail])
        conclusions.append(conclusion)
    conclusions = np.array(conclusions)
    if (grounding!=0).any():
        conclusions = np.unique(conclusions, axis=0)
        #Observe the existing triples in the test triples
    observed_triples = {}
    for triple in test_triples:
        head = triple[0]
        relation = triple[1]
        tail = triple[2]
        observed_triples[head,relation,tail]=1
    for triple in conclusions:
        head = triple[0]
        relation = triple[1]
        tail = triple[2]
        #If the conclusion matches any test triple
        if (head, relation, tail) in observed_triples:
            if (grounding!=0).any():
                leakage = np.array([head, relation, tail])
                leakages.append(leakage)
        else:
            continue
    return leakages
```

Fetch the conclusions

Fetch **(h,r,t)** from both test triples and conclusions

Return detected leakages by matching those conclusions with test set

FIGURE 4.7: Leakages detection from the test set

```python
#Unify all the leakages
all_leakages = [implication_leakages, inverse_leakages, symmetric_leakages, equivalence_leakages]

leakages_total = np.empty((0,3))
for individual_leakages in all_leakages:
    if len(individual_leakages) != 0:
        leakages_total = np.vstack([leakages_total, np.array(individual_leakages)])
leakages_total = np.array(leakages_total).astype(int)
```

FIGURE 4.8: Combining all implication, inverse, symmetric and equivalence leakages returned from *detect_leakage*() function

4.5 Statistics of Groundings and Leakages

The statistical information generated by *stat_gen.py* script has been provided in the table 4.1. This script is located in the *DataGenerator* folder. The general information, information of rules, grounding, and leakage of some popular knowledge graph datasets have been provided in this table. It is observed that inverse and symmetric groundings are mostly residing in FB15k and WN18. FB15k has mostly inverse grounding (#70427). Throughout this paragraph # denotes the number in integer. It contains significant leakages in the test set (total 72.57%), specifically, inverse (59.4%). On the other hand, FB15k-237 has rules and groundings, but it does not contain any leakage in the test set. WN18 contains only inverse and symmetric rules (#14 and #2) as well as groundings (7254 and 2092). This dataset mostly contains inverse and symmetric leakage (72.12%), and (20.98%). However, WN18RR [19] does not have any rules nor any groundings. Hence, leakages in the test set can be taken out of consideration. Kinship only contains inverse and symmetric groundings (#191 and #167). This dataset's test set contains a total leakage of 11.36%, including inverse (6.33%) and symmetric leakage (5.02%). Comparing to the other datasets, UMLS relatively contains a very less number of rules (#6 implication and #2 inverse) and groundings (#41 implication and #14 inverse groundings) with a total number of 2.72% leakage in the test set. Among all these mentioned datasets, FB15k and WN18 are the bigger ones to contain leakages in the test set. FB15k, FB15k-237, WN18, and WN18RR have significant numbers of entities. In terms of relation, FB15k contains the most among the mentioned KGs. In FB15k-237, many relations are removed due to their leakage patterns in the training set [16] [57], which is why there are no leakages to be seen in its respective test set. Authors of [19] removed these leakages from WN18RR.

Dataset	FB15k	FB15k-237	WN18	WN18RR	Kinship	UMLS
# of total triples	592213	310116	151442	93003	10666	6529
# of training data	483142	272115	141442	86835	8544	5216
# of test data	59071	20466	5000	3134	1074	661
# of validation data	50000	17535	5000	3034	1066	652
# of entities	14951	14541	40943	40943	104	135
# of relations	1345	237	18	11	25	46
# implication rule	97	9	0	0	0	6
# of inverse rule	351	12	14	0	9	2
# of symmetric rule	24	24	2	0	6	0
# of equivalence rule	68	2	0	0	0	0
# of implication grounding	4127	578	0	0	0	41
# of inverse grounding	70427	1516	7254	0	191	14
# of symmetric grounding	7740	7737	2092	0	167	0
# of equivalence grounding	8771	861	0	0	0	0
# of implication leakage	1369	0	0	0	0	12
% of implication leakage	2.3%	0%	0%	0%	0%	1.81%
# of inverse leakage	35089	0	3606	0	68	0
% inverse leakage	59.4%	0%	72.12%	0%	6.33%	0.9%
# of symmetric leakage	4216	0	1049	0	54	0
% of symmetric leakage	7.14%	0%	20.98%	0%	5.02%	0%
# of equivalence leakage	2197	0	0	0	0	0
% of equivalence leakage	3.72%	0%	0%	0%	0%	0%
# of total leakage	42871	0	4655	0	122	18
% of leakage	72.57%	0%	93.1%	0%	11.36%	2.72%
# triple in leakage free test set	18663	20466	345	3134	952	643

TABLE 4.1: General statistical, rules, grounding and leakage information of datasets

4.6 Creating Test Sets With Relational Patterns

To explore the third research question, test sets containing specific patterns are being created. Four different versions of test sets have been made for this purpose. Each of these test file contains relation specific patterns from the test set. For example, we can consider an implication rule if the premise $\{h, r_1, t\}$ and its respective conclusion $\{h, r_2, t\}$ exists in the training file. Based on the relation premises' occurrences for each specific pattern type, the relations have been sorted, and the top three are taken to be considered a medium to filter out the triples from the test set. To generate the version of test sets, we consider four types of patterns: symmetric, implication, inverse, and reflexive. Generation process for each of the test set per pattern is described below in brief.

Implication

- Combinations of all the pairwise relations (r_1, r_2) are taken into consideration.
- Formulation of premises using $\{h, r_1, t\}$ has been done. It has been checked that, for the premise $\{h, r_1, t\}$, conclusions $\{h, r_2, t\}$ exists or not in the training file.
- Topmost participating relation premises r_1 are taken as a list of significant candidates.
- In the test set, only triples with this list of significant candidate's relations (r_1) are kept. All the others are dropped.

Inverse

- Combination of all the pairwise relations (r_1, r_2) are taken into consideration (similar to implication pattern).
- Formulation of premises using $\{h, r_1, t\}$ has been done. It has been checked that, for the premise $\{h, r_1, t\}$, conclusions $\{t, r_2, h\}$ exists or not in the training set.
- Topmost participating relation premise r_1 are taken as a list of significant candidates.
- In the test set, only triples with this list of significant candidate's relations r_1 are kept. All the others are dropped.

Symmetric

- A checking has been done for each of the relation r, whether there exists any $\{h, r, t\} \implies \{t, r, h\}$ in the training set.
- Top three participating relations r for symmetric pattern has been considered.
- Only triples with these r are kept in the test set.

Reflexive

- It has been checked whether triples exist with the same head and tail ($\{h, r, t\}$ where, $h = t$) in the training set.
- Top three participating relations r for reflexive pattern has been considered.
- Only triples with these r are kept in the test set.

In order to generate these test set per patterns, three different functions have been developed. One function is responsible for generating implication and inverse test pattern ($generate\_implication\_inverse\_test\_pattern()$). The other two functions are for symmetric ($generate\_symmetric\_test\_pattern()$) and reflexive ($generate\_reflexive\_test\_pattern()$) test patterns. These functions are residing in *test_per_pattern.py* script in *DataGenerator* folder. Figure 4.9, illustrates code section of the creation of implication and inverse test patterns. The first function takes an input of training triples and the type of the pattern (implication and inverse). Depending on the type provided, it is checked for each pair of relations whether implication or inverse patterns exist in the training set or not. Then it takes the topmost occurring premises r_1, for which implication or inverse patterns are existing. Then using those r_1 only participating triples are filtered from the test set. Finally, the output *test_triple_pattern* is returned where only the triples containing such relation are existing.

For the reflexive pattern generation in the test set, topmost r, which participates in reflexive relation $\{e, r, e\}$ where $\{h, t\} \in e$ are only taken, and triples containing these r are only kept in the test set. Function $generate\_reflexive\_test\_pattern()$ has been illustrated in figure 4.11, which demonstrates the process of creating such test set which only consists of reflexive relational pattern. Figure 4.10, has been used to generate symmetric patterns, as discussed above. For symmetric test pattern. it is checked that, for each (h, r, t), symmetric conclusion (t, r, h), exists in the training file or not. Only the three most appearing relations based on reflexive pattern frequency are taken in the candidate relation list. Finally, to get the test set's symmetric version, only triples containing these relations from the candidate relation list are kept only in the test set as discussed previously.

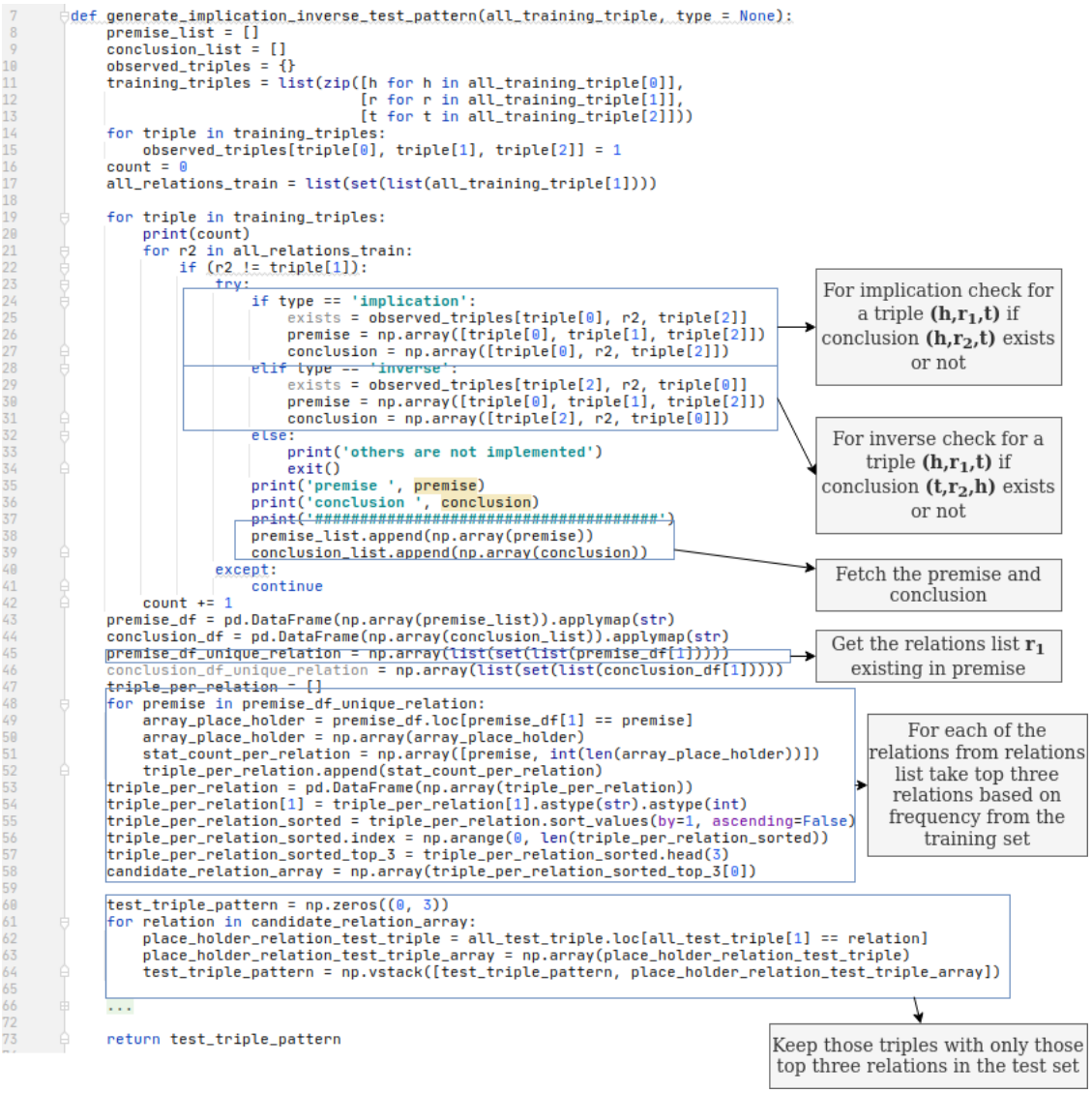


FIGURE 4.9: Test triple generation only containing implication and inverse pattern

FIGURE 4.10: Test triple generation only containing symmetric test pattern

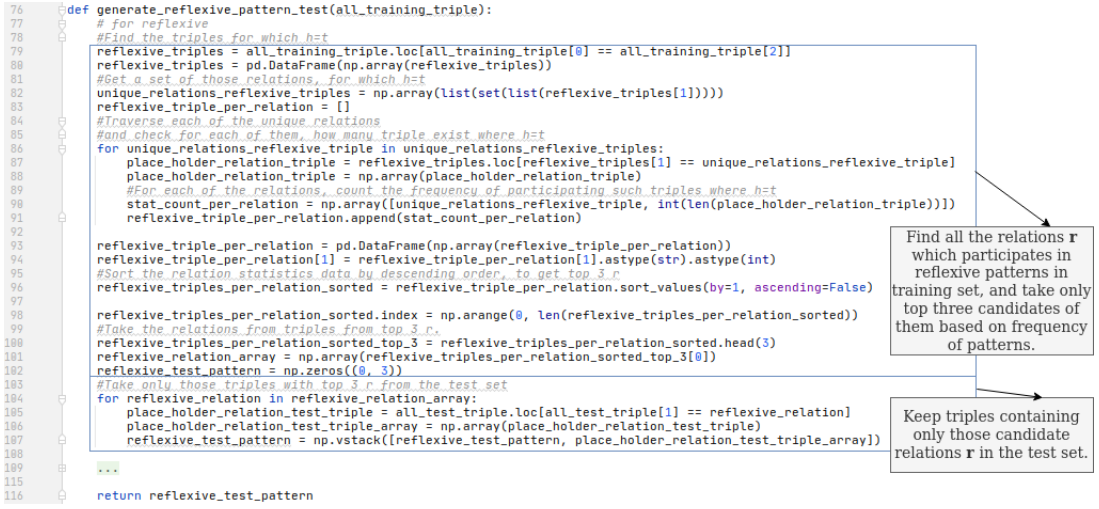


FIGURE 4.11: Test triple generation only containing reflexive test pattern

Chapter 5

Learning Relational Patterns

In this chapter, the implementation of the whole training process for the embedding models is illustrated. This chapter covers the practical level details of training and evaluation with or without injection of the groundings. Each of the training processes, such as initialization, selection of appropriate score function, and grounding injection, has been provided in detail.

Throughout the whole research, Github repository ATISE[1] by Chengjin Xu (soledad921 (github username)) has been utilized, especially for the implementation part of *RULECT*. This repository contains the implementation of [58] and [59]. For the different parts of the implementation, plenty of unpublished codes of Chengjin Xu[2] and Mojtaba Nayyeri[3] have been used. Codes from OpenKE [48][4] and RotatE [4][5] have also been used to initialize various embedding models and their score function creation. The code of LogicENN [14] has been utilized through out the whole research for implementation purpose, specially in the grounding generation, rule injection, training and evaluation parts. The Github repository of PyKeen [47][6] has been helpful in terms of different aspects such as mapping generation.

[1] https://github.com/soledad921/ATISE
[2] https://scholar.google.com/citations?user=sIts5VgAAAAJ&hl=zh-CN
[3] https://scholar.google.com/citations?user=X785350AAAAJ&hl=en
[4] https://github.com/thunlp/OpenKE
[5] https://github.com/DeepGraphLearning/KnowledgeGraphEmbedding
[6] https://github.com/pykeen/pykeen

Attribute name	**Value**	**Description**
data_dir	directory path (string)	data directory
name	embedding model name (string)	name of the embedding model
lr(α)	learning rate (float)	the learning rate of the training
negsample_num(C)	integer	the number of negative triple per positive
gamma(γ)	float/integer	the maximum margin
lam(λ / ξ_{im})	float/integer	constant multiplier for rule loss constant multiplier for implication rule loss
lam2(ξ_{inv})	float/integer	constant multiplier for inverse rule loss
lam3(ξ_{sym})	float/integer	constant multiplier for symmetric rule loss
lam4(ξ_{eq})	float/integer	constant multiplier for equivalence rule loss
test_mode	boolean (true/false)	whether to only test the trained model. In this case one has to import trained model.
saving	boolean (true/false)	whether to save trained model or not.
regul	boolean (true/false)	model training with regularization or not
train_with_groundings	boolean (true/false)	whether to inject rule/grounding or not
max_epoch	integer	the maximum number of epoch for training
dim(d)	integer	the embedding dimension for entities and relation
batch_size(β)	integer	the size of the minibatches
gpu	boolean (true/false)	Whether to use GPU or not
remove_grounding_inv	boolean (true/false)	whether conversion to be done for inverse grounding to implication.
remove_grounding_sym	boolean (true/false)	whether conversion to be done for symmetric grounding to inverse.
solver	name of the optimizer (string)	name of the optimizer should be specified
rev_set	binary (0/1)	whether to input reverse triple or not
fix_layer	boolean (true/false)	whether to use fix first layer for LogicENN or not
additional_model	boolean (true/false)	whether dual model for LogicENN to be initialized or not

TABLE 5.1: Information of the attributes for calling the training function

5.1 Starting the KGE Model's Training Process

To learn relational patterns, we start from the *train.py* which resides in the root folder of this **RULECT** project. The *train*() function has been called with the required parameters. This procedure has been illustrated in figure 5.1. The attribute for calling the *train*() function has been briefly described in table 5.1.

After the training function has been called, training, testing, and validation files from their respective dictionary specified in the parameter. If training with grounding has been selected then, the four types of grounding files are also read from the specified directory (the generation of these grounding files has been briefly discussed in chapter 4).

FIGURE 5.1: call to the training function *train*() with required parameters and model name

5.2 Model Initialization

After calling the *train*() function with specific attributes and reading the required files, the model initialization has been done. In this part, the initialization of the specified model with required parameters is demonstrated. This model initialization has been specified in *model_initialization.py* which is residing inside *model_utilities* folder. In this script, the embedding model's initialization process and their respective score function are defined. Embedding model's internal parameters are set by creating a *base_model* object. A particular model's initialization has been performed after calling the *init_embedding*() function while initializing the class attributes. The implementation of the model initialization process is demonstrated in figure 5.2. In the model initialization part, the embedding space of entities (emb_E) and relations (emb_R) are initialized. For some models, imaginary parts are present (RotatE, TransCompEx, complEx) as discussed in chapter 3. That is why emb_E_real, emb_E_im corresponds to the real and imaginary parts of entity embedding. emb_R_real, and emb_R_im corresponds to the imaginary embeddings of relations. For RotatE, phase initialization of relation is required which is defined by emb_R_phase. For LogicENN three additional layers has been defined $fc1$, $fc2$ and $fc3$ as done by the authors of [14].

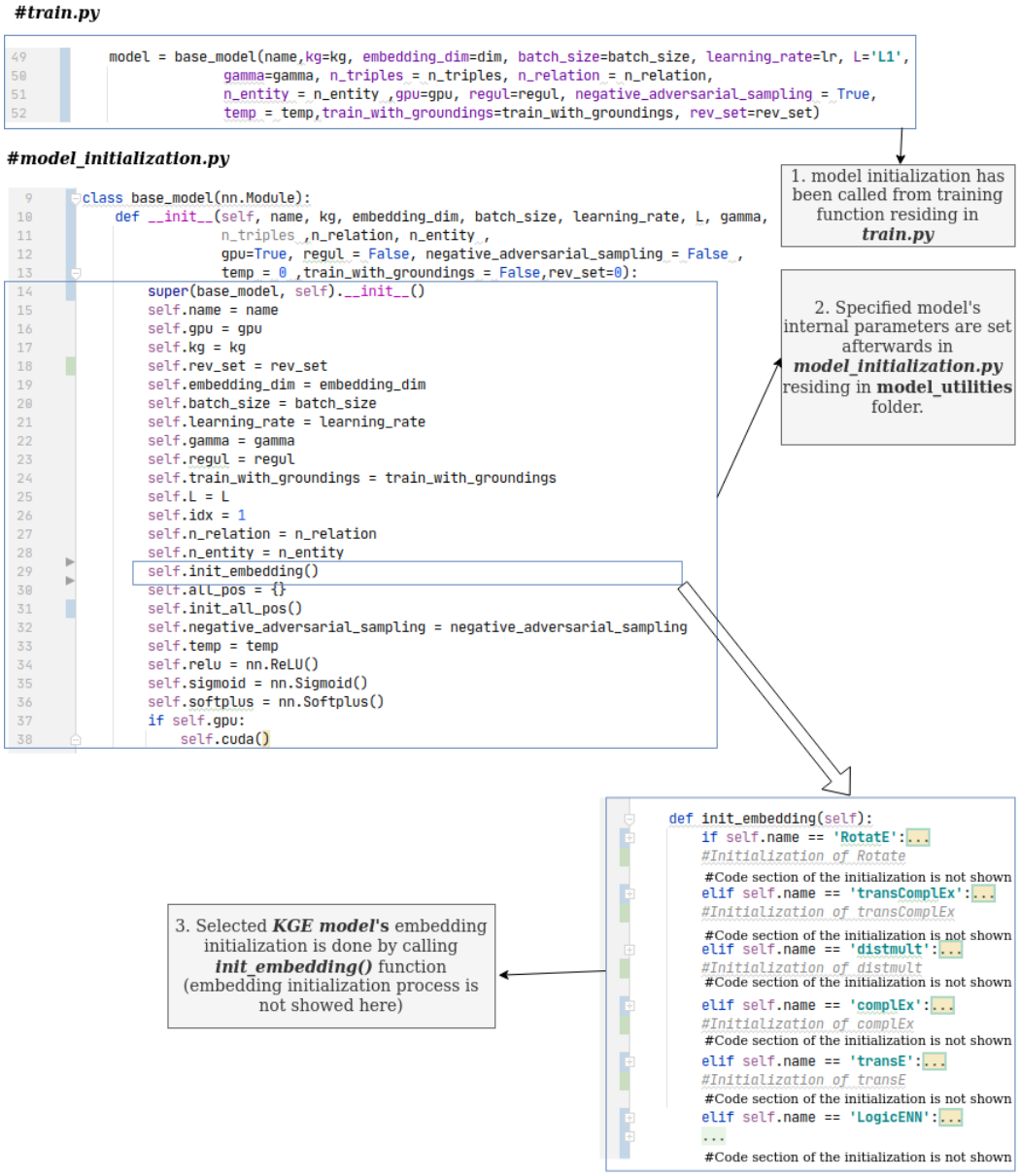


FIGURE 5.2: Embedding model initialization and setting its internal attributes

5.3 Fetching Required Grounding

The required grounding files are read from the respective directory before starting the iterative training process. Any case of empty grounding files (this happens only when particular groundings are not present for any rule) is carefully handled. The fetching of groundings and the start of iterative training is demonstrated in figure 5.3.

FIGURE 5.3: Fetching groundings and start of iterative training process

5.4 Iterative Training

The starting of the training process, fetching the positive triples, negative triples, and groundings in mini-batches, are demonstrated in figure 5.3. The loss for training triples is obtained by the generic training procedure of KGE models. At the beginning of each iteration, the training file and the grounding files per rule types are split into mini-batches. Then for each positive triple in the minibatch, C

negative triples are generated randomly by corrupting either head or tail. The positive triples and negative triples are called $iter\_triple$ and $iter\_neg$, respectively. There are also other negative sampling techniques which have been integrated into **RULECT** system, which are: Distributional Negative Sampling for Knowledge Base Completion [60] and Affinity Dependent Negative Sampling for Knowledge Graph Embeddings [61][7]. These negative sampling methods are currently out of the scope of this research. Hence, only random negative sampling is used for training. Afterward, the initialized model's $forward()$ function has been called on both positive ($iter\_triple$) and negative ($iter\_neg$) triples. The $forward()$ function extracts the head h, tail t and relation r from triples. The $calculate\_score()$ function is called from the inside of $forward()$ function as illustrated in figure 5.4. Then, the calculation of the score is performed for both $iter\_triple$ and $iter\_neg$. $calculate\_score()$ function is responsible for calculating the score of a particular KGE model. The overall process of calculating the positive and negative score is demonstrated in figure 5.4. It is shown in figure 5.4 that, *head*, *tail* and *relation* are passed to the $calculate\_score()$, which checks the models name and calculates score of triples (both positive and negative) by utilizing the score function of that KGE model. For the sake of demonstrating the entire process of obtaining the scores of triples, the respective KGE model's score calculations are not shown in figure 5.4. After getting the positive score (pos_score) and negative scores (neg_score) per batch, a particular loss function (in this case adversarial loss [4]) is called from the script $loss\_functions.py$ which resides in the *utilities* folder.

The next steps of iterative training deal with rule injection process. To inject rules, the grounding losses per rule types are obtained from the mini-batches of the groundings. The $rule\_loss\_calculation()$ function (illustrated in figure 5.6) from $loss\_functions.py$ script is responsible for calculating the individual losses for the groundings per mini-batch. The inputs of this function are: the groundings in the format $[h, t, r_1, r_2]$ for all grounding types and a constant variable ξ for each. This function returns individual grounding losses per rule types based on algorithm 4, which is discussed in chapter 3. The selected model's score function is used for calculating the premise and conclusion score which are taken from the grounding files. The score of the premise for all grounding loss calculation is similar in terms of structure. For conclusion, two different structured inputs for the same function are responsible for calculating rule losses due to the swapped positions of head(h) and tail(t) between implication, equivalence with symmetric, inverse. Equation

[7]Author of this thesis is the first author of this paper.

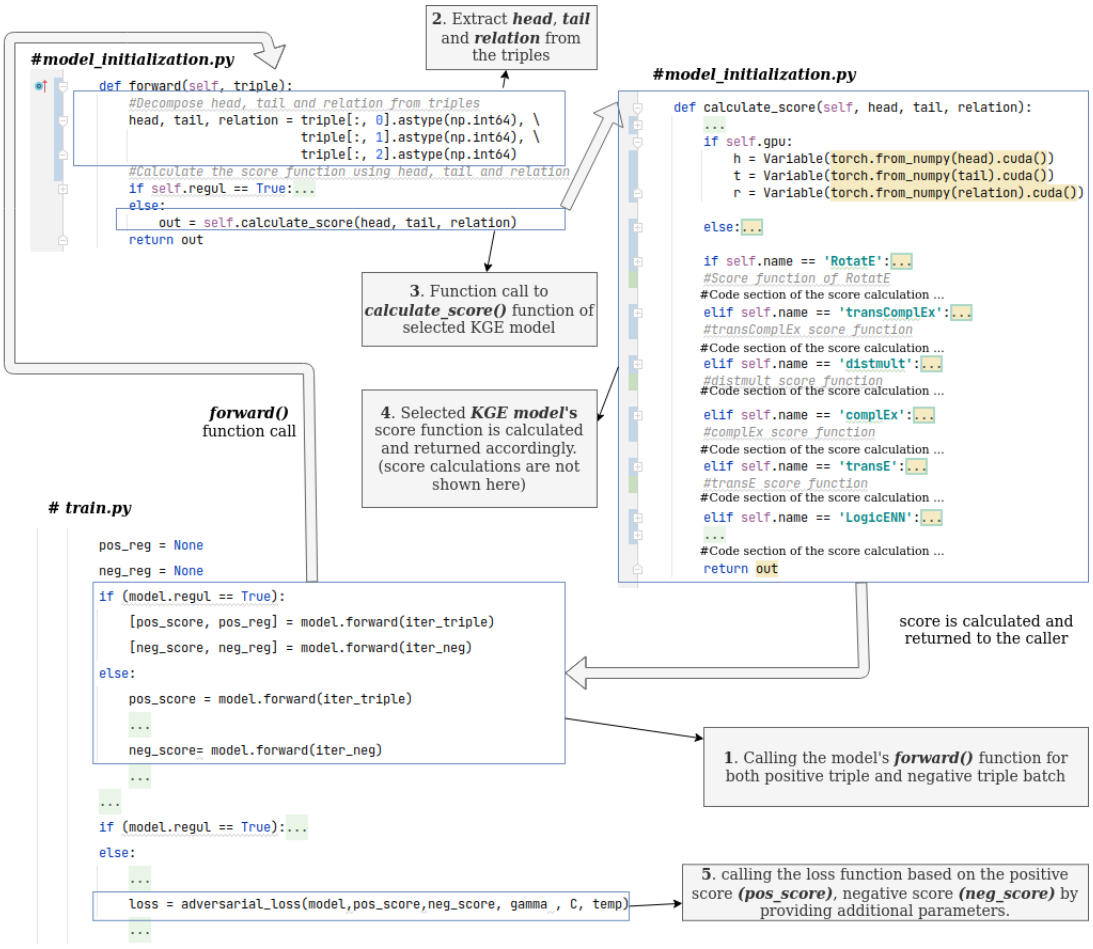


FIGURE 5.4: Call to the score function of particular KGE model

5.1 provides the premise's score function for all the grounding types, which is defined by the score function f of the selected KGE model. The output scores for implication and equivalence groundings are defined in equation 5.2. Additionally, equation 5.3 provides the output score for the conclusion of inverse and symmetric grounding types.

$$output\_premise_{\text{imp,inv,sym,equ}} = f_{\text{model}}(h, r_1, t) \tag{5.1}$$

$$output\_conclusion_{\text{imp,equ}} = f_{\text{model}}(h, r_2, t) \tag{5.2}$$

$$output\_conclusion_{\text{inv,sym}} = f_{\text{model}}(t, r_2, h) \tag{5.3}$$

In order to inject rule, the mini-batched groundings are passed to the $rule\_loss\_calculation()$ function which resides in $loss\_functions.py$ script.

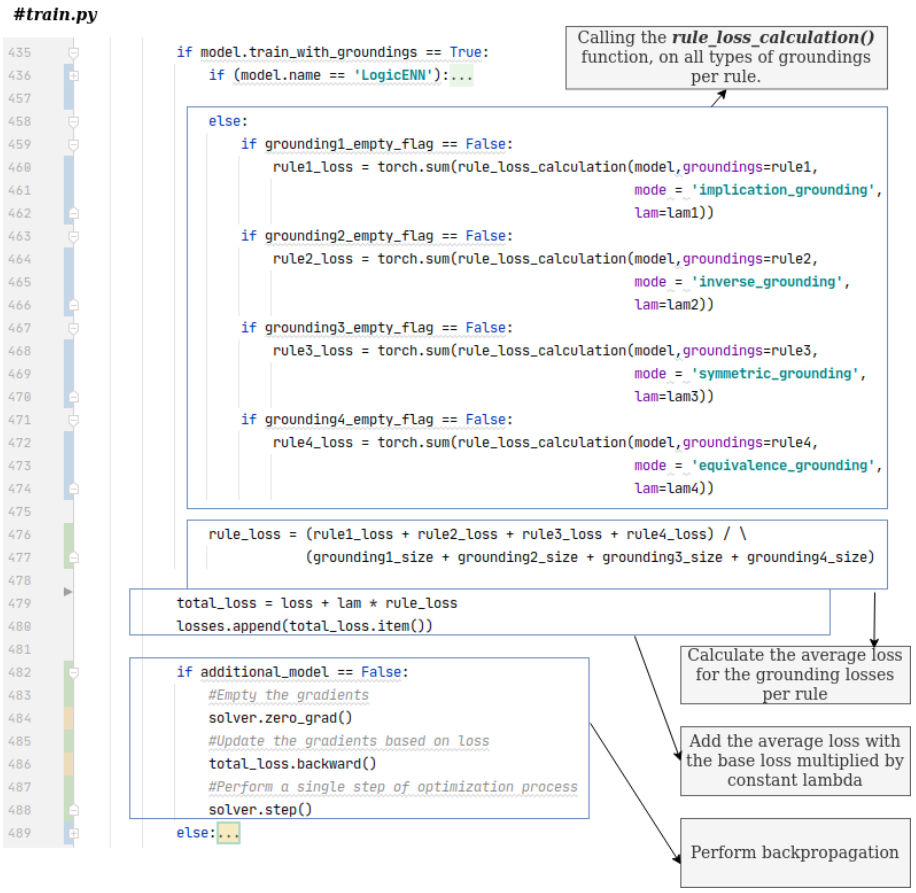


FIGURE 5.5: Injecting rule losses with base loss

After all the grounding losses are calculated, they are averaged and added to the base loss multiplied by a constant λ. Figure 5.5 shows the rest of the training where grounding loss calculation and backpropagation is done.

One of the significant parts of this research is grounding loss generation for each rule type. This has been discussed in algorithm 4 from chapter 3. Grounding loss calculation per rule types is done by calling $rule\_loss\_calculation()$ function. Now the implementation of this function is illustrated in figure 5.6.

$rule\_loss\_calculation()$ function (which is responsible for calculating the losses for each grounding type) is residing in $loss\_functions.py$ script. As already discussed in algorithm 4, the input of this function are: a defined KGE model which has a score function f, grounding data, grounding type and a constant ξ. In this script, at first the grounding data is decomposed to head(h), tail(t), premise relation(r_1) and conclusion relation(r_2). out_1 and out_2 are the score of the premise and conclusion respectively. Based on the previous discussion, in the conclusion the h and t is reversed for inverse and symmetric relation. Which is why

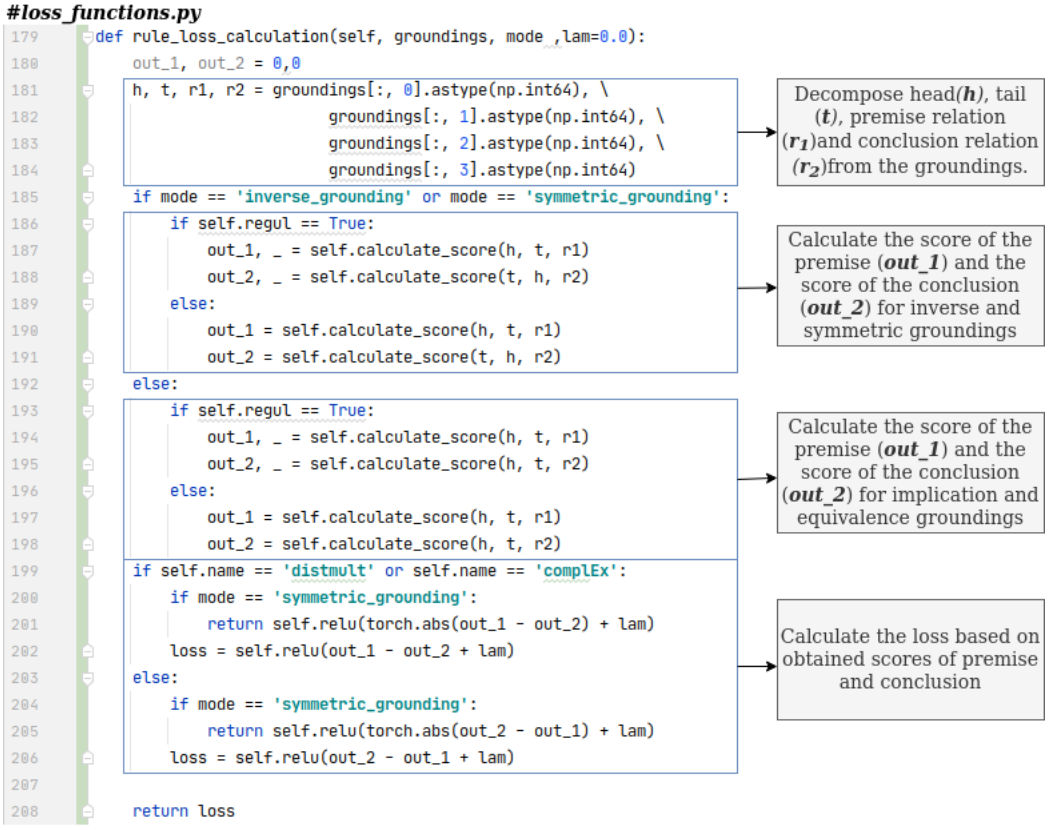


FIGURE 5.6: Function responsible for calculating grounding losses per rule type

there are different modes wrapped in *if else* condition. In the scope of the first *if* condition, the section code checks the parameter *mode*. *mode* contains the type of the grounding file in string format. Depending on the type between *inverse*, *symmetric* and *equivalence*, *implication*, a particular condition executes. Whenever *out_1* (premise output) and *out_2* (conclusion output) are calculated, based on the constrains discussed in chapter 3, grounding loss is calculated.

5.5 LogicENN Extensions in *RULECT*

As discussed in the previous chapter, the LogicENN extensions have been integrated into the system. Chapter 3 provides an idea of removal of groundings (can be seen theoretically in the subsection 3.6.1 of chapter 3). The two parameters are required at the beginning of the training process. Firstly, the removal of inverse grounding and conversion to implication is done. Secondly, symmetric groundings are removed and converted to equivalence grounding. To achieve this, depending on the value of *remove_grounding_inv* and *remove_grounding_sym* Boolean

flag, the mentioned groundings are converted and removed. According to the logic discussed in subsection 3.6.1 of chapter 3, inverse grounding is converted to implication and symmetric has been converted to equivalence.

Fixed layer implementation has been integrated as discussed in the subsection 3.6.2 of chapter 3. This process has been illustrated in figure 5.7. If the fix_layer flag has been set to $True$, then the first layer $fc1$ has been uniformly initialized between -1 and 1. The torch $requires\_grad$ variable for weight and bias is set to be $False$, which makes this layer to not learn anything. The only parameters meant for learning are provided in the optimizer function. The whole process has been illustrated in figure 5.7.

```
solver = solver
fix_layer = fix_layer
if model.name == 'LogicENN' and fix_layer == True:
    model.fc1.weight.data.uniform_(-1, 1)
    model.fc1.weight.requires_grad = False
    model.fc1.bias.requires_grad = False
    learning_layers = filter(lambda i: i.requires_grad,
                             model.parameters())
    if solver == 'adagrad':
        solver = torch.optim.Adagrad(learning_layers, model.learning_rate)
    elif solver == 'adam':
        solver = torch.optim.Adam(learning_layers, model.learning_rate)
    elif solver == 'adamW':
        solver = torch.optim.AdamW(learning_layers, model.learning_rate)
    else:
        print('only adam and adagrad solvers are available at the moment')
```

FIGURE 5.7: Fixed layer implementation

The third extension has been notably considered using the double model for LogicENN. One model is required for original groundings, and one model is required for reverse grounding (discussed in subsection 3.6.3 of chapter 3). Gathering the reverse triple has been taken from implementation of obtaining the reverse triple from the authors of LogicENN [14]. Initializing the dual model has been provided in 5.8.

```
162     model = base_model(name,kg=kg, embedding_dim=dim, batch_size=batch_size, learning_rate=lr, L='L1',
163                        gamma=gamma, n_triples = n_triples, n_relation = n_relation,
164                        n_entity = n_entity ,gpu=gpu, regul=regul, negative_adversarial_sampling = True,
165                        temp = temp,train_with_groundings=train_with_groundings)
166     #exit()
167     second_model = None
168     second_solver = None
169     if additional_model == True:
170         second_model = base_model(name, kg=kg, embedding_dim=dim, batch_size=batch_size, learning_rate=lr, L='L1',
171                                   gamma=gamma, n_triples=n_triples, n_relation=n_relation,
172                                   n_entity=n_entity, gpu=gpu, regul=regul, negative_adversarial_sampling=True,
173                                   temp=temp, train_with_groundings=train_with_groundings)
```

FIGURE 5.8: Initializing dual model

Chapter 6

Analysis and Utilization

This chapter presents the experimental results addressing the research questions. In this chapter, at first, the evaluation metrics and the hyperparameters are explained is section 6.1. Then the result and analysis are presented which is addressed by each of the research questions in section 6.2. Subsection 6.2.1 shows the effect of rule injection in KGE models, subsection 6.2.2 shows the effect of rule injection on leakage-free test sets. Another subsection 6.2.3 in section 6.2 shows the effect of rule injection on test set per pattern. Later parts namely the section 6.3 demonstrates the visualization of an extra module added to the system, which is about visualizing trained embeddings.

In this section the experimental setup and the evaluation has been done. Three key aspects is presented in the following sections:

- Experimental result of the mentioned KGE models in the system has been provided, which consists of results before and after injection of rules, addressing the first research question.
- Model evaluation has been done using leakage-free test sets since one aim is to identify whether rule injection helps if leakage-free test set is considered. This evaluation addresses the second research question.
- Evaluation result with different test sets per pattern(discussed in details in chapters 3 and 4) has been performed in order to identify whether rule injection helps if only such patterns of test sets are considered. This addresses the third research question.

6.1 Experimental Setup

6.1.1 Evaluation Metrics

Standard evaluation metrics for link prediction are used for evaluating the models. These metrics are: mean rank (MR), mean reciprocal rank (MRR), hit@1, hit@3, hit@5, and hit@10. Test triples are ranked after corrupting each test triple (either with head or tail) with all the possible entities. The position of a particular true test triple against all the corrupted ones is considered as its rank. In filtered settings, the candidate test triples do not appear in training, test, or validation triples [4]. On the other hand, it may or may not appear in train, test, or validation set in raw settings. If the position of a particular true test triple comes on top against all the other corresponding corrupted test triples, then it is considered as a hit@1. For the other metrics such as hit@3, hit@5, and hit@10, the candidate test triple has to appear among the top 3, 5, or 10.

6.1.2 Hyperparameter settings

For the fairness of the experiments, the hyperparameters are kept the same for rule injected embedding model and no-rule injected model. All the models are evaluated in terms of standard link prediction evaluation metrics: mean rank (MR), mean reciprocal rank (MRR), hit@1, hit@3, hit@5, and hit@10.

For FB15k, for models (RotatE, TransComplEx, Distmult, TransE and ComplEx) the learning rate is α=0.1, rule loss multiplier is $\lambda = 0.01$, implication rule loss constant $\xi_{\text{Im}} = 0.01$, inverse rule loss constant $\xi_{\text{In}} = 0.1$, symmetric rule loss constant $\xi_{\text{Sym}} = 0.01$, and equivalence rule loss constant $\xi_{\text{Eq}} = 0.1$, the embedding dimension is $d = 200$, the number of negative sample is C=10, margin is $\gamma = 30$ (for all models except TransE), γ=10 (for TransE), temperature is τ=0.0 and the epoch number is E=50. LogicENN uses a learning rate α=0.001. For the LogicENN model, to obtain better results, the inverse grounding and symmetric grounding have been removed and converted to implication and equivalence. The reverse triple parameter *rev_set* is set to True to include reverse triples for the LogicENN model. the learning rate α is 0.001 for LogicENN. For RotatE, TransComplEx, and TransE $L1$ norm has been used, and the *regul* parameter

is set to False. For ComplEx and Distmult regularization term has been added to achieve better results; in this case, the *regul* parameter is set to True. For RotatE, TransComplEx, TransE and LogicENN the *regul* parameter is by default set to False in this system. While initializing these models, a parameter is passed whether the $L1$ or $L2$ norm has to be selected. For ComplEx and Distmult, the *regul* parameter needs to be set to true value to select the respective regularization as a separate calculation. For ComplEx and Distmult, the combination of logistic loss function [5] and self-adversarial negative sampling [4] has been used as the loss function. For LogicENN same loss function has been used, but without regularization, and at the end of each epoch, normalization of embeddings has been done. For TransE, TransComplEx, and RotatE, the adversarial loss [4] has been used. In case of RotatE, TransE, TransComplEx, ComplEx and Distmult adaptive gradient descent optimizer (Adagrad) has been used. For the optimization of LogicENN Adam optimizer has been utilized. For FB15k-237, every setting mentioned above kept the same except for LogicENN. In LogicENN for FB15k-237, the optimizer is AdamW [62] and the epoch E is set to 100. For WN18 dataset, similar settings have been used for all the models. For LogicENN and WN18 dataset, the epoch E is kept at 200, no reverse settings (rev_set=False) has been used, and no rule conversion has been done.

The parameter for Kinship dataset are: learning rate α=0.1, embedding dimension d=200, the number of negative sample C=40. rule loss multiplier $\lambda = 0.01$, inverse rule loss constant $\xi_{\text{In}} = 0.1$, symmetric rule loss constant $\xi_{\text{Sym}} = 0.01$ (since, here only symmetric and inverse groundings are present). These settings are the same for all models for the kinship dataset. The epoch E=100 for all models, except TransComplEx (E=200). The margin γ is set to 24 for RotatE. γ=15 is set for TransComplEx, ComplEx, LogicENN, and Distmult. For TransE γ=10 is set. For LogicENN and RotatE, temperature τ is set to 0.01. For ComplEx, Distmult, TransComplEx, TransE temperature τ is set to 0. For RotatE, ComplEx, TransComplEx, Distmult, and TransE Adagrad optimizer has been used. On the other hand, For LogicENN, AdamW optimizer has been used as mentioned previously. Reverse settings rev_set has been turned to False for all the models except LogicENN. We used the converted groundings (inverse to implication, symmetric to equivalence) settings for LogicENN to obtain better results. Similar Loss functions have been used as mentioned for FB15k, FB15k-237, and WN18.

For UMLS dataset, all the models have a learning rate α=0.1, except LogicENN,

which has a learning rate λ=0.001. The negative sampling C=50 is kept for all the mentioned models. The embedding dimension is: d=200 for RotatE, TransComplEx, Distmult, ComplEx and TransE. For RotatE, the embedding dimension is d=100. The temperature τ=0.0 is set for all the models. The batch size β=1000 is kept for all the models except LogicENN. Batch size β for LogicENN is set at the value 2750. Similar loss functions, as previously mentioned, are used for all the models. For All the models except LogicENN, no grounding conversion has been done, and no reverse settings, namely rev_set is set to False. For LogicENN inverse grounding is converted to implication. UMLS dataset has only grounding for implication and inverse rule. The inverse rule loss constant ξ_{In} is kept at as previously 0.1 and symmetric rule loss constant ξ_{Sym} is set to 0.01 similar to other datasets. The rule loss multiplier is set to, $\lambda = 0.01$.

6.2 Result and Analysis

6.2.1 Evaluation With Versus Without Rule Injection

In this section, it is observed that whether rule injection improves the performance of embedding models or not. From table 6.1 shows that injection of rule improves the performance of embedding models. Table 6.1 illustrates that RotatE, ComplEx and TransComplEx have significant improvements in embedding model performance in terms of almost all performance metrics (MR, MRR, Hit@1, Hit@3, Hit@5, and Hit@10) for both Raw and Filtered settings. The filtered Hit@1, Hit@3, Hit@5 and Hit@10 for RotatE have been improved from 0.5061, 0.7255, 0.7801, 0.8385 to 0.6016, 0.8004, 0.8381, 0.8758 respectively. For TransComplEx, filtered Hit@3 has been improved from 0.7181 to 0.7466, filtered Hit@1 becomes from 0.5762 to 0.6052, and filtered Hit@5 becomes from 0.8141 to 0.8326. The complEx also has much improvement after injection of rules, especially in Hit@1, Hit@3 and Hit@5. The other models also have a slight gain in performance improvement for the FB15k dataset.

Table 6.2 shows the effect of rule injection in FB15k-237 dataset. It can be observed that on this dataset, rule injection has very less effect on performance. Even in some cases, performance deteriorates slightly (especially in RotatE, TransComplEx, TransE, and LogicENN) across several performance metrics. Only ComplEx

Model	MR		MRR		Hit@1		Hit@3		Hit@5		Hit@10	
	Raw	Filtered	Raw	Filtered	Raw	Filtered	Raw	Filtered	Raw	Filtered	Raw	Filtered
RotatE	202	48	0.2556	0.6324	0.1387	0.5061	0.2888	0.7255	0.3854	0.7801	0.5152	0.8385
RotatE (Injection)	166	46	0.3020	0.7116	0.1813	0.6016	0.3465	0.8004	0.4392	0.8381	0.5590	0.8758
ComplEx	279	125	0.2566	0.6958	0.1428	0.5983	0.2896	0.7741	0.3837	0.8072	0.5100	0.8415
ComplEx (Injection)	243	117	0.2875	0.7455	0.1711	0.6643	0.3276	0.8113	0.4201	0.8355	0.5390	0.8610
Distmult	259	115	0.2651	0.6205	0.1497	0.4912	0.3014	0.7191	0.3947	0.7772	0.5181	0.8351
Distmult (Injection)	240	113	0.2810	0.6508	0.1653	0.5282	0.3200	0.7466	0.4104	0.8008	0.5319	0.8489
TransComplEx	187	41	0.2798	0.6851	0.1628	0.5762	0.3160	0.7677	0.4115	0.8141	0.5389	0.8606
TransComplEx (Injection)	159	40	0.3030	0.7092	0.1856	0.6052	0.3433	0.7910	0.4350	0.8326	0.5573	0.8746
TransE	162	53	0.2643	0.4478	0.1564	0.3252	0.2960	0.5119	0.3788	0.5844	0.4934	0.6770
TransE (Injection)	160	54	0.2672	0.4511	0.1590	0.3285	0.3005	0.5173	0.3805	0.5889	0.4958	0.6806
LogicENN	276	146	0.2702	0.5134	0.1549	0.3740	0.3062	0.5973	0.3961	0.6892	0.5217	0.7854
LogicENN (Injection)	286	157	0.2675	0.5115	0.1534	0.3740	0.3034	0.6000	0.3936	0.6897	0.5179	0.7867

TABLE 6.1: Evaluation of rule injected model versus no rule injected model on FB15k dataset

Model	MR		MRR		Hit@1		Hit@3		Hit@5		Hit@10	
	Raw	Filtered	Raw	Filtered	Raw	Filtered	Raw	Filtered	Raw	Filtered	Raw	Filtered
RotatE	384	183	0.1614	0.2846	0.0959	0.1942	0.1632	0.3118	0.2145	0.3781	0.3004	0.4681
RotatE (Injection)	383	183	0.1611	0.2810	0.0956	0.1906	0.1631	0.3081	0.2124	0.3719	0.3016	0.4643
ComplEx	936	721	0.0410	0.1072	0.0169	0.0691	0.0335	0.1127	0.0465	0.1355	0.0738	0.1627
ComplEx (Injection)	935	721	0.0418	0.1094	0.0175	0.0706	0.0335	0.1159	0.0483	0.1394	0.0751	0.1664
Distmult	937	724	0.0563	0.1366	0.0242	0.0844	0.0494	0.1467	0.0722	0.1828	0.1132	0.2333
Distmult (Injection)	927	714	0.0576	0.1382	0.0249	0.0855	0.0512	0.1494	0.0726	0.1846	0.1149	0.2354
TransComplEx	352	185	0.1690	0.2622	0.1029	0.1765	0.1744	0.2836	0.2220	0.3467	0.3467	0.4387
TransComplEx (Injection)	351	185	0.1697	0.2577	0.1045	0.1706	0.1743	0.2811	0.2209	0.3432	0.3046	0.4344
TransE	347	172	0.1687	0.2687	0.1011	0.1802	0.1753	0.2949	0.2246	0.3569	0.3079	0.4455
TransE (Injection)	348	174	0.1691	0.2650	0.1025	0.1760	0.1752	0.2901	0.2224	0.3550	0.3074	0.4463
LogicENN	887	680	0.0963	0.1588	0.0486	0.0910	0.0928	0.1685	0.1260	0.2170	0.1910	0.2987
LogicENN (Injection)	890	683	0.0898	0.1513	0.0417	0.0823	0.0849	0.1605	0.1202	0.2120	0.1890	0.2952

TABLE 6.2: Evaluation of rule injected model versus no rule injected model on FB15k-237 dataset

and Distmult is showing slight improvement in performance. For example, for ComplEx, the Hit@10 improves slightly from 0.1627 to 0.1664 in filtered settings.

Table 6.3 illustrates that rule injection has slight improvement margin on WN18 dataset. For RotatE and ComplEx, it can be observed that rule injection improvement in raw settings and slight improvements in filtered settings. For example, in the case of RotatE, Hit@10 improves from 0.8150 to 0.8352 in raw settings, and Hit@10 increases from 0.9567 to 0.9584 in filtered settings. For TransComplEx, performance is decreased after rule injection in almost all the metrics. For TransE, performance slightly decreases upon rule injection. From the Distmult model's evaluations, it is seen that rule injection improves the performance by a tiny amount of margin. LogicENN has a good effect of rule injection in this dataset since there are observable improvements in performance after rule injection for LogicENN and WN18 dataset. For LogicENN Hit@1 has been improved from 0.8059 to 0.8848, Hit@5 has been improved from 0.8889 from 0.9429 and Hit@10 has been improved from 0.9223 to 0.9458 in filtered settings. In raw settings, significant performance improvement can be seen.

Table 6.4 shows the effect of rule injection on Kinship dataset. It can be observed that rule injection improves all the embedding models' performance in terms of

Model	MR		MRR		Hit@1		Hit@3		Hit@5		Hit@10	
	Raw	Filtered	Raw	Filtered	Raw	Filtered	Raw	Filtered	Raw	Filtered	Raw	Filtered
RotatE	353	338	0.5914	0.9453	0.4711	0.9386	0.6691	0.9500	0.7456	0.9531	0.8150	0.9567
RotatE (Injection)	339	328	0.6029	0.9463	0.4794	0.9398	0.6834	0.9504	0.7623	0.9537	0.8352	0.9584
ComplEx	862	850	0.5971	0.9435	0.4812	0.9417	0.6718	0.9445	0.7477	0.9454	0.8159	0.9462
ComplEx (Injection)	801	791	0.6008	0.9434	0.4844	0.9413	0.6755	0.9447	0.7539	0.9457	0.8231	0.9471
Distmult	651	638	0.5028	0.8091	0.3471	0.6947	0.6048	0.9202	0.7078	0.9332	0.8006	0.9453
Distmult (Injection)	658	646	0.5094	0.8177	0.3548	0.7111	0.6112	0.9202	0.7123	0.9340	0.8043	0.9457
TransComplEx	286	274	0.5094	0.7882	0.3531	0.6887	0.6088	0.8760	0.7130	0.9193	0.8119	0.9469
TransComplEx (Injection)	246	233	0.4464	0.6669	0.2922	0.5245	0.5227	0.7855	0.6516	0.8728	0.7844	0.9278
TransE	242	230	0.3323	0.4811	0.0416	0.0925	0.5629	0.8650	0.6912	0.9221	0.8067	0.9466
TransE (Injection)	217	204	0.3231	0.4719	0.0466	0.1008	0.5340	0.8366	0.6718	0.9131	0.8003	0.9444
LogicENN	481	467	0.5644	0.8514	0.4426	0.8059	0.6423	0.8889	0.7189	0.9062	0.7945	0.9223
LogicENN (Injection)	487	480	0.6996	0.9117	0.5976	0.8848	0.7791	0.9371	0.8342	0.9429	0.8773	0.9458

TABLE 6.3: Evaluation of rule injected model versus no rule injected model on WN18 dataset

Model	MR		MRR		Hit@1		Hit@3		Hit@5		Hit@10	
	Raw	Filtered	Raw	Filtered	Raw	Filtered	Raw	Filtered	Raw	Filtered	Raw	Filtered
RotatE	11	2	0.1422	0.7788	0.0154	0.6555	0.0824	0.8827	0.1853	0.9469	0.5349	0.9767
RotatE (Injection)	11	2	0.1530	0.7956	0.0233	0.6853	0.0968	0.8901	0.2058	0.9437	0.5587	0.9823
ComplEx	12	3	0.1343	0.7265	0.0121	0.5940	0.0722	0.8366	0.1760	0.9055	0.4986	0.9562
ComplEx (Injection)	11	3	0.1417	0.7385	0.0149	0.6024	0.0857	0.8506	0.1867	0.9162	0.5331	0.9669
Distmult	14	6	0.1232	0.4513	0.0158	0.3012	0.0791	0.4977	0.1504	0.6196	0.3822	0.8203
Distmult (Injection)	14	6	0.1269	0.4539	0.0186	0.2970	0.0796	0.5047	0.1606	0.6280	0.3957	0.8315
TransComplEx	10	4	0.1939	0.5387	0.0526	0.3482	0.1681	0.6550	0.2924	0.7993	0.5926	0.9385
TransComplEx (Injection)	10	4	0.1930	0.5401	0.0480	0.3450	0.1723	0.6667	0.2947	0.8031	0.5978	0.9437
TransE	15	9	0.1317	0.2607	0.0079	0.0196	0.0982	0.3901	0.1997	0.5442	0.4516	0.7556
TransE (Injection)	14	8	0.1340	0.2657	0.0098	0.0228	0.1043	0.3976	0.1941	0.5689	0.4520	0.7584
LogicENN	11	2	0.1497	0.8148	0.0191	0.7104	0.1010	0.9064	0.2034	0.9483	0.5456	0.9772
LogicENN (Injection)	11	2	0.1491	0.8108	0.0186	0.7025	0.1015	0.9055	0.2016	0.9516	0.5480	0.9818

TABLE 6.4: Evaluation of rule injected model versus no rule injected model on Kinship dataset

most of the performance metrics. Especially in case of filtered Hit@1 for RotatE which has an improvement from 0.6555 to 0.6853. For ComplEx filtered Hit@3 improved from 0.8366 to 0.8506.

Model	MR		MRR		Hit@1		Hit@3		Hit@5		Hit@10	
	Raw	Filtered	Raw	Filtered	Raw	Filtered	Raw	Filtered	Raw	Filtered	Raw	Filtered
RotatE	19	1	0.1183	0.8892	0.0121	0.8079	0.0809	0.9667	0.1596	0.9856	0.3714	0.9955
RotatE (Injection)	18	1	0.1239	0.8791	0.0174	0.7905	0.0870	0.9637	0.1611	0.9924	0.3790	0.9977
ComplEx	21	4	0.0999	0.5802	0.0076	0.3926	0.0590	0.7194	0.1195	0.8253	0.2844	0.9395
ComplEx (Injection)	20	4	0.1055	0.6111	0.0091	0.4365	0.0719	0.7383	0.1278	0.8525	0.2995	0.9402
Distmult	23	7	0.0900	0.4667	0.0061	0.3079	0.0492	0.5499	0.0983	0.6483	0.2390	0.8011
Distmult (Injection)	23	6	0.1001	0.4850	0.0113	0.3275	0.0658	0.5620	0.1165	0.6762	0.2693	0.8124
TransComplEx	18	1	0.1203	0.8869	0.0151	0.8185	0.0840	0.9470	0.1664	0.9796	0.3684	0.9955
TransComplEx (Injection)	18	1	0.1300	0.8833	0.0212	0.8116	0.0976	0.9440	0.1747	0.9750	0.3722	0.9955
TransE	15	9	0.1317	0.2607	0.0079	0.0196	0.0982	0.3901	0.1997	0.5442	0.4516	0.7556
TransE (Injection)	14	8	0.1340	0.2657	0.0098	0.0228	0.1043	0.3976	0.1941	0.5689	0.4520	0.7584
LogicENN	18	1	0.1341	0.9115	0.0250	0.8585	0.1021	0.9614	0.1762	0.9818	0.3858	0.9939
LogicENN (Injection)	18	1	0.1363	0.9109	0.0280	0.8563	0.1036	0.9622	0.1793	0.9773	0.3880	0.9947

TABLE 6.5: Evaluation of rule injected model versus no rule injected model on UMLS dataset

Table 6.5, provides an illustration of the effect of rule injection on UMLS dataset. From the result, it is observable that rule injection improves all the embedding models' performance. It is observable that Hit@3 improved from 0.7194 to 7383 for ComplEx after injection in filtered settings. For Distmult Hit@3 has an improvement from 0.5499 to 0.5620 in filtered settings.Hit@1 also has some significant

improvement in filtered settings, especially for ComplEx and Distmult. Improvement in performance for Hit@5 is also noticeable for ComplEx (from 0.8253 to 0.8525) and Distmult (from 0.6483 to 0.6762).

6.2.2 Evaluation With Versus Without Rule Injection on Leakage Free Test Set

This subsection provides the evaluation of the previously trained models on leakage-free test sets. The process and the idea of leakage removal from the test set have already been discussed in section 4.4 at chapter 4. Underlying leakage in the test set makes the link prediction task easier. Previously a portion of this research [6], showed the injection of rules improves the performance of RotatE and TransE, and in case of the removal of leakages from the test set, rule injection does not help that much. A comprehensive evaluation of the previously trained models on the mentioned datasets is performed on leakage free test sets in this subsection.

Model	MR		MRR		Hit@1		Hit@3		Hit@5		Hit@10	
	Raw	Filtered	Raw	Filtered	Raw	Filtered	Raw	Filtered	Raw	Filtered	Raw	Filtered
RotatE	316	147	0.2305	0.3979	0.1409	0.2824	0.2540	0.4593	0.3158	0.5290	0.4134	0.6126
RotatE (Injection)	317	148	0.2311	0.3937	0.1418	0.2781	0.2541	0.4543	0.3183	0.5229	0.4138	0.6085
ComplEx	563	392	0.2023	0.3615	0.1263	0.2666	0.2178	0.4074	0.2720	0.4664	0.3609	0.5395
ComplEx (Injection)	545	374	0.2027	0.3614	0.1257	0.2657	0.2175	0.4071	0.2738	0.4676	0.3622	0.5436
Distmult	529	361	0.2081	0.3461	0.1300	0.2451	0.2241	0.3941	0.2807	0.4572	0.3716	0.5394
Distmult (Injection)	529	360	0.2083	0.3466	0.1299	0.2458	0.2246	0.3924	0.2800	0.4584	0.3722	0.5416
TransComplEx	294	126	0.2524	0.4255	0.1665	0.3211	0.2744	0.4793	0.3344	0.5441	0.4294	0.6226
TransComplEx (Injection)	295	127	0.2542	0.4242	0.1695	0.3199	0.2742	0.4780	0.3355	0.5401	0.4288	0.6207
TransE	236	110	0.2602	0.3773	0.1707	0.2711	0.2848	0.4240	0.3493	0.4928	0.4411	0.5837
TransE (Injection)	238	112	0.2580	0.3754	0.1689	0.2688	0.2824	0.4221	0.3449	0.4932	0.4402	0.5820
LogicENN	584	421	0.2022	0.3075	0.1214	0.2070	0.2218	0.3487	0.2786	0.4184	0.3670	0.5087
LogicENN (Injection)	548	386	0.2055	0.3129	0.1244	0.2126	0.2249	0.3535	0.2801	0.4232	0.3729	0.5152

TABLE 6.6: Evaluation of rule injected model versus no rule injected model on FB15k dataset on leakage-free test set

Table 6.6 shows the evaluation result of the effect of rule injection if leakages are removed from the test set of FB15k. It is observed that, for RotatE, TransComplEx, and TransE, previously trained rule injected models do not perform well (no improvement in performance) on the FB15k dataset. On the other hand, rule injected models perform slightly better on ComplEx, Distmult, and LogicENN. FB15k-237 does not have any leakages in the test set, which is why no such evaluation has been done for FB15k-237. Generally, all the previously trained models perform worse compared to original test sets on leakage-free test sets (in the case of both rule injected and no rule injected trained models).

Model	MR		MRR		Hit@1		Hit@3		Hit@5		Hit@10	
	Raw	Filtered	Raw	Filtered	Raw	Filtered	Raw	Filtered	Raw	Filtered	Raw	Filtered
RotatE	4910	4891	0.1908	0.2620	0.1130	0.2043	0.2261	0.2884	0.2913	0.3275	0.3348	0.3768
RotatE (Injection)	4753	4734	0.1845	0.2570	0.1000	0.1928	0.2246	0.2884	0.2971	0.3304	0.3507	0.3971
ComplEx	12331	12312	0.1135	0.1779	0.0609	0.1551	0.1536	0.1870	0.1826	0.2014	0.1971	0.2116
ComplEx (Injection)	11466	11447	0.1094	0.1743	0.0551	0.1493	0.1464	0.1812	0.1841	0.2000	0.1986	0.2246
Distmult	9239	9220	0.1299	0.1968	0.0710	0.1652	0.1696	0.2043	0.2072	0.2261	0.2275	0.2522
Distmult (Injection)	9357	9339	0.1178	0.1856	0.0536	0.1493	0.1580	0.2029	0.2029	0.2246	0.2290	0.2536
TransComplEx	3970	3951	0.1241	0.1605	0.0464	0.0739	0.1406	0.1899	0.2101	0.2478	0.2913	0.3275
TransComplEx (Injection)	3352	3334	0.1292	0.1640	0.0522	0.0797	0.1493	0.1986	0.2072	0.2478	0.2957	0.3246
TransE	3323	3304	0.0705	0.0872	0.0014	0.0043	0.0652	0.1043	0.1449	0.1870	0.2478	0.2768
TransE (Injection)	2941	2922	0.0727	0.0899	0.0029	0.0043	0.0710	0.1101	0.1406	0.1855	0.2493	0.2826
LogicENN	6630	6611	0.0972	0.1379	0.0493	0.1058	0.1159	0.1464	0.1536	0.1652	0.1841	0.1928
LogicENN (Injection)	6964	6945	0.1057	0.1497	0.0551	0.1101	0.1333	0.1696	0.1681	0.1928	0.2014	0.2188

TABLE 6.7: Evaluation of rule injected model versus no rule injected model on WN18 dataset on leakage-free test set

Model	MR		MRR		Hit@1		Hit@3		Hit@5		Hit@10	
	Raw	Filtered	Raw	Filtered	Raw	Filtered	Raw	Filtered	Raw	Filtered	Raw	Filtered
RotatE	11	2	0.1414	0.7606	0.0158	0.6324	0.0751	0.8687	0.1849	0.9380	0.5431	0.9737
RotatE (Injection)	11	2	0.1419	0.7721	0.0158	0.6523	0.0793	0.8724	0.1875	0.9354	0.5389	0.9800
ComplEx	12	3	0.1328	0.7072	0.0116	0.5704	0.0678	0.8188	0.1754	0.8955	0.5026	0.9506
ComplEx (Injection)	12	3	0.1327	0.7132	0.0084	0.5693	0.0741	0.8277	0.1691	0.9039	0.5173	0.9627
Distmult	15	6	0.1199	0.4257	0.0142	0.2705	0.0720	0.4690	0.1460	0.5993	0.3824	0.8072
Distmult (Injection)	15	6	0.1212	0.4280	0.0152	0.2700	0.0725	0.4722	0.1528	0.5972	0.3813	0.8146
TransComplEx	10	4	0.1928	0.5231	0.0536	0.3314	0.1644	0.6397	0.2889	0.7847	0.5914	0.9322
TransComplEx (Injection)	11	4	0.1860	0.5209	0.0415	0.3246	0.1654	0.6444	0.2852	0.7847	0.5893	0.9359
TransE	15	9	0.1332	0.2582	0.0084	0.0194	0.1003	0.3797	0.2059	0.5394	0.4575	0.7542
TransE (Injection)	14	9	0.1341	0.2614	0.0095	0.0226	0.1056	0.3866	0.1991	0.5583	0.4496	0.7489
LogicENN	11	2	0.1482	0.7912	0.0184	0.6770	0.0951	0.8887	0.2027	0.9401	0.5525	0.9743
LogicENN (Injection)	11	2	0.1453	0.7856	0.0163	0.6660	0.0914	0.8881	0.1980	0.9443	0.5499	0.9795

TABLE 6.8: Evaluation of rule injected model versus no rule injected model on Kinship dataset on leakage-free test set

Table 6.7 shows the effect of rule injection on leakage-free test sets on WN18. The overall performance metrics have been decreased after using leakage-free test sets drastically for the trained models. As already said these models has been trained for the evaluation of previous subsection namely subsection 6.2.1. For some models, rule injection slightly improves performance measures if the test set has been replaced with a leakage-free version. It can be observed that For RotatE in WN18 dataset rule injection helps in terms of Hit@5 (filtered hit@10 increased from 0.3275 to 0.3304) and Hit@10 (filtered hit@10 increased from 0.3768 to 0.3971). For other models such as ComplEx, Distmult, and TransE, rule injection enables slightly better performance for previously trained rule injected models. TransComplEx provides slightly worse performance in terms of Hit@10.

For the Kinship dataset, slight improvement can be seen according to table 6.8. RotatE, ComplEx, Distmult, TransComplEx, TransE, and LogicENN showed slightly better performance metrics in Hit@10. In terms of Hit@1 in filtered settings, RotatE only showed better performance (Hit@1 increased from 0.6324 to 0.6523). In terms of Hit@3 filtered settings, all the models except LogicENN performed slightly better than non-rule injected models on leakage-free test sets. In the case of Hit@5, filtered settings, only ComplEx and LogicENN performed slightly better.

Model	MR		MRR		Hit@1		Hit@3		Hit@5		Hit@10	
	Raw	Filtered	Raw	Filtered	Raw	Filtered	Raw	Filtered	Raw	Filtered	Raw	Filtered
RotatE	18	2	0.1195	0.8796	0.0124	0.7916	0.0816	0.9658	0.1610	0.9844	0.3764	0.9953
RotatE (Injection)	18	2	0.1209	0.8479	0.0148	0.7348	0.0824	0.9603	0.1571	0.9922	0.3771	0.9977
ComplEx	18	2	0.1214	0.8651	0.0156	0.7784	0.0848	0.9448	0.1680	0.9790	0.3733	0.9953
ComplEx (Injection)	18	2	0.1261	0.8515	0.0179	0.7543	0.0918	0.9417	0.1695	0.9743	0.3678	0.9953
Distmult	23	7	0.0905	0.4652	0.0062	0.3056	0.0490	0.5474	0.0980	0.6509	0.2411	0.8040
Distmult (Injection)	23	7	0.0962	0.4808	0.0070	0.3219	0.0599	0.5544	0.1120	0.6788	0.2667	0.8165
TransComplEx	18	2	0.1214	0.8651	0.0156	0.7784	0.0848	0.9448	0.1680	0.9790	0.3733	0.9953
TransComplEx (Injection)	18	2	0.1261	0.8515	0.0179	0.7543	0.0918	0.9417	0.1695	0.9743	0.3678	0.9953
TransE	19	3	0.1063	0.467	0.0039	0.0420	0.0708	0.8966	0.1369	0.9619	0.3437	0.9891
TransE (Injection)	19	3	0.1034	0.4554	0.0023	0.0350	0.0661	0.8725	0.1314	0.9432	0.3336	0.9899
LogicENN	17	1	0.1357	0.9002	0.0257	0.8390	0.1034	0.9565	0.1781	0.9813	0.3911	0.9938
LogicENN (Injection)	18	1	0.1376	0.9015	0.0288	0.8406	0.1034	0.9603	0.1796	0.9767	0.3935	0.9946

TABLE 6.9: Evaluation of rule injected model versus no rule injected model on UMLS dataset on leakage-free test set

Table 6.9 shows the effect of rule injection on leakage-free test sets of UMLs dataset. It is observed that RotatE, Distmult, TransE, and LogicENN gained slight improvement in performance in terms of Hit@10 filtered settings. Hit@5 improved for RotatE and Distmult. Hit@1 decreased for all the models after rule injection.

6.2.3 Evaluation With Versus Without Rule Injection on Test Set per Pattern

In this subsection, the evaluation of the above trained models (both rule injected and no rule injected trained models) is demonstrated on test sets of specific patterns. Test set generation considering such patterns is briefly described in section 4.6 of chapter 4. Table 6.10, 6.11, 6.12, 7.1 (in Appendix) and 7.2 (in Appendix) demonstrates the previously trained model's performance on the test set considering specific pattern, namely implication, inverse, symmetric and reflexive test pattern. The investigation is done to determine whether rule injection makes the link prediction task easier for such test sets or not, specifically for investigating the third research question. It turns out that the embedding models are capable of performing well enough on such test sets even without rule injection. Upon careful observation from comparing the result from tables of subsection 6.2.1 with this subsection (6.2.3), it is clear that the same trained models perform well on test sets with specific patterns compared to original test set. It means that models are trained in such a way so that they become capable of identifying those patterns. Additionally, rule injection makes these models learn those patterns more easily.

Table 6.10 demonstrates that all the model's prediction performance increases if the test set consisting of such patterns are only considered on FB15k. On top

of that, rule injected models show an additional boost in the performance. For implication test patterns, rule injection helps RotatE, ComplEx, TransE, Distmult, and LogicENN. The same things can be said for inverse patterns for these models except TransE and LogicENN in this dataset. For symmetric, the performance increases in RotatE, ComplEx and LogicENN. For reflexive patterns, each model performs well whether rule injection is performed or not.

Result for test set per pattern on FB15k-237 is demonstrated in table 6.11. It can be observed that rule injection is making some models perform worse than non-rule injected model. Only ComplEx is performing better with injected rules for test sets consisting of implication patterns. For predicting inverse test patterns, RotatE and ComplEx are performing well with the rule injected model. In the case of symmetric pattern, RotatE is performing better with a trained rule injected model.

Table 6.12 demonstrates the evaluation performance on test set per pattern on WN18. It is observed that for implication, symmetric and inverse test, and reflexive test patterns, LogicENN is performing well upon injection of rules. The performance of RotatE and ComplEx slightly increases in terms of filtered hit@10. Other models such as RotatE and ComplEx have a slight increase in performance in Hit@10 across all test patterns.

The evaluation tables for Kinship (table 7.1 in Appendix) and UMLS (table 7.2 in Appendix) for test set per pattern have been moved to Appendix. Table 7.1 from the Appendix shows the evaluation result of Kinship. It is observed that, for symmetric test pattern of Kinship dataset, all the rule injected models perform well upon rule injection across all the performance metrics. In the inverse pattern, rule injected models performed comparably worse than no rule injected models in the kinship dataset except Distmult.

The reflexive pattern is not detected in the UMLS dataset. According to table 7.2 from Appendix which is demonstrating the evaluation result of UMLS, Rule injected RotatE model is performing better than non rule injected RotatE model across all the test patterns (implication, inverse, and symmetric). Though some models do have a slightly better performance margin, in most cases, performance decreases with rule injection in this dataset.

Test Pattern	Model	MRR	Hit@1	Hit@3	Hit@5	Hit@10
Implication	**RotatE**	0.8771	0.8156	0.9322	0.9502	0.9661
	RotatE (Injection)	0.9125	0.8698	0.9527	0.9669	0.9741
	ComplEx	0.9033	0.8518	0.9510	0.9638	0.9738
	ComplEx (Injection)	0.9084	0.8572	0.9563	0.9669	0.9761
	Distmult	0.8283	0.7411	0.9022	0.9296	0.9599
	Distmult (Injection)	0.8473	0.7694	0.9140	0.9474	0.9643
	TransComplEx	0.8989	0.8382	0.9556	0.9751	0.9854
	TransComplEx (Injection)	0.8948	0.8310	0.9556	0.9733	0.9833
	TransE	0.6506	0.5455	0.7098	0.7740	0.8580
	TransE (Injection)	0.6753	0.5722	0.7401	0.7974	0.8631
	LogicENN	0.6295	0.4617	0.7524	0.8618	0.9489
	LogicENN (Injection)	0.6527	0.4944	0.7655	0.8716	0.9520
Inverse	**RotatE**	0.7499	0.6495	0.8286	0.8786	0.9210
	RotatE (Injection)	0.7969	0.7114	0.8642	0.9037	0.9340
	ComplEx	0.8476	0.7905	0.8977	0.9161	0.9298
	ComplEx (Injection)	0.8497	0.7926	0.8962	0.9168	0.9326
	Distmult	0.7464	0.6291	0.8510	0.8985	0.9284
	Distmult (Injection)	0.7448	0.6278	0.8510	0.8964	0.9260
	TransComplEx	0.7912	0.6987	0.8685	0.9084	0.9391
	TransComplEx (Injection)	0.7914	0.7035	0.8637	0.9036	0.9376
	TransE	0.4238	0.3002	0.4763	0.5607	0.6782
	TransE (Injection)	0.4173	0.2935	0.4693	0.5535	0.6680
	LogicENN	0.5011	0.3266	0.6083	0.7341	0.8584
	LogicENN (Injection)	0.5107	0.3426	0.6075	0.7334	0.8530
Symmetric	**RotatE**	0.4299	0.0001	0.8373	0.8935	0.9419
	RotatE (Injection)	0.4500	0.0000	0.8955	0.9354	0.9623
	ComplEx	0.4697	0.0596	0.8656	0.9216	0.9570
	ComplEx (Injection)	0.5007	0.0960	0.8958	0.9379	0.9636
	Distmult	0.4505	0.0420	0.8366	0.9114	0.9586
	Distmult (Injection)	0.4475	0.0385	0.8398	0.9100	0.9570
	TransComplEx	0.4211	0.0154	0.7999	0.8694	0.9371
	TransComplEx (Injection)	0.4168	0.0061	0.8035	0.8780	0.9386
	TransE	0.3275	0.0000	0.5961	0.6919	0.8018
	TransE (Injection)	0.3275	0.0000	0.5961	0.6919	0.8018
	LogicENN	0.5335	0.3493	0.6541	0.7651	0.8906
	LogicENN (Injection)	0.5384	0.3508	0.6637	0.7774	0.8952
Reflexive	**RotatE**	1.0000	1.0000	1.0000	1.0000	1.0000
	RotatE (Injection)	1.0000	1.0000	1.0000	1.0000	1.0000
	ComplEx	1.0000	1.0000	1.0000	1.0000	1.0000
	ComplEx (Injection)	1.0000	1.0000	1.0000	1.0000	1.0000
	Distmult	1.0000	1.0000	1.0000	1.0000	1.0000
	Distmult (Injection)	1.0000	1.0000	1.0000	1.0000	1.0000
	TransComplEx	0.9943	0.9886	1.0000	1.0000	1.0000
	TransComplEx (Injection)	0.9972	0.9943	1.0000	1.0000	1.0000
	TransE	1.0000	1.0000	1.0000	1.0000	1.0000
	TransE (Injection)	1.0000	1.0000	1.0000	1.0000	1.0000
	LogicENN	0.9260	0.8807	0.9631	0.9801	0.9943
	LogicENN (Injection)	0.9256	0.8864	0.9545	0.9858	0.9943

TABLE 6.10: Evaluation of rule injected model versus no rule injected model on FB15k dataset on test set consisting specific patterns

Test Pattern	Model	MRR	Hit@1	Hit@3	Hit@5	Hit@10
Implication	**RotatE**	0.1935	0.0310	0.2610	0.4212	0.5646
	RotatE (Injection)	0.1906	0.0271	0.2636	0.4199	0.5646
	ComplEx	0.1333	0.0116	0.1550	0.3023	0.4393
	ComplEx (Injection)	0.1369	0.0103	0.1705	0.3217	0.4522
	Distmult	0.1638	0.0207	0.2028	0.3708	0.5142
	Distmult (Injection)	0.1587	0.0181	0.2093	0.3656	0.5142
	TransComplEx	0.1856	0.0310	0.2481	0.3837	0.5401
	TransComplEx (Injection)	0.1892	0.0336	0.2558	0.3902	0.5297
	TransE	0.1927	0.0362	0.2532	0.3824	0.5491
	TransE (Injection)	0.1764	0.0271	0.2481	0.3643	0.5090
	LogicENN	0.1642	0.0568	0.1848	0.2907	0.4070
	LogicENN (Injection)	0.1273	0.0181	0.1525	0.2623	0.3734
Inverse	**RotatE**	0.2233	0.0217	0.3105	0.5253	0.6986
	RotatE (Injection)	0.2205	0.0181	0.3069	0.5271	0.7148
	ComplEx	0.1795	0.0181	0.2112	0.4097	0.6011
	ComplEx (Injection)	0.1861	0.0162	0.2365	0.4440	0.6209
	Distmult	0.2217	0.0289	0.2816	0.5000	0.6968
	Distmult (Injection)	0.2020	0.0144	0.2708	0.4928	0.6787
	TransComplEx	0.2141	0.0199	0.3014	0.4801	0.6733
	TransComplEx (Injection)	0.2194	0.0235	0.3051	0.4874	0.6570
	TransE	0.2123	0.0199	0.2906	0.4567	0.6643
	TransE (Injection)	0.1978	0.0162	0.2888	0.4350	0.6282
	LogicENN	0.2052	0.0650	0.2365	0.3827	0.5361
	LogicENN (Injection)	0.1713	0.0271	0.2112	0.3556	0.5054
Symmetric	**RotatE**	0.1819	0.0000	0.2435	0.4383	0.6153
	RotatE (Injection)	0.1874	0.0000	0.2549	0.4513	0.6347
	ComplEx	0.1419	0.0000	0.1623	0.3409	0.5097
	ComplEx (Injection)	0.1511	0.0032	0.1883	0.3718	0.5195
	Distmult	0.1758	0.0032	0.2224	0.4091	0.5893
	Distmult (Injection)	0.1662	0.0000	0.2175	0.4140	0.5795
	TransComplEx	0.1786	0.0016	0.2451	0.3994	0.5990
	TransComplEx (Injection)	0.1877	0.0000	0.2679	0.4318	0.5974
	TransE	0.1707	0.0000	0.2273	0.3701	0.5601
	TransE (Injection)	0.1639	0.0000	0.2289	0.3588	0.5519
	LogicENN	0.1753	0.0487	0.1948	0.3247	0.4708
	LogicENN (Injection)	0.1465	0.0162	0.1672	0.3003	0.4481
Reflexive	**RotatE**	1.0000	1.0000	1.0000	1.0000	1.0000
	RotatE (Injection)	1.0000	1.0000	1.0000	1.0000	1.0000
	ComplEx	1.0000	1.0000	1.0000	1.0000	1.0000
	ComplEx (Injection)	1.0000	1.0000	1.0000	1.0000	1.0000
	Distmult	0.9737	0.9730	0.9730	0.9730	0.9730
	Distmult (Injection)	1.0000	1.0000	1.0000	1.0000	1.0000
	TransComplEx	0.9943	0.9886	1.0000	1.0000	1.0000
	TransComplEx (Injection)	0.9972	0.9943	1.0000	1.0000	1.0000
	TransE	1.0000	1.0000	1.0000	1.0000	1.0000
	TransE (Injection)	1.0000	1.0000	1.0000	1.0000	1.0000
	LogicENN	0.7732	0.7027	0.8108	0.8446	0.9054
	LogicENN (Injection)	0.9077	0.8514	0.9527	0.9730	0.9730

TABLE 6.11: Evaluation of rule injected model versus no rule injected model on FB15k-237 dataset on test set consisting specific patterns

Test Pattern	Model	MRR	Hit@1	Hit@3	Hit@5	Hit@10
Implication	**RotatE**	0.9568	0.9531	0.9593	0.9610	0.9635
	RotatE (Injection)	0.9568	0.9527	0.9597	0.9615	0.9652
	ComplEx	0.9502	0.9493	0.9508	0.9511	0.9513
	ComplEx (Injection)	0.9506	0.9493	0.9510	0.9521	0.9533
	Distmult	0.8074	0.6879	0.9232	0.9382	0.9500
	Distmult (Injection)	0.8171	0.7062	0.9240	0.9390	0.9508
	TransComplEx	0.7439	0.6144	0.8587	0.9149	0.9508
	TransComplEx (Injection)	0.6091	0.4313	0.7619	0.8703	0.9342
	TransE	0.4291	0.0003	0.8528	0.9225	0.9508
	TransE (Injection)	0.4113	0.0000	0.8153	0.9101	0.9480
	LogicENN	0.8722	0.8338	0.9045	0.9168	0.9288
	LogicENN (Injection)	0.9156	0.8859	0.9447	0.9501	0.9523
Inverse	**RotatE**	0.9546	0.9505	0.9573	0.9582	0.9609
	RotatE (Injection)	0.9545	0.9509	0.9562	0.9580	0.9624
	ComplEx	0.9492	0.9484	0.9494	0.9498	0.9505
	ComplEx (Injection)	0.9495	0.9481	0.9499	0.9513	0.9524
	Distmult	0.7393	0.5645	0.9093	0.9298	0.9471
	Distmult (Injection)	0.7514	0.5883	0.9091	0.9300	0.9466
	TransComplEx	0.7895	0.6859	0.8814	0.9275	0.9535
	TransComplEx (Injection)	0.7296	0.6054	0.8389	0.8986	0.9405
	TransE	0.4474	0.0034	0.8912	0.9341	0.9516
	TransE (Injection)	0.4380	0.0047	0.8717	0.9281	0.9507
	LogicENN	0.8732	0.8348	0.9072	0.9187	0.9300
	LogicENN (Injection)	0.9394	0.9315	0.9466	0.9484	0.9496
Symmetric	**RotatE**	0.9374	0.9316	0.9410	0.9453	0.9491
	RotatE (Injection)	0.9380	0.9307	0.9444	0.9474	0.9517
	ComplEx	0.9330	0.9307	0.9346	0.9350	0.9358
	ComplEx (Injection)	0.9325	0.9299	0.9341	0.9350	0.9376
	Distmult	0.9328	0.9277	0.9363	0.9371	0.9393
	Distmult (Injection)	0.9353	0.9303	0.9384	0.9410	0.9440
	TransComplEx	0.6589	0.4919	0.8058	0.8781	0.9290
	TransComplEx (Injection)	0.3566	0.0804	0.5843	0.7913	0.8995
	TransE	0.3862	0.0000	0.7536	0.8832	0.9354
	TransE (Injection)	0.3552	0.0000	0.6856	0.8589	0.9286
	LogicENN	0.8482	0.8041	0.8841	0.8978	0.9136
	LogicENN (Injection)	0.8432	0.7630	0.9226	0.9371	0.9414
Reflexive	**RotatE**	0.9526	0.9483	0.9553	0.9586	0.9618
	RotatE (Injection)	0.9537	0.9479	0.9590	0.9609	0.9646
	ComplEx	0.9478	0.9460	0.9493	0.9497	0.9497
	ComplEx (Injection)	0.9479	0.9460	0.9493	0.9497	0.9511
	Distmult	0.9484	0.9465	0.9488	0.9497	0.9516
	Distmult (Injection)	0.9505	0.9479	0.9516	0.9544	0.9567
	TransComplEx	0.6731	0.5079	0.8175	0.8873	0.9409
	TransComplEx (Injection)	0.3621	0.0000	0.6997	0.8696	0.9385
	TransE	0.3947	0.0000	0.7733	0.8957	0.9455
	TransE (Injection)	0.3621	0.0000	0.6997	0.8696	0.9385
	LogicENN	0.8640	0.8226	0.8971	0.9101	0.9250
	LogicENN (Injection)	0.8555	0.7751	0.9353	0.9497	0.9539

TABLE 6.12: Evaluation of rule injected model versus no rule injected model on WN18 dataset on test set consisting specific patterns

6.3 Trained Embedding Visualization

In this section, the visualization of trained embedding is demonstrated. This section aims to explain how similar entities are overlapped or separated in the lower-dimensional projection of embedding vectors. The process of visualizing those embedding vectors are briefly described in section 3.7 of chapter 3. T-SNE has been used to project the higher dimensional embedding vectors to a lower one. In the following images, the type of embedding to be visualized are: */organization/endowed_organization*, */film/film*, */tv/tv_program*, */music/instrument*, */location/statistical_region*, and */music/group_member*.

Hyperparameters are specified for TSNE. The perplexity and number of the iteration are set to 50 and 1000. For initializing T-SNE scikit-learn's [9] standard TSNE library[1] has been used. For visualization, Seaborn [63] library has been utilized and used. Figure 6.1, provides an illustration of untrained embedding, which means these corresponding entities are not trained, just vector values are randomly initialized for each corresponding types as the embedding space is randomly initialized for KGE models at the beginning of the training process. The separation of the embedding vectors of corresponding types can not be observed in this figure.

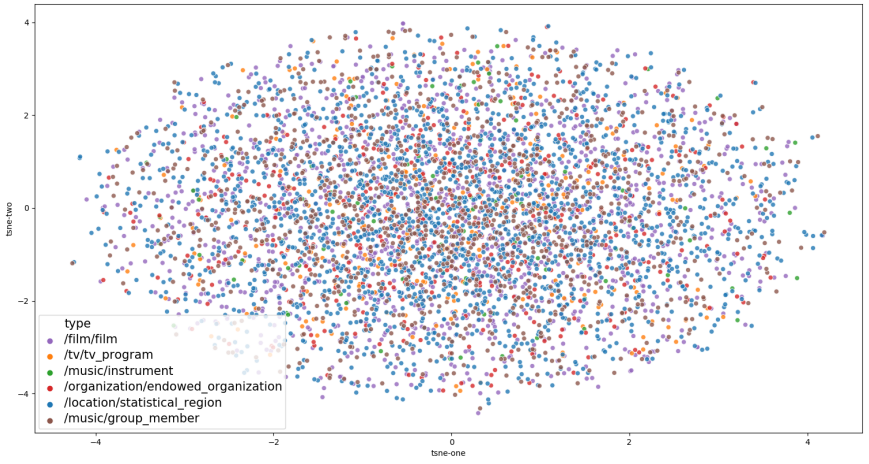


FIGURE 6.1: Embedding visualization of untrained embedding

Figure 6.2 and 6.3 illustrates the embedding vectors after 50 iteration of training for RotatE (both with rule injection and no rule injection). It appears that not much of a difference can be visualized for both of the models. If there was a

[1]https://scikit-learn.org/stable/modules/generated/sklearn.manifold.TSNE.html

very low accuracy before or after injecting rules, then both images could have been very different as types would not be separated very well, or similar types of entities would not be clustered for poorly trained embedding.

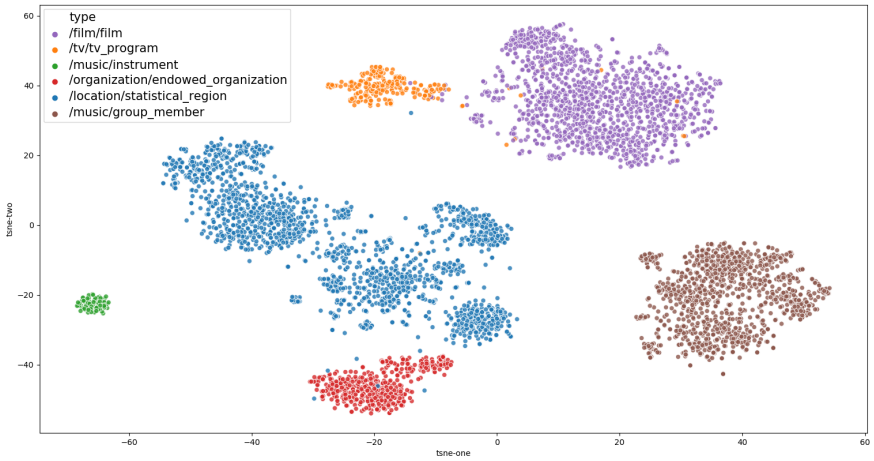


FIGURE 6.2: Rule injected trained embedding visualization of RotatE

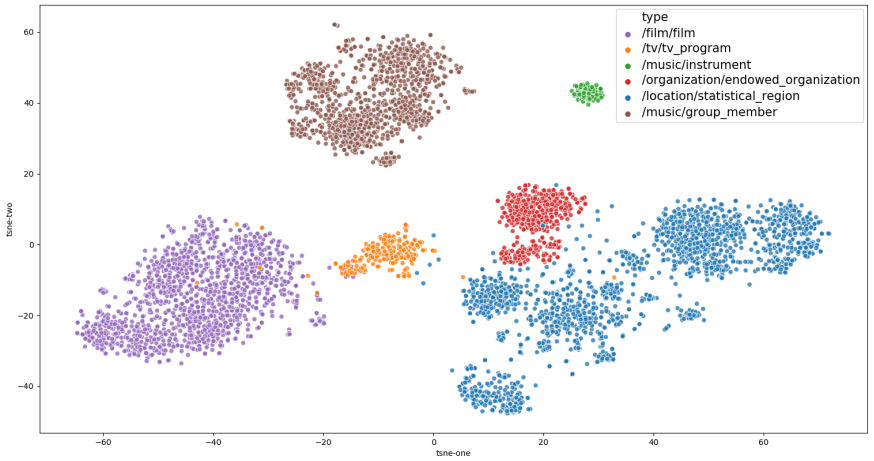


FIGURE 6.3: No Rule injected trained embedding visualization of RotatE

Now in a separate scenario is considered in another figure which is illustrated in figure 7.1 in Appendix, where a separate class *people/person* has been taken as one of the indicated types, in order to show this type can potentially overlap with */music/group_member* since both of the categories can be considered similar. This figure clarifies that similar types of trained embeddings can overlap since their embedding vectors are similar. It ensures the training of the entity embedding is meaningful.

Chapter 7

Conclusion and Future work

In this thesis, a rule injection system **RULECT** has been developed where three different modules exist for separate tasks: extraction of rules from general RDF knowledge graph datasets, preprocessing of the required data and training of knowledge graph embedding models with the rule injection process. Focusing on several problems were acting as key factors in this thesis. Firstly, to know about how an embedding model reacts when rule injection has been performed, especially in terms of performance. Secondly, leakages might exist in the test set in popular KGs, which may essentially make the link prediction task easier than expected. To find out relevant answers, leakages have been removed from the test sets of mentioned Knowledge Graphs. Same trained KGE models (which is used to identify of rule injection effect over KGE models on regular test sets) is utilized make an inference on leakage-free test set. Thirdly, it is known that embedding models are capable of learning implication, inverse, symmetric and reflexive patterns from training sets. One important part is identifying whether rule injection makes the training more effective so that the same trained models on test sets containing such patterns show performance increase in case of rule injection. For training and evaluation, six training models: RotatE, ComplEx, Distmult, TransComplEx, TransE, and LogicENN are integrated in the system in such a way that groundings can be injected while training of those models. After successful training, these models can be saved to perform inference on different test sets.

7.1 Conclusion and Discussion

Three key research questions are addressed at the beginning of this thesis. To draw out and briefly discuss the conclusions, those research questions are repeated here again.

Research Question 1: Can we leverage injection of logical rules in the KGEmodels towards increasing their performance?

To answer this research question, a brief investigation has been done on several datasets; namely, RotatE, TransComplEx, ComplEx, Distmult, TransE, and LogicENN on both rule injected and no rule injected model. It is already mentioned that, rules are injected in the form of groundings. For each embedding model, to perform fair analysis, the same hyperparameter settings have been considered for the rule injected, and no rule injected model. It has been observed that, in most of the cases, rule injected model performs better than no rule injected embedding models, especially RotatE performs very well on almost every dataset if rules are injected while training. This effect can be clearly seen in FB15k dataset after rule injection in RotatE model from table 6.1 in section 6.2.1 of chapter 6. This subsection (subsection 6.2.1 from chapter 6) also depicts that other models are also performing well if rules are injected in most of the cases with the only exception of FB15k-237. From table 4.1 in chapter 4, it can be seen that, for FB15k-237, no leakages exists in the test set. At this point, another research question arises, which is also repeated as follows:

Research Question 2: What can be the effect of rule injection in KGE models for leakage-free test set?

To get a clear answer to this one, the same KGE models trained for the first research question have been used. Only the test sets were replaced by refined test versions of the original. Preparation of such test sets is briefly discussed in section 4.4 of chapter 4. It has been observed that the same KGE models which were trained previously are performing worse than it performed on the original test set and rule injection merely helped the models to have better performance on leakage-free test sets. Though it can be seen that, some KGE models slightly performed better across the performance metrics if rules are injected in this case (this can be seen in subsection 6.2.2 of section 6.2 at chapter 6). In many cases,

rule injection has significant effects if the test sets have leakages (FB15k and WN18 have significant amount of leakages); that might be a potential cause of the significant or slight improvement in performance which has been observed after injection. The third research question is to identify the effect of rule injection if only mentioned patterns are kept in the test set.

Research Question 3: Can rule injection be helpful when only specific patterns appear in the test set?

To analyze this research question, test sets considering only implication, inverse, symmetric and reflexive patterns have been taken. Other triples are filtered out from the test set. The same trained embedding models have been used which are used for the analysis of previous research questions. It turns out that rule injection helped in order to identify those patterns. The evaluation results of FB15k and WN18 from subsection 6.2.3 clearly demonstrate the effects of rule injection on such patterns.

7.2 Future Work

The future direction of this research has the potential to take several pathways. For the rule extraction part, AMIE+ [15] is used for mining rules from KGs. Later from these rules, groundings are generated. This research's future direction can lead to the development of another standalone rule mining system, where logical rules are dynamically generated while training in a batch wise manner instead of fetching them before training. From those rules, groundings can be also generated and injected in runtime. In this way, the rule injection system can be more robust and efficient per batch of training data. Though it is a question of computational capabilities for KGE model's training, it is to be seen in the future. In this research, only four logical rules are considered for injection: implication, inverse, symmetric, and equivalence. There is a plan to work with other logical rules such as transitive, composition, asymmetric as well. In terms of the development of **RULECT** system, the plan is to make it more automated, robust and user interaction oriented. Also, enriching the system with more recent embedding models is a potential plan. In conclusion, this research carved a pathway for leveraging rule injection and their effects in KGE models, which may potentially lead the area of link prediction to the next level.

Bibliography

[1] Mojtaba Nayyeri, Chengjin Xu, Yadollah Yaghoobzadeh, Hamed Shariat Yazdi, and Jens Lehmann. Toward understanding the effect of loss function on then performance of knowledge graph embedding. *arXiv preprint arXiv:1909.00519*, 2019.

[2] Antoine Bordes, Nicolas Usunier, Alberto Garcia-Duran, Jason Weston, and Oksana Yakhnenko. Translating embeddings for modeling multi-relational data. In C. J. C. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 26, pages 2787–2795. Curran Associates, Inc., 2013. URL `https://proceedings.neurips.cc/paper/2013/file/1cecc7a77928ca8133fa24680a88d2f9-Paper.pdf`.

[3] Zhen Wang, Jianwen Zhang, Jianlin Feng, and Zheng Chen. Knowledge graph embedding by translating on hyperplanes. In *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence*, pages 1112–1119, 2014.

[4] 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.

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

[6] Mirza Mohtashim Alam, Mojtaba Nayyeri, Chengjin Xu, Md Rashad Al Hasan Rony, Hamed Shariat Yazdi, Afshin Sadeghi, and Jens Lehmann. The effect of rule injection in leakage free dataset. In *Workshop of Knowledge Representation and Representation Learning co-organized at ECAI Conference*, 2020.

[7] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. dAlché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 32, pages 8026–8037. Curran Associates, Inc., 2019. URL https://proceedings.neurips.cc/paper/2019/file/bdbca288fee7f92f2bfa9f7012727740-Paper.pdf.

[8] Charles R. Harris, K. Jarrod Millman, St'efan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fern'andez del R'ıo, Mark Wiebe, Pearu Peterson, Pierre G'erard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. Array programming with NumPy. *Nature*, 585 (7825):357–362, September 2020. doi: 10.1038/s41586-020-2649-2. URL https://doi.org/10.1038/s41586-020-2649-2.

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

[10] The pandas development team. pandas-dev/pandas: Pandas, February 2020. URL https://doi.org/10.5281/zenodo.3509134.

[11] Wes McKinney. Data Structures for Statistical Computing in Python. In Stéfan van der Walt and Jarrod Millman, editors, *Proceedings of the 9th Python in Science Conference*, pages 56 – 61, 2010. doi: 10.25080/Majora-92bf1922-00a.

[12] Pauli Virtanen, Ralf Gommers, Travis E Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, et al. Scipy 1.0: fundamental algorithms for scientific computing in python. *Nature methods*, 17(3):261–272, 2020.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] Xi Victoria Lin, Richard Socher, and Caiming Xiong. Multi-hop knowledge graph reasoning with reward shaping. *arXiv preprint arXiv:1808.10568*, 2018.

[18] Stanley Kok and Pedro Domingos. Statistical predicate invention. In *Proceedings of the 24th international conference on Machine learning*, pages 433–440, 2007.

[19] Tim Dettmers, Minervini Pasquale, Stenetorp Pontus, and Sebastian Riedel. Convolutional 2d knowledge graph embeddings. In *Proceedings of the 32th AAAI Conference on Artificial Intelligence*, pages 1811–1818, February 2018. URL `https://arxiv.org/abs/1707.01476`.

[20] Mojtaba Nayyeri, Chengjin Xu, Mirza Mohtashim Alam, Jens Lehmann, Yazdi, and Hamed Shariat. Logicenn: A neural based knowledge graphsembedding model with logical rules. Under review at Transactions on Pattern Analysis and Machine Intelligence, 2020.

[21] Mojtaba Nayyeri, Mirza Mohtashim Alam, Jens Lehmann, and Sahar Vahdati. 3d learning and reasoning in link prediction over knowledge graphs. *IEEE Access*, 8:196459–196471, 2020.

[22] 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.

[23] Xiaotian Zhou. An evaluation of translation-based knowledge graph embeddings with a focus on loss function. Master's thesis, Rheinische Friedrich-Wilhelms-Universität Bonn, September 2020.

[24] Nadezhda Vassilyeva. Relational patterns: Explicit vs implicit learning. Master's thesis, Rheinische Friedrich-Wilhelms-Universität Bonn, September 2020.

[25] Richard H Richens. Preprogramming for mechanical translation. *Mech. Transl. Comput. Linguistics*, 3(1):20–25, 1956.

[26] Amit Sheth, Swati Padhee, and Amelie Gyrard. Knowledge graphs and knowledge networks: The story in brief. *IEEE Internet Computing*, 23(4):67–75, 2019.

[27] Ezio Marchi and Osvaldo Miguel. On the structure of the teaching-learning interactive process. *International Journal of Game Theory*, 3(2):83–99, 1974.

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

[29] 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.

[30] 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.

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

[32] 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, 2007.

[33] Amit Singhal. Introducing the knowledge graph: things, not strings. *Official google blog*, 5, 2012.

[34] 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.

[35] Tom M Mitchell et al. Machine learning. 1997. *Burr Ridge, IL: McGraw Hill*, 45(37):870–877, 1997.

[36] Claude Lemaréchal. Cauchy and the gradient method. *Doc Math Extra*, 251: 254, 2012.

[37] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12(61):2121–2159, 2011. URL `http://jmlr.org/papers/v12/duchi11a.html`.

[38] Tijmen Tieleman and Geoffrey Hinton. Lecture 6.5-rmsprop, coursera: Neural networks for machine learning. *University of Toronto, Technical Report*, 2012.

[39] Alex Graves. Generating sequences with recurrent neural networks. *arXiv preprint arXiv:1308.0850*, 2013.

[40] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[41] Sebastian Ruder. An overview of gradient descent optimization algorithms. *arXiv preprint arXiv:1609.04747*, 2016.

[42] 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, 2013.

[43] Luc Dehaspe and Hannu Toivonen. Discovery of frequent datalog patterns. *Data Mining and knowledge discovery*, 3(1):7–36, 1999.

[44] Luc Dehaspe and Hannu Toivonen. Discovery of relational association rules. In *Relational data mining*, pages 189–212. Springer, 2001.

[45] Ross D King, Ashwin Srinivasan, and Luc Dehaspe. Warmr: a data mining tool for chemical data. *Journal of Computer-Aided Molecular Design*, 15(2): 173–181, 2001.

[46] Bart Goethals and Jan Van den Bussche. Relational association rules: Getting warmer. In David J. Hand, Niall M. Adams, and Richard J. Bolton, editors, *Pattern Detection and Discovery*, pages 125–139, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg. ISBN 978-3-540-45728-2.

[47] Mehdi Ali, Max Berrendorf, Charles Tapley Hoyt, Laurent Vermue, Sahand Sharifzadeh, Volker Tresp, and Jens Lehmann. Pykeen 1.0: A python library for training and evaluating knowledge graph emebddings. *arXiv preprint arXiv:2007.14175*, 2020.

[48] Xu Han, Shulin Cao, Lv Xin, Yankai Lin, Zhiyuan Liu, Maosong Sun, and Juanzi Li. Openke: An open toolkit for knowledge embedding. In *Proceedings of EMNLP*, 2018.

[49] Eric Miller. An introduction to the resource description framework. *Bulletin of the American Society for Information Science and Technology*, 25(1):15–19, 1998.

[50] . URL `https://pytorch.org/docs/stable/autograd.html`.

[51] Matthew D Zeiler. Adadelta: an adaptive learning rate method. *arXiv preprint arXiv:1212.5701*, 2012.

[52] . URL `https://pytorch.org/docs/stable/optim.html`.

[53] . URL `https://pytorch.org/docs/stable/generated/torch.nn.Module.html`.

[54] Changsung Moon, Paul Jones, and Nagiza F Samatova. Learning entity type embeddings for knowledge graph completion. In *Proceedings of the 2017 ACM on conference on information and knowledge management*, pages 2215–2218, 2017.

[55] Changsung Moon, Steve Harenberg, John Slankas, and Nagiza Samatova. Learning contextual embeddings for knowledge graph completion. In *Pacific Asia Conference on Information Systems (PACIS)*, volume 10, 2017.

[56] Laurens van der Maaten and Geoffrey Hinton. Visualizing data using t-sne. *Journal of machine learning research*, 9(Nov):2579–2605, 2008.

[57] Kristina Toutanova, Danqi Chen, Patrick Pantel, Hoifung Poon, Pallavi Choudhury, and Michael Gamon. Representing text for joint embedding of text and knowledge bases. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*, pages 1499–1509, Lisbon, Portugal, September 2015. Association for Computational Linguistics. doi: 10.18653/v1/D15-1174. URL https://www.aclweb.org/anthology/D15-1174.

[58] Chengjin Xu, Mojtaba Nayyeri, Fouad Alkhoury, Jens Lehmann, and Hamed Shariat Yazdi. Temporal knowledge graph embedding model based on additive time series decomposition. *arXiv preprint arXiv:1911.07893*, 2019.

[59] Chengjin Xu, Mojtaba Nayyeri, Fouad Alkhoury, Hamed Shariat Yazdi, and Jens Lehmann. Tero: A time-aware knowledge graph embedding via temporal rotation. *arXiv preprint arXiv:2010.01029*, 2020.

[60] Sarthak Dash and Alfio Gliozzo. Distributional negative sampling for knowledge base completion. *arXiv preprint arXiv:1908.06178*, 2019.

[61] Mirza Mohtashim Alam, Hajira Jabeen, Mehdi Ali, Karishma Mohiuddin, and Jens Lehmann. Affinity dependent negative sampling for knowledge graph embeddings. In *Proceedings of the 3rd Workshop on Deep Learning for Knowledge Graphs co-located with ESWC 2020*, 2020.

[62] Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization. *arXiv preprint arXiv:1711.05101*, 2017.

[63] Michael Waskom and the seaborn development team. mwaskom/seaborn, September 2020. URL https://doi.org/10.5281/zenodo.592845.

List of Figures

List of Tables

Appendix

Test Pattern	Model	MRR	Hit@1	Hit@3	Hit@5	Hit@10
Inverse	**RotatE**	0.8085	0.7016	0.8918	0.9443	0.9689
	RotatE (Injection)	0.8180	0.7295	0.8934	0.9344	0.9672
	ComplEx	0.7656	0.6475	0.8787	0.9262	0.9607
	ComplEx (Injection)	0.7666	0.6459	0.8672	0.9197	0.9590
	Distmult	0.2347	0.0967	0.2033	0.3508	0.6574
	Distmult (Injection)	0.2404	0.0902	0.2295	0.3689	0.6836
	TransComplEx	0.5630	0.3820	0.6770	0.8164	0.9246
	TransComplEx (Injection)	0.5367	0.3426	0.6656	0.8066	0.9361
	TransE	0.3149	0.0246	0.5246	0.6984	0.8377
	TransE (Injection)	0.3133	0.0131	0.5311	0.7148	0.8590
	LogicENN	0.8382	0.7525	0.9115	0.9426	0.9705
	LogicENN (Injection)	0.8611	0.7918	0.9213	0.9459	0.9689
Symmetric	**RotatE**	0.8481	0.7409	0.9562	0.9818	0.9927
	RotatE (Injection)	0.8513	0.7518	0.9526	0.9818	1.0000
	ComplEx	0.7488	0.6095	0.8686	0.9453	0.9745
	ComplEx (Injection)	0.7844	0.6569	0.9161	0.9453	0.9854
	Distmult	0.8204	0.7080	0.9234	0.9635	0.9927
	Distmult (Injection)	0.8364	0.7263	0.9489	0.9781	0.9964
	TransComplEx	0.5002	0.2956	0.6022	0.7883	0.9489
	TransComplEx (Injection)	0.5558	0.3540	0.6825	0.8321	0.9599
	TransE	0.1358	0.0000	0.1058	0.2117	0.4964
	TransE (Injection)	0.1562	0.0000	0.1752	0.2737	.5073
	LogicENN	0.8881	0.8029	0.9781	0.9854	0.9964
	LogicENN (Injection)	0.8659	0.7737	0.9453	0.9854	1.0000

TABLE 7.1: Evaluation of rule injected model versus no rule injected model on Kinship dataset on test set consisting specific patterns

Test Pattern	Model	MRR	Hit@1	Hit@3	Hit@5	Hit@10
Implication	**RotatE**	0.9236	0.8678	0.9736	0.9880	0.9952
	RotatE (Injection)	0.9196	0.8630	0.9688	0.9976	1.0000
	ComplEx	0.5393	0.3389	0.6803	0.7957	0.9423
	ComplEx (Injection)	0.5915	0.4087	0.7260	0.8510	0.9423
	Distmult	0.3914	0.2476	0.4952	0.5385	0.6466
	Distmult (Injection)	0.3918	0.2572	0.4736	0.5240	0.6418
	TransComplEx	0.9239	0.8750	0.9688	0.9880	1.0000
	TransComplEx (Injection)	0.9236	0.8750	0.9736	0.9880	1.0000
	TransE	0.4644	0.0000	0.9231	0.9832	0.9976
	TransE (Injection)	0.4627	0.0000	0.9062	0.9736	0.9976
	LogicENN	0.9232	0.8726	0.9760	0.9856	0.9928
	LogicENN (Injection)	0.9312	0.8894	0.9712	0.9808	0.9952
Inverse	**RotatE**	0.9236	0.8678	0.9736	0.9880	0.9952
	RotatE (Injection)	0.9196	0.8630	0.9688	0.9976	1.0000
	ComplEx	0.5393	0.3389	0.6803	0.7957	0.9423
	ComplEx (Injection)	0.5915	0.4087	0.7260	0.8510	0.9423
	Distmult	0.3914	0.2476	0.4952	0.5385	0.6466
	Distmult (Injection)	0.3918	0.2572	0.4736	0.5240	0.6418
	TransComplEx	0.9239	0.8750	0.9688	0.9880	1.0000
	TransComplEx (Injection)	0.9236	0.8750	0.9736	0.9880	1.0000
	TransE	0.4644	0.0000	0.9231	0.9832	0.9976
	TransE (Injection)	0.4627	0.0000	0.9062	0.9736	0.9976
	LogicENN	0.9232	0.8726	0.9760	0.9856	0.9928
	LogicENN (Injection)	0.9312	0.8894	0.9712	0.9808	0.9952
Symmetric	**RotatE**	0.9236	0.8678	0.9736	0.9880	0.9952
	RotatE (Injection)	0.9196	0.8630	0.9688	0.9976	1.0000
	ComplEx	0.5393	0.3389	0.6803	0.7957	0.9423
	ComplEx (Injection)	0.5915	0.4087	0.7260	0.8510	0.9423
	Distmult	0.3914	0.2476	0.4952	0.5385	0.6466
	Distmult (Injection)	0.3918	0.2572	0.4736	0.5240	0.6418
	TransComplEx	0.9239	0.8750	0.9688	0.9880	1.0000
	TransComplEx (Injection)	0.9236	0.8750	0.9736	0.9880	1.0000
	TransE	0.4644	0.0000	0.9231	0.9832	0.9976
	TransE (Injection)	0.4627	0.0000	0.9062	0.9736	0.9976
	LogicENN	0.9232	0.8726	0.9760	0.9856	0.9928
	LogicENN (Injection)	0.9312	0.8894	0.9712	0.9808	0.9952

TABLE 7.2: Evaluation of rule injected model versus no rule injected model on UMLS dataset on test set consisting specific patterns

FIGURE 7.1: Embedding visualization of trained embedding(RotatE) for overlapping classes