RHEINISCHE FRIEDRICH-WILHELMS-UNIVERSITÄT
BONN
INSTITUT FÜR INFORMATIK III

Indexed Negative Sampling And Scalable Knowledge Graph Embedding Models

Master Thesis

Tasneem Tazeen Rashid
Matriculation number: 3032710
March 5, 2021

Supervisor: Dr. Hajira Jabeen

Reviewer 1: Prof. Dr. Jens Lehmann
Reviewer 2: Dr. Giulio Napolitano

UNIVERSITÄT BONN

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 Any published (or unpublished) ideas or techniques from the work of others are fully acknowledged in accordance with the standard referencing practices.

Tasneem Tazeen Rashid

Signed:

Date: 05.03.2021

Acknowledgements

I would like to start conveying my gratitude to ***Prof. Dr. Jens Lehmann****. Ever since I participated in his course on Knowledge Graph Analysis at the University of Bonn, I felt inspired to explore more to the research area of Knowledge Graph Embeddings Models and their vast applications. I also am grateful to him for being the first examiner for my thesis. At the same time, I would like to express my gratitude to* ***Dr. Giulio Napolitano*** *for agreeing to be my second examiner.*

At the initial stage, I was inspired by ***Dr. Hamed Shariat Yazdi*** *to start working on my thesis. And I am taking this opportunity to thank him as well.*

My special wholehearted gratitude to my thesis supervisor, ***Dr. Hajira Jabeen*** *for her intensive guidance and invaluable support to help complete the thesis gracefully. I may add that it was a challenging path but she always guided me in the right direction with encouragement.*

I would also like to thank ***Mr. Mirza Mohtasim Alam****, who provided his insights into my thesis topic that benefited my work.*

My parents ***Mr. Harunur Rashid*** *and* ***Mrs. Humaera Tazeen*** *have been the pillar and the inspiring source of all my achievements. Their constant support, love and care always assisted me to accomplish my aim to attend higher studies in Germany and wished for a brighter future in my career.*

Lastly, I would like to thank the ***University of Bonn*** *for providing me the opportunity to be a part of this esteemed institute that aided me to realize my dream to study for this prestigious M.Sc. degree. This institute inspired me to increase my knowledge, skills and competency for a greater future.*

Contents

Chapter 1

Introduction

To perform a complex task efficiently, a program must have insights into the knowledge of the world in which it operates [1]. In recent years, it has emerged that data has become more valuable than oil [2], and Knowledge Graph (KG) has developed to be a methodical way to represent all these data of the world. Knowledge Graphs are a representation of data as entities connected by their relations. Entities behave like nodes and relations as the edges between the nodes of the graph. Knowledge Graph strengthens the magnitude of other multitudes of fields, such as Natural Language Processing, Artificial Intelligence, Machine Learning, etc. However, Knowledge Graph suffers from the "incompleteness problem" [3], and the Knowledge Graph Embedding model has appeared to be a popular solution to this. This model is a latent feature space representation of the Knowledge Graph acting as a learning model [4]. With limited resources of the machinery and exponential growth of the data, achieving a well-trained model with optimized time and utilizing more inexpensive but available machinery should be opted as an important sector to explore.

1.1 Problem Statement

Over the last few years, constructing a more efficient scoring function and loss function has been the most focused sector for improving different Knowledge Graph Embedding models [5]. Even some researchers use different optimization algorithms for a well-trained model [6] [7] [8]. However, avoiding the generation of nonsensical negatives samples is important for these models and is often overlooked [9]. For example, Knowledge Graph Embedding models can suffer from the vanishing gradient problem if the generated negative samples are not proper [10]. Hence, it is safe to convey that creating meaningful negative samples might help a better-trained model.

There are various attempts of creating meaningful negative samples for the Knowledge Graph Embedding models [5] [9] [11]. The Knowledge Graph follows the concept of the closed world assumption [11]. Anything that is not present in Knowledge Graph is considered unknown. Thus, the Knowledge Graph Embedding model's generic structure imposes checking if the generated negative sample is unknown to the Knowledge Graph. Even if it is a meaningful negative sample. In the real world, the sizes of these Knowledge Graphs are large in number. In consequence, this checking can contribute to the training time of the model. With limited amounts of machinery resources, reducing the Knowledge Graph Embedding model's processing time might be beneficial.

The principal target of the various Knowledge Graph Embedding models is contributing to the Knowledge Graph Completion process [3]. Hence, the performance of these models is the supreme factor. All the researchers embarked upon this area are focused on proposing efficient model training. Therefore, exploring the proposition of optimizing the training time should be carefully crafted to avoid compromising the model's performance.

With the elevation of the Knowledge Graphs in terms of the graph's number of facts, their model's execution time also explodes. This demands an increase in the computation power of the computing machinery [12]. Distributed Computing comes as a scalable solution in such a scenario, which facilitates the applications into multiple machines [13]. Application of the Knowledge Graph Embedding models in a distributed computing manner maintaining the scalability surely will have a huge advantage.

Optimization of the Knowledge Graph Embedding models runs on the embeddings per epoch. However, distributing this optimization is a big challenge since it comes with its own benefits and constraints [14]. In a sequential approach, the training phase takes one batch, computes the loss, and optimizes the entity and relation embeddings with respect to that loss [15]. Thus for distributed models, the crucial query is how the parameters are updated per batch such that no information gets lost.

1.2 Research Questions

To understand the core idea behind the problem statement, four research questions are crafted carefully. Based on these questions, three variants of Negative Sampling Algorithms and a distributed approach of the Knowledge Graph Embedding models are proposed in this thesis. These research questions and how they are unraveled in this thesis are mentioned below:

Question 1. Can meaningful negative sampling improve the training of the Knowledge Graph Embedding models?

Three original Negative Sampling Algorithms: RNS [15], DNS [9], and ADNS [11] are considered to explore this inquiry. Three variants of Negative Sampling Algorithms are proposed in this thesis by injecting a new concept into these algorithms. This question is addressed in two sections. First, it is checked if the proposed algorithm's variants create more meaningful negative samples than RNS. Second, it is checked how they are performing with respect to DNS and ADNS since both of these algorithms aim for creating meaningful negative samples. The proposed Negative Sampling Algorithms are implemented with PyTorch. The evaluation metrics related to the link prediction problem are checked for the trained KGE model's wellness.

Question 2. Can training time of the Knowledge Graph Embedding models be improved with the available Negative Sampling Algorithms?

Training time with DNS and ADNS is exponential compared to RNS, the first proposed Negative Sampling Algorithm. In fact, one of the inspirations of ADNS has been an attempt to reduce the processing time of DNS. The three variants of the Negative Sampling Algorithms proposed in this thesis check how the processing time can be optimized with respect to both DNS and ADNS.

Question 3. Does the improvement of the training time compromise the performance of the Knowledge Graph Embedding models?

Principally, the proposed Negative Sampling Algorithm's motivation has been optimizing the training time compared to the current Negative Sampling Algorithms. It is observed as well if the performance is not compromised with the help of evaluation metrics.

Question 4. How does the distributed approach of Knowledge Graph Embedding models perform?

To inspect this question, three Knowledge Graph Embedding models: TransE [15], DistMult [16], and ComplEx [17] are implemented with a popular distributed framework called Spark and BigDL library. BigDL is a library built on the top of Spark, which offers various optimizers. The implemented model's performances are assessed with respect to evaluation metrics related to the link prediction problem.

Keeping all these research questions in mind, the goal of this thesis can be defined as:

"Create meaningful negative sampling with optimized training time and explore distributed Knowledge Graph Embedding model's scalability."

1.3 Methodology

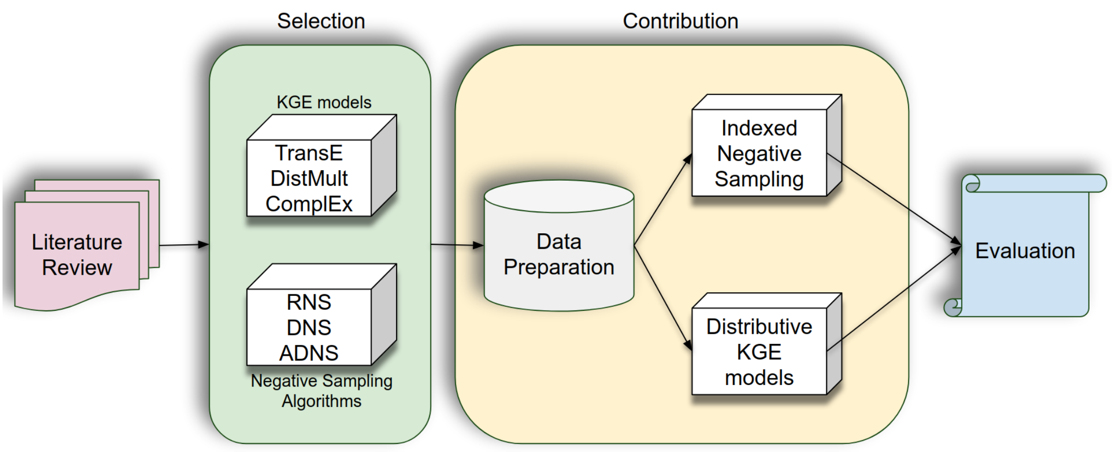


FIGURE 1.1: Overall methodology of the thesis

Fundamentally, an in-depth exploration of the literature review has been done to acknowledge the most important aspects of the KGE models and their functionalities. This study acted as the motivation to address the problem statement mentioned in Section 1.1 which sets the direction of this thesis. With the comprehensive research to acknowledge the research questions elaborated in Section 1.2, three variants of Negative Sampling Algorithms and distributed approaches of three KGE models have been proposed. The overall methodology of this thesis is shown in Figure 1.1.

RNS[15], DNS[9], ADNS[11] Negative Sampling Algorithms are considered to evaluate how injecting a new concept named *Indexed Dataset* benefits the KGE models with respect to the performance and optimization of the training time. The proposed Negative Sampling Algorithms are identified as *Indexed RNS*, *Indexed DNS* and *Indexed ADNS* respectively. In general they can be recognized as *Indexed Negative Sampling Algorithms*. TransE [15], DistMult [16] and ComplEx [17] KGE models are selected to test these proposed algorithms. Additionally, the selected KGE model's distributed approach is explored in this thesis, and their performances are evaluated.

The overall contribution of this thesis can be structured into three major parts elaborated in the following chapters. The contributions are:

1. Data preparation with both Python and Spark frameworks.
2. Implementation of the three proposed Negative Sampling algorithms with PyTorch.
3. Exploration of distributed Knowledge Graph Embedding models with Spark and BigDL.

Chapter 2 details the conceptual and technical background required for this thesis. **Chapter 3** describes related work regarding different Knowledge Graph Embedding models and different Negative Sampling Algorithms. **Chapter 4** focuses on how the data are prepared to fit the structure of training the Knowledge Graph Embedding models. Everything in this chapter is implemented in both Python and Spark. **Chapter 5** proposes three variants of Negative Sampling Algorithms and the motivation behind them. These algorithms are implemented with PyTorch. **Chapter 6** explores the distributed implementations of three Knowledge Graph Embedding models with Spark framework and BigDL library. **Chapter 7** is the chapter where all the results and findings of the contributions of this thesis are elaborated. **Chapter 8** outlines the conclusion and future work of this thesis.

Chapter 2

Background

This chapter is categorized into two parts, the conceptual background and the technical background required for this thesis. Section 2.1 emphasizes the background of the Knowledge Graph, the basic understanding of the Knowledge Graph Embedding models, their fundamental architecture, and ranges of applications. Section 2.2 briefly illustrates the implementation tools, programming languages, and the necessary libraries.

2.1 Conceptual Background

2.1.1 Knowledge Graph

A Knowledge Graph (KG) is structured information of entities connected through a set of relations. In the history of Artificial Intelligence, this form of data representation has been in practice for years. Scientists are trying to develop competent concepts to delineate the data so that machines can solve human-level problems efficiently for quite some time now. The earliest attempt at the modern KG was introduced by R. H. Richnes in 1956 [18]. He established a preprogramming for mechanical translation by representing sentences as graphs by linking nodes with respect to the relation shown in Figure 2.1.

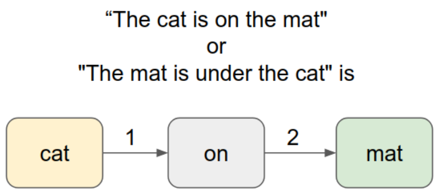


FIGURE 2.1: R. H. Richnes' example of mechanical translation of relation "on" and entities "cat" and "mat" from a sentence

Modern Knowledge Graph (KG) consists of a head (h), a tail (t), and the relation (r) between the head and tail. This is depicted as a triple format of the KG. The triple can be considered as the subject, predicate, and object of the natural language. The head (h) or subject and the tail (t) or object are considered as entities, and the relation (r) or predicate can be considered as the relation. The entire KG is connected through these triples as shown in Figure 2.2.

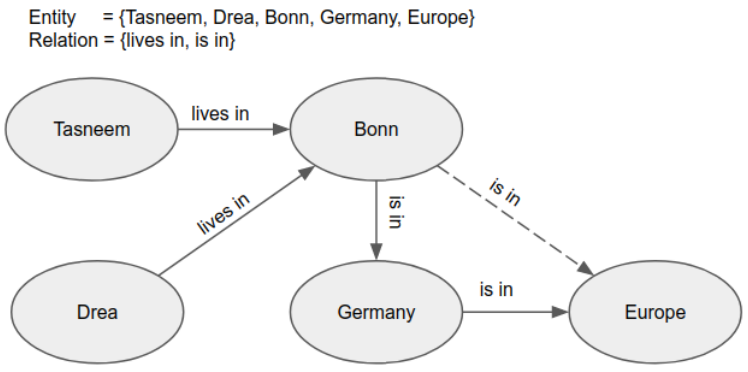

Entity = {Tasneem, Drea, Bonn, Germany, Europe}
Relation = {lives in, is in}
Tasneem
lives in
Bonn
lives in
is in
is in
Drea
Germany
is in
Europe

FIGURE 2.2: A sample Knowledge Graph with a list of entities and relations

There are various available KGs, which are: DBpedia [19], Freebase [20], WordNet [21], YAGO [22], Kinship [23], Nations [24], UMLS [25] etc. In 2012, Google announced an addition of Knowledge Graph to their search engine to facilitate a broader, deeper and easier search for the user [26].

2.1.2 Knowledge Graph Embedding (KGE) Models

Knowledge Graph Embeddings (KGE) are low dimension vector representations of all the entities and the relations of a Knowledge Graph. Various models are introduced in the past couple of years that use these KGE to infer the Knowledge Graph's relations. This thesis is focused on the latent feature models. These models intend to apprehend the semantics, their homogeneity, and corresponding pattern. The input of these models is in the RDF triple format of the Knowledge Graph, and they are trained to predict future unknown facts. For example, in Figure 2.2 a missing fact (*Bonn*, *is in*, *Europe*) is indicated by the dotted line, which can be predicted with these trained KGE models. The entities and relations are mapped to latent feature vector spaces, which can be named embedding vectors, and the weights of these vectors are learned while training. Some of the most popular KGE models are RESCAL [4], TransE [15], TransH [10], TransR [3], DistMult [16], ComplEx [17], RotatE [27], etc. The following section will discuss the basic anatomy of these KGE models.

2.1.3 General Architecture Of KGE Models

KGE models thoroughly follow a neural architecture for learning the embedding vectors of the graph. The basic architecture of the KGE models are shown in Algorithm 1 and explained in the following sections. Initially, the KGE model creates a vector embedding space with the provided dimension, and then the embeddings are randomly initialized. Until the loss of the model is not optimal or iteration has not reached a maximum number, the following steps are repeated: generating a sample batch of positive triples, creating negative samples of the positive batch, Score calculation of the positive and negative batch, loss calculation, optimization of the embedding with respect to loss at a given learning rate.

Algorithm 1 General Architecture of the KGE Models

Input Triple (h, r, t) of Knowledge Graph, dimension K of the embeddings, learning rate l of the gradient step
Returns Trained embeddings of entities E and relation R

```
1: Map the list of entities $e$ to $K$ dimension vector space $E$
2: Map the list of relations $r$ to $K$ dimension vector space $R$
3: Initialize $E$ and $R$ randomly
4: while loss is not optimal or minimum number of epochs do
5:     Generate $P \leftarrow$ batch of positive triples
6:     Generate $N \leftarrow$ batch of negative triples
7:     $P_{\text{score}} \leftarrow \text{Score}(P)$
8:     $N_{\text{score}} \leftarrow \text{Score}(N)$
9:     Calculate $\text{Loss}(P_{\text{score}}, N_{\text{score}})$
10:    Update embeddings $E$ and $R$ w.r.t Loss and $l$
11: end while
12: return $E$ and $R$
```

2.1.3.1 Embedding Initialization

Proper initialization of the embedding vector is important for fast convergence, and feasible learning process [28]. KGE models like TransE, DistMult, ComplEx etc. initialize the embedding by Equation 2.1 proposed a decade ago [29].

$$W_{\text{ij}} \leftarrow U[-\frac{1}{\sqrt{n}}, \frac{1}{\sqrt{n}}] \tag{2.1}$$

Here, W_{ij} is the weight of the embeddings and U indicates uniform distribution within a range and n is the dimension of the embeddings.

2.1.3.2 Negative Sampling

Knowledge Graphs are built on true facts. Thus, a negative sample for KGE models are those triples that are not present in the Knowledge Graph, similar to the closed world assumption problem. For the Knowledge Graph shown in Figure 2.2 (*Bonn*, *lives in*, *Bonn*) will be a negative sample. For better training of the KGE models, meaningful negative sampling is important [9][11][30]. The basic structure of generating these negative samples is either consider head (h) or tail (t) to be replaced by another entity which will not create a triple true for the

Knowledge Graph as shown in Equation 2.2.

$$
\begin{gathered}
S^{-} = \{(h', r, t) | h' \in \varepsilon, h' \neq h \; where \; (h, r, t) \in S^{+}\} \\
OR \\
\{(h, r, t') | t' \in \varepsilon, t' \neq t \; where \; (h, r, t) \in S^{+}\}
\end{gathered}
\tag{2.2}
$$

Parameters of the Equation 2.2 depicts, S^{+} the set of positive samples, S^{-} set of negative samples, h' as the replaced entity in head position, and t' is the replaced entity in the tail position of the triple (h, r, t) of the Knowledge Graph.

2.1.3.3 Score Function

The score function of the KGE models refers to the scalar value calculation indicating the wellness of the trained KGE model. During training, KGE models calculate the score of the positive sample and the negative sample. This calculation aims to generalize the models on the training data to predict unknown facts of the Knowledge Graph with higher accuracy. The goal is to have higher scores for positive samples and lower scores for negative samples. Also, this score is the basis to compute the overall loss of the model. Different KGE algorithms use different scoring functions [15][16][17].

2.1.3.4 Loss Function

The scores of positive and negative samples are passed through a loss function which is the pathway to optimize the embedding vectors of KGE models. Loss function basically calculates the difference between the true facts and false facts of the Knowledge Graph; hence the difference can also be labeled as *Error*. To train the model better, this error or loss needs to be minimized. The loss of the model tends to be higher at the beginning of the training phase. However, with each iteration, it reduces, which can indicate adequate learning of the model. There are many available loss functions [15][27][31][32][33] to help the model's performance.

2.1.3.5 Optimization

The optimization, more precisely the loss optimization phase, is the most important segment of the training process of the KGE models. This optimization's ultimate goal is to minimize the loss while training the model to estimate the model parameters. Gradient descent is one of the basic optimizations of any neural network [34]. There are other optimization algorithms available such as Adam [7], AdaGrad [8], SGD [6] etc. In each iteration, the KGE model's parameters are updated by taking a gradient step with a constant learning rate [15]. Learning rate is significant for the optimization, which essentially determines how big steps the gradient descent will take to achieve the local minimum of the loss of the KGE models.

2.1.4 Applications Of KGE Models

There are different applications of the KGE models that yield to the benefit of the process, like Knowledge Graph Completion [3]. These applications can be classified into two sectors, the "$in-KG$ *applications*" and the "$out-of-KG$ *applications*" [35]. An "$in - KG$" applications are exercised inside of the Knowledge Graph, mostly for evaluation protocols of the KGE models [36]. This includes link prediction, triple classification, entity classification, and entity resolution. The "$out - of-KG$" applications target the wider fields of Knowledge Graphs such as relation extraction, question answering, recommender systems, etc.

2.1.4.1 Link Prediction

The core concept of link prediction is predicting an entity that has a specific relation with another entity. For a triple (h, r, t) where h denotes head, r denotes relation, and t denotes tail of a fact, link prediction is defined as either $(?, r, t)$ or $(h, r, ?)$. For instance, (?, *lives in*, *Bonn*) means to find out who lives in *Bonn*. This task is done by a simple procedure of ranking the scores of a possible set of entities for the missing position of the triple. Link prediction can also be known as entity prediction [37].

2.1.4.2 Triple Classification

Triple classification is a binary classification task [38] which estimates whether a triple is true or false. A threshold value of the triple score with respect to a relation is set to margin the positive triples from the negative triples. For example, a threshold value δ is set for relation r, and the topmost scores for any triple (h, r, t) are considered true fact [35].

2.1.4.3 Entity Classification

Entity classification aims to predict entities that are referred to the same semantic categories. For example, for triple (*Bonn*, *is a*, *city*), relation "*is a*" can encode the entities to a certain type from the learned Knowledge Graph. Hence, it is restricted to predict (_, *is a*, ?) where '_' denoted any given entity. Entity classification can also be called a version of link prediction [35].

2.1.4.4 Entity Resolution

Entity resolution can be used to prune entities with the same semantics since in Knowledge Graphs one entity can be represented in several ways, for example, name, place, etc. This entity resolution can be obtained in two ways. First, by the help of the learned model of a Knowledge Graph, which indicates the similar semantics with a relation, such as (a, *equals to*, b) [39]. In such a case, measuring the triple score's correctness will achieve entity resolution similar to triple classification. However, such indicating relations may not be provided by the Knowledge Graphs. Thus, likelihood of two entities (a, b) is calculated with a formula $e^{-||a-b||^{2/c}}$ [4] , where c is a constant for entity resolution.

2.1.4.5 Relation extraction

Relation Extraction aims to predict semantic relations between pairs of entities from a text, given that the entities are already learned, which can be denoted as (h, ?, t) [40]. For example, "*Tasneem lives in Bonn*" where "*Tasneem*" and "*Bonn*" are detected as entities, the relation extraction task will detect "*lives in*" as a relation. This is a very pivotal aspect of NLP. One way of performing this task is to combine a KGE model with a text-based extractor from a text corpus [41] and calculate a new score as shown in Equation 2.3.

$$S_{\text{text+KG}}(h,r,t) = \sum_{m \in M_{\text{h,t}}} S_{\text{text}}(m,r) + S_{\text{KG}}(h,r,t) \tag{2.3}$$

$S_{\text{text}}(m,r)$ is the text-based extractor score between relation r and its textual mention m and $S_{\text{KG}}(h,r,t)$ is the score from the KGE model.

2.1.4.6 Question Answering

Question answering system with respect to the Knowledge Graph basically depicts how to acquire the correct answer as a form of triple or set of triples from a Knowledge Graph [42][43]. For example: "*Who lives in Bonn*?" should result in (*Tasneem*, *lives in*, *Bonn*) from the corresponding Knowledge Graph shown in Figure 2.2. This method aims to embed both the Knowledge Graph and the question into a lower vector space to keep the question's embedding vector and its corresponding answer closer.

2.1.4.7 Recommender Systems

Recommender systems suggest "*relevant*" items to users which can interest them for further wishful acquisitions or prospective applications. One of the possible approaches of this recommender system aims to combine collaborative filtering of the different semantic reparations of the item from the Knowledge Graph. The facts of the triples, the textual knowledge, and the domain's visual knowledge are integrated into the approach for improvement [44].

2.2 Technical Background

2.2.1 PyTorch

PyTorch[1] is a fast-maturing machine learning framework that provides easy application of mathematical functions and computes their gradients, and supports the Graphical Processing Unit (GPU) [45]. PyTorch offers simpler Tensor operations[2], high-level optimization[3] tasks by implementing neural network structures[4].

2.2.2 Pandas

Pandas[5] is a library written for Python as the foundation for Data Analysis and Statistics [46]. Pandas helps to read data as DataFrames[6], offers a magnitude of numerical libraries such as NumPy[7], SciPy[8] etc.

2.2.3 Python

Python[9] is a functional, imperative, and object-oriented programming language which provides fundamental numerical libraries [47]. Python brings the benefits of PyTorch and Pandas under one platform.

[1]https://pytorch.org/
[2]https://pytorch.org/docs/stable/torch.html
[3]https://pytorch.org/docs/stable/optim.html
[4]https://pytorch.org/docs/stable/generated/torch.nn.Module.htmltorch.nn.Module
[5]https://pandas.pydata.org/
[6]https://pandas.pydata.org/docs/reference/frame.html
[7]https://numpy.org/
[8]https://www.scipy.org/
[9]https://www.python.org/

2.2.4 Spark

Apache Spark[10] is a distributed computing system for big data enabling jobs to be completed faster and better than previous big data tools [48]. Spark provides the opportunity to run applications in clusters[11] and read RDDs[12] in a good manner.

2.2.5 BigDL

BigDL[13] is a distributed deep learning framework built on the top of Apache Spark. Providing the distributed datasets to the neural network helps perform distributed training on Spark [49]. BigDL covers vast applications of Tensors[14] (modeled after Torch[15]) and optimizers[16] running on the Spark clusters.

2.2.6 Scala

Scala[17] is one of the languages Spark provides for its developed APIs. Scala is a high-level language offering a combination of object-oriented and functional programming under one platform [50]. Advantages of Spark and BigDL are achieved through Scala applications.

[10]https://spark.apache.org/
[11]https://spark.apache.org/docs/latest/rdd-programming-guide.htmlinitializing-spark
[12]https://spark.apache.org/docs/latest/rdd-programming-guide.html
[13]https://bigdl-project.github.io/0.10.0/
[14]https://bigdl-project.github.io/0.10.0/APIGuide/Data/tensor
[15]http://torch.ch/
[16]https://bigdl-project.github.io/0.10.0/APIGuide/Optimizers/Optim-Methods/adam
[17]https://docs.scala-lang.org/tour/tour-of-scala.html

Chapter 3

Related Work

This chapter briefly discusses the scholarly studies related to the core areas of this thesis. Section 3.1 will emphasize on different KGE models and Section 3.2 will focus on different Negative Sampling Algorithms.

3.1 Knowledge Graph Embedding (KGE) Models

3.1.1 RESCAL

RESCAL [4] is one of the primary approaches of KGE models. It is a factorization of multi-relational data that basically computes a three-way factorization of adjacent tensors of the Knowledge Graph. Entities are represented as d - dimensional vector x (x_{h} for head and x_{t} for tail) and each relation is represented as $d \times d$ dimensional vector W_{r} and the corresponding score function of the model is given by Equation 3.1.

$$f_{\mathrm{RESCAL}} = {x_{\mathrm{h}}}^{\mathrm{T}} W_{\mathrm{r}} x_{\mathrm{t}} \tag{3.1}$$

3.1.2 SME

SME [39] regards the relation embedding equal to entity embedding. This method aims to apprehend the correlation between them by multiple linear matrix products.

3.1.3 TransH

TransH [10] proposes to translate the relations (r) on a hyperplane with a normal vector. The entity embeddings head (h) and tail (t) are translated on the hyperplane of W_{r} for r denoted as $h\_$ and $t\_$. Then Equation 3.2 is used as the score function of the model.

$$f_{\mathrm{TransH}} = ||h\_ \ + \ r \ - \ t\_||_2^2 \tag{3.2}$$

3.1.4 TransR

TransR [3] proposes two distinct spaces. One for entities, and the other is relation-specified entity space to capture the diversity of the relations denoted as $\mathrm{M_r}$. Score, as shown in Equation 3.4, of the model is calculated by the translated entities (h, t) with respect to corresponding relation (r) as shown in Equation 3.3.

$$h_{\mathrm{r}} = hM_{\mathrm{r}} \ , \qquad t_{\mathrm{r}} \ = \ tM_{\mathrm{r}} \tag{3.3}$$

$$f_{\mathrm{TransR}} = ||hM_{\mathrm{r}} \ + \ r \ - \ tM_{\mathrm{r}}||_2^2 \tag{3.4}$$

3.1.5 RotatE

RotatE [27] proposes a model where each relation is presented as a rotation from the source entity to target entity in a complex vector. Score function of this model is defined as shown in Equation 3.5.

$$f_{\mathrm{RotatE}} = ||h^{\circ}r \ - \ t|| \tag{3.5}$$

Here $h^{\circ}r$ depicts the element wise rotation (r) of relation from head (h) to tail (t).

3.2 Negative Sampling Algorithms Of KGE Models

3.2.1 Random Negative Sampling (RNS)

Random Negative Sampling algorithm or RNS is the simplest technique for generating negative samples for positive samples in the training datasets. Head (h) or tail (t) of a triple (h, r, t) is selected randomly to be replaced by another entity from the entire entity set. This chosen entity is also selected randomly. TransE [15] used RNS for triple corruption. Negative samples produced by RNS may not be useful. There is no guarantee that the negative sample is not true for the training dataset and meaningful, affecting the KGE model's performance.

3.2.2 Corrupting Positive Instances (C)

Corrupting Positive Instances (C) [51] creates a pool of heads (h) and tail (t) with respect to the relation (r). Then it selects randomly from the pool to corrupt the triple (h, r, t) for closer proximity of the chosen entity with the replaced entity. Adding to that, the entity is chosen so that the generated negative sample is not true for the training set. A problem with this algorithm is the scarcity of the available pool.

3.2.3 Typed Negative Sampling (TNS)

Typed Negative Sampling (T) [30] uses datasets that have strong typed relations such as NELL [52], FreeBase [20]. Which means a relation is always connected to a particular type of entity. While generating a negative sample, the entity is chosen from the same typed entity list for a more coherent training process.

3.2.4 Distributional Negative Sampling (DNS)

Distributional Negative Sampling or DNS [9] attempt to create meaningful negative sampling by calculating cosine similarity [53] between the vectors of the entity embeddings and the replacement candidate. The core concept of DNS is similar to TNS without providing a typed dataset. Principally, the cosine similarity learns the type of entity, and negative samples are created with the highest cosine similarity among the entire entity list. To create multiple negative samples for each positive triple, the entire process of calculating the cosine similarity is executed again and again, which creates an overhead of the model's training time. The formula for cosine similarities between vectors is given in Equation 3.6.

$$Similarity(X, Y) = \frac{X \cdot Y}{|X| \times |Y|} = \frac{\sum_{i=1}^{n} W_{\mathrm{Xi}} \times W_{\mathrm{Yi}}}{\sqrt{\sum_{i=1}^{n} {W_{\mathrm{Xi}}}^2} \times \sqrt{\sum_{i=1}^{n} {W_{\mathrm{Yi}}}^2}} \tag{3.6}$$

Here, $X = [W_{\mathrm{X0}}, W_{\mathrm{X1}} .... W_{\mathrm{Xn}}]$ and $Y = [W_{\mathrm{Y0}}, W_{\mathrm{Y1}} .... W_{\mathrm{Yn}}]$ are two vectors. W_{Xi} and W_{Yi} can be considered as the embedding weights.

3.2.5 Affinity Dependent Negative Sampling (ADNS)

Affinity Dependent Negative Sampling or ADNS [11] solves the calculation looping issue of DNS by introducing an additional affinity function or fitness function that acts as a probability vector for each entity in the list of entities. This enables the algorithm to choose multiple entities to create multiple negative samples for one positive triple at once. This function is defined as shown in Equation 3.7 where M_{i} is the cosine similarity score for one entity at i^{th} position and $\sum_j M_{\mathrm{j}}$ is the summation of cosine similarity scores of all the entities.

$$Fitness_{\mathrm{i}} = \frac{M_{\mathrm{i}}}{\sum_j M_{\mathrm{j}}} \tag{3.7}$$

Chapter 4

Data Preparation

This chapter focuses on preparing the dataset in a manner to use them inside the KGE models. Section 4.1 emphasises on how dictionaries from entities and relations are created. Section 4.2 shows how the original datasets are translated according to the created dictionaries, Section 4.3 elaborates how datasets are divided into the train, test, and validation datasets, and Section 4.4 describes the process of creating a probability table that is used to generate meaningful negative samples during the training phase of the KGE models. Finally, Section 4.5 explains the motivation behind creating a new version of the training dataset called *Indexed dataset* and the algorithm behind generating it. This dataset is used inside the proposed *Indexed Negative Sampling Algorithms* in this thesis.

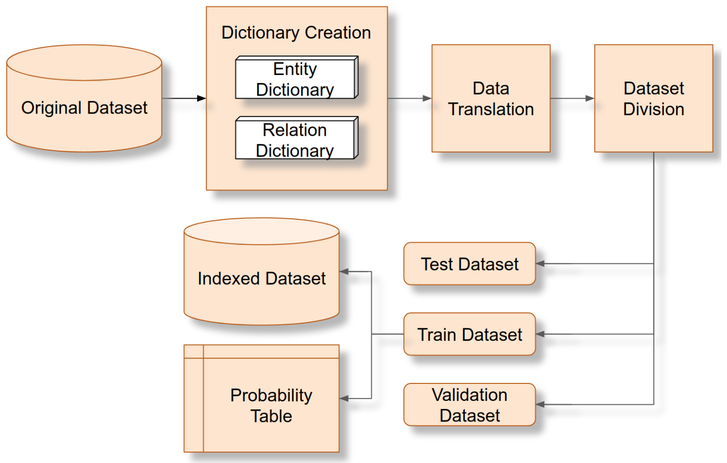


FIGURE 4.1: Architecture of the dataset preparation

In the real world, available datasets are Strings, represented in triple format. To fit these datasets into the KGE models, some pre-processings of the original datasets must be executed. This step can be called *Data Preparation* as shown in Figure 4.1. Successful completion of this phase will have the following files:

- Entity dictionary.
- Relation dictionary.
- Translated dataset from the original dataset using the dictionaries.
- Train, Test and validation dataset.
- The probability table for the corruption of the triples.
- The indexed dataset from the original dataset.

All these files are produced by both Python and Spark frameworks and will be discussed in the following parts of this chapter.

Head (h)	**Relation (r)**	**Tail (t)**
Tasneem	lives is	Bonn
Bonn	is in	Germany
Germany	is in	Europe
Uni bonn	is located at	Bonn
Drea	lives in	Bonn

TABLE 4.1: Sample training dataset

4.1 Creation Of The Dictionary

Algorithm 2 Dictionary Creation

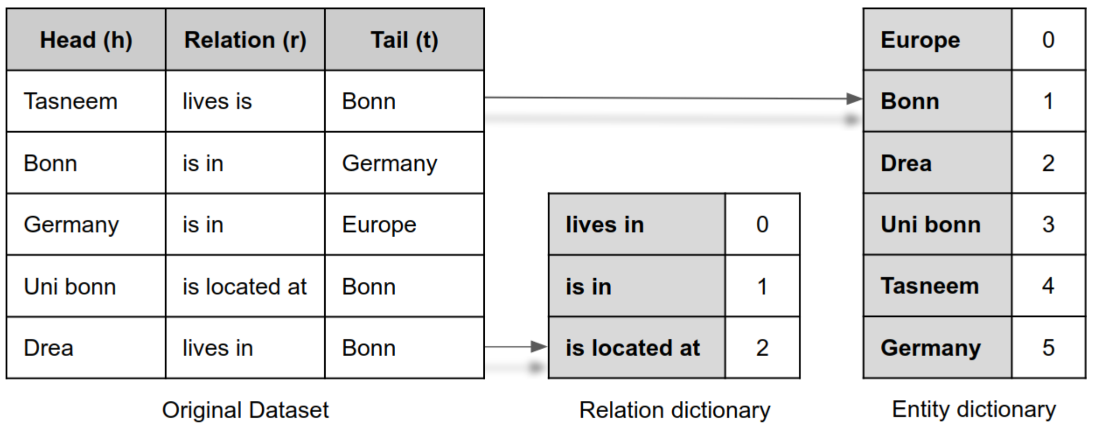
Input Original dataset $S = \{(h, r, t)\}$
Returns Entity dictionary ε, Relation dictionary $\mathcal{R}$

```
 1: $\varepsilon, \mathcal{R} = \phi$                    ▷ Initialization of the dictionaries
 2: $H \leftarrow$ Select unique heads $(h)$ from the dataset
 3: $T \leftarrow$ Select unique tails $(t)$ from the dataset
 4: $E \leftarrow (H \cap T)$
 5: $entityCount = 0$
 6: for $entity \in E$ do
 7:     Append $(entity, entityCount)$ to $\varepsilon$
 8:     Increment $(entityCount)$
 9: end for
10: $R \leftarrow$ Select unique relations $(r)$ from the dataset
11: $relationCount = 0$
12: for $relation \in R$ do
13:     Append $(relation, relationCount)$ to $\mathcal{R}$
14:     Increment $(relationCount)$
15: end for
16: return $(\varepsilon, \mathcal{R})$
```

Algorithm 2 shows the procedure to create the entity and relation dictionary from the original dataset, which is stored in String format. The steps are pretty simple. All the heads and tails are put under one list for entity dictionary with no duplicates. Once this process is done, a unique number is assigned to each of the entities. Similarly, for the relation dictionary, the list of unique relations is extracted from the entire dataset, and a unique number is set against each of the relations. The numbers for the dictionaries start at 0 from Python[1] implementation and at 1 from Spark[2] implementation and are incremented by 1. In this way,

[1]Index of Tensor in PyTorch starts at 0
[2]Index of Tensor in BigDL starts at 1

the dictionaries shown in Figure 4.2 generated from Table 4.1 will help indicate which index in the embedding vector corresponds to which entity or relation.



Head (h)	Relation (r)	Tail (t)
Tasneem	lives is	Bonn
Bonn	is in	Germany
Germany	is in	Europe
Uni bonn	is located at	Bonn
Drea	lives in	Bonn

Original Dataset

lives in	0
is in	1
is located at	2

Relation dictionary

Europe	0
Bonn	1
Drea	2
Uni bonn	3
Tasneem	4
Germany	5

Entity dictionary

FIGURE 4.2: Sample dictionaries

4.1.1 Python Implementation

Creating a dictionary with Python is relatively straightforward. The dataset is read as a DataFrame and sent as the input of the function shown in Figure 4.3. The unique combination of all the heads and tails is extracted in a column, as shown in lines 2 - 3. Another column is created by line 4 with a numerical value starting from 0. This column is considered as the corresponding ID of the entities. Similarly, all the unique relations are extracted in one column, and in the second column, an incremental number is set. The column headers of the dictionaries are set as *string* and *id*.

```python
1 def create_dictionary(data):
2     entity = pd.DataFrame(data=data['h']
3                           .append(data['t']).unique())
4     entity[1] = np.arange(len(entity))
5     entity = entity.rename(columns={0: 'string', 1: 'id'})
6     relation = pd.DataFrame(data=data['r'].unique())
7     relation[1] = np.arange(len(relation))
8     relation = relation.rename(columns={0: 'string', 1: 'id'})
9     return (entity, relation)
```

FIGURE 4.3: Python code for creating the dictionary from the original dataset

4.1.2 Spark Implementation

```
 1 val entityList = data.select("h").withColumnRenamed("h","t")
 2                      .union(data.select("t")).distinct.collect
 3                      .map(r => {r(0).asInstanceOf[String]})
 4 val relationList = data.select("r").distinct.collect
 5                        .map(r => {r(0).asInstanceOf[String]})
 6
 7 def CreateDictionary(entityList:Array[String],
 8                      relationList:Array[String],
 9                      spark:SparkSession): (DataFrame, DataFrame) = {
10     val schema = new StructType().add("string",StringType,true)
11                                  .add("id",IntegerType,true)
12     val sc = spark.sparkContext
13     val entityDictionary = spark.createDataFrame(
14                                  sc.parallelize(entityList)
15                                  .map(r => {
16                                  Row(r, entityList.indexOf(r)+1)}),
17                                  schema)
18     val relationDictionary = spark.createDataFrame(
19                                    sc.parallelize(relationList)
20                                    .map(r => {
21                                    Row(r, relationList.indexOf(r)+1)}),
22                                    schema)
23     (entityDictionary, relationDictionary)
24   }
```

FIGURE 4.4: Spark code for creating the dictionary from the original dataset

Figure 4.4 shows the implementation of Algorithm 2 with Spark using Scala. *CreateDictionary* function takes the lists of unique entities and unique relations, and a SparkSession[3]. The lists are created before calling the functions with basic Spark *Action* and *Transformation* methods[4][48] as shown in lines 1 - 5. The function returns the created entity and relation dictionaries as DataFrames. At the beginning of the *Data Preparation* phase, entities and relations are extracted as a list of arrays. Since this phase has the liberty to create the dictionaries, the list's index is set as the corresponding ID of entity or relation for the easiest solution. Generally, the index of a list or array starts at 0. Thus, for creating the dictionary IDs, the index of the list will be incremented by 1 so that IDs start at 1[5] as shown in lines 13 - 22. To save the dictionaries, two DataFrames are

[3]SparkSession is the entry point to programming Dataset and DataFrame API
[4]https://spark.apache.org/docs/latest/rdd-programming-guide.html
[5]For training in BigDL, the Tensor's index starts at 1

created by mapping the String and the ID as rows of a RDD using SparkContext[6]. The RDD[Row] is then converted to DataFrames with a defined *Schema*, which basically depicts the column's structure type. Column names (*string* and *id*) can also be mentioned inside the schema. If the column's fields can contain *null* value, true is set as the schema's parameter.

4.2 Translation Of The Original Dataset

Algorithm 3 Data Translation

Input Original dataset $S = \{(h, r, t)\}$, entity dictionary ε and relation dictionary $\mathcal{R}$

Returns Translated dataset S'

```
 1: $S' \leftarrow \phi$                                   ▷ Stores the converted dataset
 2: for $triple \in S$ do
 3:     $h \leftarrow$ head of the $triple$
 4:     $r \leftarrow$ relation of the $triple$
 5:     $t \leftarrow$ tail of the $triple$
 6:     $h' \leftarrow$ Extract ID of $h$ from $\varepsilon$
 7:     $r' \leftarrow$ Extract ID of $r$ from $\mathcal{R}$
 8:     $t' \leftarrow$ Extract ID of $t$ from $\varepsilon$
 9:     Append $(h', r', t',)$ to $S'$
10: end for
11: return $S'$
```

The basic structure of translating the original dataset from String to numerical format with respect to the dictionaries of entities and relation is shown in Algorithm 3. Using the dictionaries created from Algorithm 2 as shown in Figure 4.2, Table 4.1 is translated to Table 4.2.

Head (h)	**Relation (r)**	**Tail (t)**
4	0	1
1	1	5
5	1	0
3	2	1
2	0	1

TABLE 4.2: Translated dataset from the original sample dataset using the dictionaries

[6]SparkContext renders the connection to a Spark cluster which can create RDD

4.2.1 Python Implementation

For the implementation of the Algorithm 3 in Python shown in Figure 4.5, the original dataset, entity dictionary, and relation dictionary are taken as DataFrames as input. First, the entity and relation DataFrames are converted into dictionaries with the command *set_index*, which sets the index of the DataFrame using one or more existing columns. Then the column representing the IDs is taken as shown in lines 3-4. Following this, the triple dataset is mapped with respect to the modified dictionaries, as shown in lines 5-7.

```python
1 def data_translation(data, entity, relation):
2     new_data = data
3     entity_dict = entity.set_index('string')['id']
4     relation_dict = relation.set_index('string')['id']
5     new_data['h'] = new_data['h'].map(entity_dict)
6     new_data['r'] = new_data['r'].map(relation_dict)
7     new_data['t'] = new_data['t'].map(entity_dict)
8     return new_data
```

FIGURE 4.5: Python code for translating original dataset with respect to the dictionaries

4.2.2 Spark Implementation

DataFrames are immutable in Spark [48]. Hence, one can not change any value inside a current DataFrame rather map a new one. As previously mentioned in Section 4.1 (1 + index of the entity or relation list) is set as the ID. Thus, to translate the original dataset, function $indexOf()$ is used to get the corresponding ID (index) and mapped to a new set of rows. The function $DataTranslation$ shown in Figure 4.6 takes the original dataset, entity list, relation list, and a SparkSession as input and returns the translated dataset as a new DataFrame. The output DataFrame is created similarly to the process mentioned in Section 4.1 with a *schema*.

```
val entityList = data.select("h").withColumnRenamed("h","t")
                     .union(data.select("t")).distinct.collect
                     .map(r => {r(0).asInstanceOf[String]})
val relationList = data.select("r").distinct.collect
                       .map(r => {r(0).asInstanceOf[String]})

def DataTranslation (data:DataFrame, entityList:Array[String],
                   relationList:Array[String], spark:SparkSession):
                   (DataFrame) = {
    val schema = new StructType().add("h",IntegerType,true)
                                 .add("r",IntegerType,true)
                                 .add("t",IntegerType,true)
    val newData = spark.createDataFrame(data.rdd
                       .map(r => {
                       Row((entityList.indexOf(r(0))+1).toString,
                           (relationList.indexOf(r(1))+1).toString,
                           (entityList.indexOf(r(2))+1).toString)}),
                           schema)
    (newData)
  }
```

FIGURE 4.6: Spark code for translating original dataset with respect to the dictionaries

4.3 Division Of Dataset

The original dataset should be divided into three different datasets. They are the train dataset - used to train the KGE model, the valid dataset that validates the training, and the test dataset, which tests how well the model is trained. Generally, 80:10:10 is the ratio of splitting into three datasets.

4.3.1 Python Implementation

```
train = data.sample(frac=0.80, replace=False, random_state=0)
temp  = data.drop(train.index)
test  = temp.sample(frac=0.50, replace=False, random_state=0)
valid = temp.drop(test.index)
```

FIGURE 4.7: Python code to divide original dataset to train, test and validation datasets

Figure 4.7 shows how datasets can be split into multiple datasets. Dataset is read as a Pandas *DataFrame*. This splitting is performed in two steps. First the 80% from the dataset is taken into the *train* variable with the command *sample*(). Parameters of this command depict as following: $frac = 0.80$ means the fraction of the sample to be taken. The second parameter is *replace*, which is set to false so that rows are not repeated while creating the sample. Lastly, *random_state* is set to 0 because this acts as a seed. This seed ensures that the same results will be returned even after executing the command several times. To create the test and valid dataset, line 2 - 4 is executed in Figure 4.7. The rest of the 20% dataset is stored in temp variables that are not present in train data. This temp dataset is divided into 50-50 to achieve the ratio of 80:10:10 of the train, valid and test datasets.

4.3.2 Spark Implementation

```
1 val temp = data.randomSplit(Array(0.80, 0.10, 0.10), 11L)
2 val trainSet = temp(0)
3 val testSet  = temp(1)
4 val validSet = temp(2)
```

FIGURE 4.8: Spark code to divide original dataset to train, test and validation datasets

Figure 4.8 shows the Scala command to split the dataset in line 1. The method is called $randomSplit(weights, seed)$. This method takes parameters of an array with the weights that the dataset should be divided into and a long value that acts as the seed. The split datasets are stored as an array of datasets.

4.4 Creation Of The Probability Table

This probability table is inspired from TransH [10]. In this paper, the probabilities for replacing the head or tail for creating negative samples depend on the mapping type of the relation, for example, one-to-many, many-to-one, many-to-many. If the relation is mapped to one-to-many, then the higher probability is given to replace the head. If the relation is mapped to many-to-one, then the tail is given a higher probability to be replaced for creating negative samples. The target for this approach is to reduce the generation of false-negative samples. Therefore two calculations are made: the average number of tails per heads denoted as *tph* shown in Equation 4.1 and the average number of heads per tails denoted as *hpt* shown in Equation 4.2.

$$tail\ per\ head\ (tph) = \frac{number\ of\ all\ tails}{number\ of\ unique\ heads} \tag{4.1}$$

$$head\ per\ tail\ (hpt) = \frac{number\ of\ all\ heads}{number\ of\ unique\ tails} \tag{4.2}$$

A Bernoulli distribution [54] is used to get the corresponding probability of replacing the head or tail with the *tph* and *hpt*. Given a triple (h, r, t) the probability of corrupting a head, P_{h} is as shown in Equation 4.3 and the probability of corrupting tail, P_{t} is as shown in Equation 4.4 and a table is created with these two probabilities.

$$p_{\mathrm{h}} = \frac{tph}{tph\ + hpt} \tag{4.3}$$

$$p_{\mathrm{t}} = \frac{hpt}{tph\ + hpt} \tag{4.4}$$

4.4.1 Python Implementation

For this implementation, the dataset sent in the function shown in Figure 4.9 has to be a translated dataset as mentioned in Section 4.2. Also, the number of entities and relations are passed as parameters. Two empty n-dimensional arrays are created to calculate the head and tail entity's occurrences with respect to a particular relation, as shown in lines 2 - 6. Then, the probability table is created for the Equation 4.3 and Equation 4.4. Function $sum(hr[i,:])$ calculates the total number of head or tail entities that appeared per relation. This includes duplicate entities as well. Function $np.sum(hr[i,:] > 0)$ is used to calculate how many indices are non-zero in the array, as shown in lines 9 - 10, which basically indicates how many unique head or tail entities appeared per relation.

```python
def probability_table(data, n_entity, n_relation):
    hr = np.zeros([n_relation, n_entity])
    tr = np.zeros([n_relation, n_entity])
    for index, row in new_data.iterrows():
        hr[row['r'], row['h']] += 1
        tr[row['r'], row['t']] += 1
    prob_table = np.zeros([n_relation, 2])
    for i in range(0, n_relation):
        prob_table[i, 0] = sum(hr[i, :]) / np.sum(hr[i, :] > 0)
        prob_table[i, 1] = sum(tr[i, :]) / np.sum(tr[i, :] > 0)
    prob_table = pd.DataFrame(prob_table,
                              columns=['tph', 'hpt'])
    return prob_table
```

FIGURE 4.9: Python code to create the probability table

```scala
1  def ProbabilityTable(data:DataFrame, spark:SparkSession){
2      import spark.implicits._
3      var probabilityTable = "tph\thpt\n"
4      val relation = data.select("r").distinct.collect.map(r =>
5                                    {r(0).asInstanceOf[String]})
6      relation.foreach(r => {
7        val temp = data.where($"r" === r)
8        val totalOccurence = temp.count.toDouble
9        val head = temp.select("h").distinct.count.toDouble
10       val tail = temp.select("t").distinct.count.toDouble
11       probabilityTable = probabilityTable + totalOccurence/head + "\t"
12                                           + totalOccurence/tail + "\n"
13     })
14     (probabilityTable)
15 }
```

FIGURE 4.10: Spark code to create the probability table

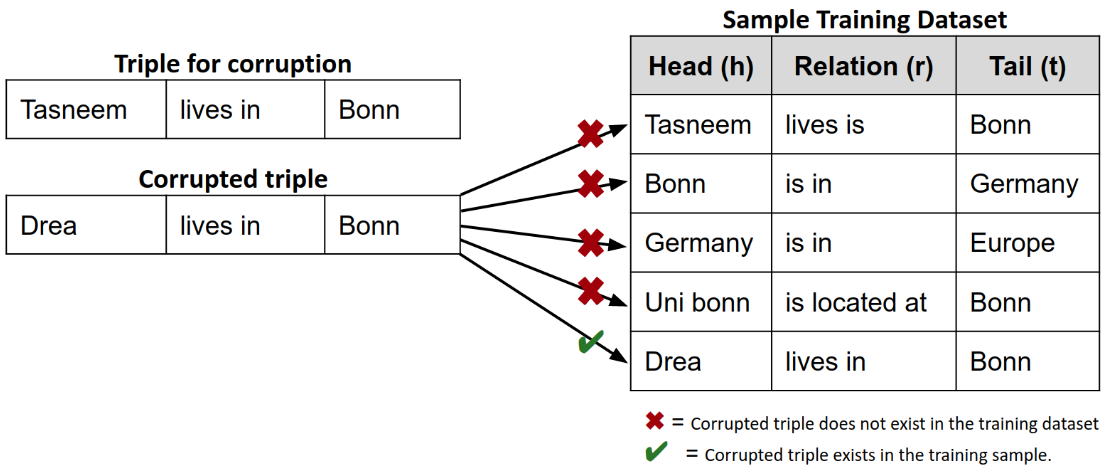
4.4.2 Spark Implementation

Figure 4.10 illustrates the Scala implementation of creating the probability table. Lists of relations are extracted from the input dataset. For each relation, the sub-dataset is extracted, having only one unique relation, as shown in line 7. After that, the number of total occurrences of the rows, unique numbers of head and tail in the sub-dataset are calculated as shown in lines 8 - 10. With these three values *tph* and *hpt* is calculated. Again, DataFrame in Scala is immutable. Thus for a simpler approach, the table is returned as a tab-separated string which will be saved in the local machine as a *tsv* (tab-separated) file.

4.5 Creation Of The Indexed Dataset

Creating the *Indexed Dataset* is important for the proposed variants of the Negative Sampling Algorithms in this thesis. Basically, the original datasets are indexed in a new table to improve the current variants of Negative Sampling Algorithms with respect to processing time.

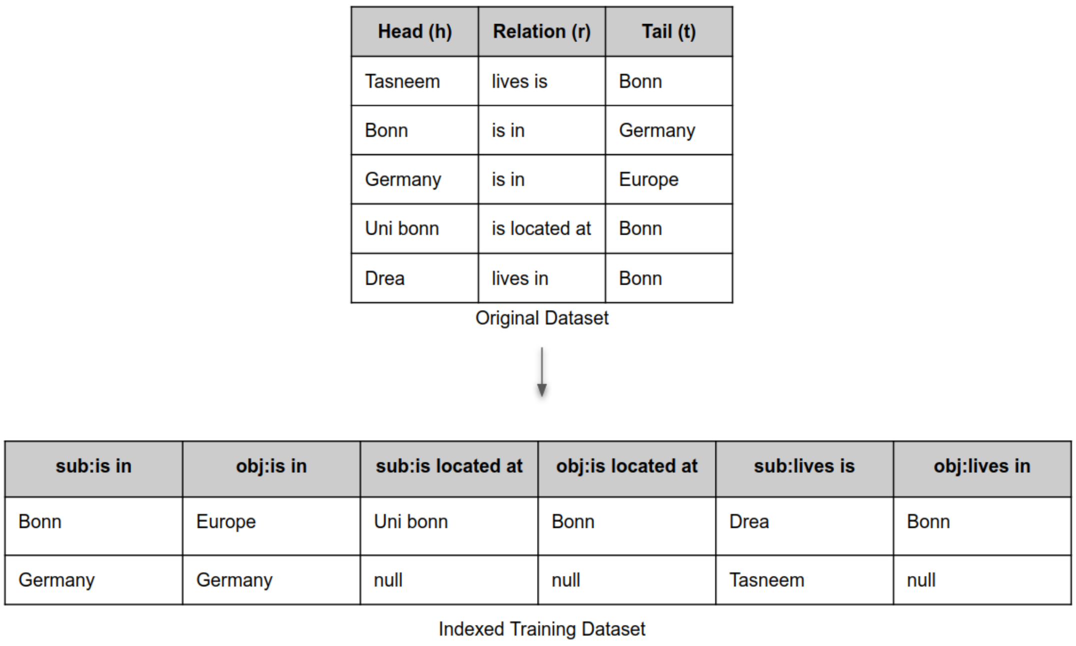
4.5.1 Motivation



Triple for corruption
Tasneem | lives in | Bonn
Corrupted triple
Drea | lives in | Bonn
Sample Training Dataset
Head (h) | Relation (r) | Tail (t)
Tasneem | lives is | Bonn
Bonn | is in | Germany
Germany | is in | Europe
Uni bonn | is located at | Bonn
Drea | lives in | Bonn
✖ = Corrupted triple does not exist in the training dataset
✓ = Corrupted triple exists in the training sample.

FIGURE 4.11: Example of checking if the corrupted triple exists in the training dataset

As indicated in Chapter 1, one of this thesis's primary tasks is to find a way to minimize the training time of the KGE models without compromising the results. Triple corruption or creating negative samples is one of the most important components of these models. RNS [15], DNS [9] and ADNS [11] mentioned in Chapter 3 do not guarantee that the generated negative sample is not true for the training dataset. Moreover, the KGE model's training algorithm with these Negative Sampling Algorithms includes a check box to refine the acceptable negative samples. Let's take a basic example of how the worst-case scenario of checking the generated triple in the training dataset will look. Figure 4.11 shows an example of the scenario. Head of the triple (*Tasneem*, *lives in*, *Bonn*) is selected for corruption. None of the negative sampling algorithms mentioned above can ensure that *Drea* will not be chosen as the replacement entity. In this case, negative triple (*Drea*, *lives in*, *Bonn*) is true for the training dataset, and to check it, the entire dataset has to be traversed. This issue motivated the idea that before creating the negative triple, if the KGE algorithm already knows which entities are bad and ignores them, the check box is unnecessary. Now the question comes, how to tell the algorithm which entities should be ignored for each relation while corrupting the head or the tail. Here the concept *Indexed Dataset* comes in the benefit of the training. For example, if the relation is *'lives in'* the corresponding column headers of the *Indexed Dataset* will be *'sub : lives in'* and *'obj : lives in'* as shown in Figure 4.12[7] . This advantage will be detailed in Chapter 5.



Head (h)	Relation (r)	Tail (t)
Tasneem	lives is	Bonn
Bonn	is in	Germany
Germany	is in	Europe
Uni bonn	is located at	Bonn
Drea	lives in	Bonn

sub:is in	obj:is in	sub:is located at	obj:is located at	sub:lives is	obj:lives in
Bonn	Europe	Uni bonn	Bonn	Drea	Bonn
Germany	Germany	null	null	Tasneem	null

FIGURE 4.12: Creation of indexed dataset from the original dataset

[7]For better understanding original dataset will be used as an example rather than the translated dataset, which is the real scenario for KGE training

4.5.2 Algorithm And Implementation

Algorithm 4 shows the pseudo-code of creating *Indexed Dataset* from the translated training dataset.

Algorithm 4 Creation of Indexed Dataset

```
Input Training set $S = \{(h, r, t)\}$
Returns Indexed dataset $I$

1:  $I \leftarrow \phi$                         ▷ Stores list of heads and tails per relation
2:  $R \leftarrow$ extract list of relations from $S$
3:  for $r \in R$ do
4:      $h \leftarrow$ List of unique heads for $r$
5:      $C_{\mathrm{h}} \leftarrow h$ as Column with header "sub:r"
6:      $t \leftarrow$ List of unique tails for $r$
7:      $C_{\mathrm{t}} \leftarrow t$ as Column with header "obj:r"
8:      $C \leftarrow$ Join $(C_{\mathrm{h}}, C_{\mathrm{t}})$
9:      $I \leftarrow$ Join $(I, C)$
10: end for
11: return $I$
```

4.5.2.1 Python Implementation

```python
1  def indexed_dataset(data):
2      relation = data['r'].unique()
3      indexed_data = pd.DataFrame()
4      for r in relation:
5          head = pd.DataFrame(data.loc[data['r'] == r]['h']
6                              .unique()).rename(columns={0: 'sub:' + r})
7          tail = pd.DataFrame(data.loc[data['r'] == r]['t']
8                              .unique()).rename(columns={0: 'obj:' + r})
9          temp = pd.concat([head, tail], axis=1)
10         indexed_data = pd.concat([indexed_data, temp], axis=1)
11     return indexed_data
```

FIGURE 4.13: Python code to create the indexed dataset

For Python implementation of Algorithm 4, as shown in Figure 4.13, the original dataset is taken as the input, and the indexed dataset is returned. For each relation, all the unique heads are extracted in a column that is renamed as $'sub:'$ concatenated with the relation as shown in lines 5 - 6. In the same process, for each relation, all the unique tails are extracted in a column that is renamed as *'obj :'* concatenated with the relation as shown in lines 7 - 8. Once the two columns are ready, they are added to the output data.

4.5.2.2 Spark Implementation

```
def IndexedDataSet(data:DataFrame, spark:SparkSession): (DataFrame) = {
    import spark.implicits._
    val relation = data.select("r").collect
                       .map(r => {r(0).asInstanceOf[String]})
    var indexedData = spark.emptyDataFrame
    var temp = spark.emptyDataFrame
    var flag = false
    relation.foreach(r => {
      val head = data.where($"r" ===  r).select("h").distinct
                     .withColumn("ID",
                                row_number.over(Window.orderBy("h")))
                     .withColumnRenamed("h", "sub:"+r)
      val tail = data.where($"r" === r).select("t").distinct
                     .withColumn("ID",
                                 row_number.over(Window.orderBy("t")))
                     .withColumnRenamed("t", "obj:"+r)
      if(head.count < tail.count){
        temp = tail.join(head,Seq("ID"),"left_outer") }
      else{
        temp = head.join(tail, Seq("ID"),"left_outer") }
      if(!flag){
        indexedData = temp
        flag = true
      }else{
        if(indexedData.count > temp.count){
           indexedData = indexedData.join(temp,
                                          Seq("ID"),"left_outer")
        }else{
           indexedData = temp.join(indexedData,
                                   Seq("ID"),"left_outer")} }})
    (indexedData.drop("ID"))
}
```

FIGURE 4.14: Spark code to create the indexed dataset

The *IndexedDataSet* function shown in Figure 4.14 is the Scala implementation of creating indexed dataset from the original dataset. *Left outer join* of the DataFrame is the key concept to construct the Figure 4.12. The original dataset and a SparkSession are the function parameters and *Indexed Dataset* is the output. For each relation in the train dataset, the list of unique head entities is extracted as a column, and the header of the column is renamed to *'sub :'* concatenated with the relation. A new column named *ID* with the row numbers is created to join the column, as shown in lines 9 - 11. Likewise, the columns with the unique tails with the column name *'obj :'* concatenated with the relation, and the row numbers are created. It is important to set the larger DataFrame on the left of the *.join* method with *left_outer* settings. Otherwise, the result will be cast to the size of the DataFrame on the left side. Information may get lost if the size of the DataFrame on the right side of the method is larger. Thus, a check block is kept to see which DataFrame is bigger. Everything is joined with the column *ID*, which is the only common column among all the DataFrames. This column will be dropped before returning the final result since it is created for the joining purpose. A mutable variable called *indexedData* is created at the beginning to hold the coming columns. A flag is kept to set the first outcome of the relation list directly inside the *indexedData* variable, and then the flag is set to *true* to execute the joining of the produced DataFrames as shown in lines 17 - 30.

Chapter 5

Indexed Negative Sampling Algorithm

This chapter proposes three variants of the Negative Sampling Algorithms with the help of *Indexed Dataset* created in Chapter 4. They can be named the *Indexed Negative Sampling Algorithms* in general. Multiple negative triples are generated per positive triple in these algorithms to help train the KGE models more efficiently [9][11]. General structure of the proposed algorithm is described in Section 5.1, pseudo-codes and their implementations of *Indexed RNS*, *Indexed DNS* and *Indexed ADNS*, the three proposed variants, are elaborated in Section 5.2, Section 5.3 and Section 5.4 respectively. An overall architecture of the framework is described in Section 5.5.

5.1 General Structure Of The Indexed Negative Sampling Algorithm

Algorithm 5 General Structure Of The Indexed Negative Sampling Algorithm

Input Triple (h, r, t), Entity set ε, Indexed dataset S_{I}
Returns Available entity list for triple corruption ε_{I}

```
1: Select head or tail to be replaced for triple corruption
2: $\varepsilon_{\mathrm{I}} \leftarrow \phi$                ▷ Initialize the indexed entity set
3: if Corrupt head $h$ then
4:     $\varepsilon_{\mathrm{h}} \leftarrow$ Select column with column name $'sub : r'$    ▷ $'sub : r'$ is the column
       where all the heads $h$ w.r.t. the relation $r$ of the triple are available
5:     $\varepsilon_{\mathrm{I}} \leftarrow \varepsilon$ - $\varepsilon_{\mathrm{h}}$
6: else if Corrupt tail $t$ then
7:     $\varepsilon_{\mathrm{t}} \leftarrow$ Select column with column name $'obj : r'$    ▷ $'obj : r'$ is the column
       where all the tails $t$ w.r.t. the relation $r$ of the triple are available
8:     $\varepsilon_{\mathrm{I}} \leftarrow \varepsilon$ - $\varepsilon_{\mathrm{t}}$
9: end if
10: return $\varepsilon_{\mathrm{I}}$
```

For each relation, *Indexed dataset* shown in Figure 4.12 will have two columns. One consisting of all the entities acting as the head, and the second consisting of all the entities acting as the tail per relation. Then Negative Sampling Algorithm takes an entity that is not present in the *Indexed Dataset* with respect to the relation instead of the entire entity set to select the replacement entity. Algorithm 5 shows the basic mechanism of this process. For example, to corrupt the head of the triple (*Tasneem*, *lives in*, *Bonn*) the proposed Negative Sampling Algorithm will first extract the list of entities by matching the column header as $'sub : lives\ in'$ from the table. Then the list will be excluded from the entire entity set. If any entity from this set is chosen to replace the head, there is no chance that the created triple will be true for the training dataset. Thus, from the current KGE models, the checking box while creating the negative sample can be discarded.

Special case: The inquisitive minds may have the concern what will be the scenario when ε_{h} or ε_{t} from Algorithm 5 will be equal to ε. In this case, the available pool ε_{I} used to create a good corrupted triple will be empty.

Solution: If this exception occurs, then the general rule of triple corruption will be followed by the algorithm of choosing an entity. This process will be elaborated in Section 5.2, Section 5.3 and Section 5.4.

5.2 Indexed Random Negative Sampling (Indexed RNS)

Algorithm 6 Indexed Random Negative Sampling

Input Training set $S = \{(h, r, t)\}$, indexed training set $\mathrm{S}_{indexed}$, entity and relation sets ε and $\mathcal{R}$, batch size b, number of negative samples C.
Returns Given a batch $B = \{(h, r, t)_k\}$, $1 \leq k \leq b$, return corrupted triples B' for each triple in the batch.

```
1:  $B' = \phi$                                         $\triangleright$ Stores the negative triples for batch $B$
2:  for $triple = (h, r, t) \in$ batch $B$ do
3:      Use Random distribution to conclude that $t$ should be corrupted
4:      $\varepsilon_{\text{indexed}} \leftarrow$ extract Set($t_{\text{r}}$) from $S_{\text{indexed}}$
                          $\triangleright$ $t_{\text{r}}$ denotes set of all the $t$'s w.r.t. $r$ from $\mathrm{S}_{indexed}$
5:      $\varepsilon' \leftarrow \varepsilon - \varepsilon_{\text{indexed}}$
                          $\triangleright$ $\varepsilon'$ denotes set of all the entities which are not as $t$ for $r$
6:      $negative\_entities$ = Random($\varepsilon'$, $C$)
                          $\triangleright$ Randomly chooses $C$ entities from $\varepsilon'$ to create negative triples
7:      $N \leftarrow$ Create($h, r, negative\_entities$)
                          $\triangleright$ $N$ denotes the generated $C$ negative triples for $triple$
8:      $B' \leftarrow B' \cup N$
9:  end for
10: return $B'$
```

In the *Indexed RNS* as illustrated in Algorithm 6, the entity set extracted from Algorithm 5 will be the pool of entities to choose from randomly. Algorithm 6 is an example of when the tail should be corrupted. The rest of the procedure is similar to RNS. Another addition to the *Indexed RNS* from RNS [15] is, this algorithm creates multiple negative triples for each positive triple. This implementation will be elaborated in Figure 5.1.

Implementation: The function *indexed_random* from Figure 5.1 takes four parameters. The positive batch from the training, the number of negative triples to be generated per positive triple (C), the number of entities, and the *Indexed Dataset*. The output of this function is the corresponding C numbers of negative triples per positive triple. For each triple, an empty array N is created with the dimension of $[C \times 3]$. To store triple (h, r, t) 3 is used. Created negative samples will be stored here. *corrupt_index* , as shown in line 6, basically indicates whether the head or the tail should be replaced. Since the head is in index 0 and tail is in index 2 of the triple, with the help of *numpy* library, the probability of randomly choosing any

```
1  def indexed_random(positive_batch, C, number_of_entities, indexed_table):
2      negative_batch = []
3      t = positive_batch.copy()
4      for triple in t:
5          N = np.zeros((C, 3), dtype=int)
6          corrupt_index = np.random.choice([0, 2], p=[0.5, 0.5])
7          candidate = triple[corrupt_index]
8          all_entity = np.arange(0, number_of_entities, 1)
9          if corrupt_index == 0:
10             des_str = 'sub:' + str(triple[1])
11         else:
12             des_str = 'obj:' + str(triple[1])
13         temp_data = [i for i in np.array(indexed_table[des_str])
14                      if ~np.isnan(i)]
15         all_unique_entity = np.setxor1d(all_entity, temp_data)
16                                 .astype(int)
17         try:
18             N = np.random.choice(all_unique_entity, C, replace=False)
19         except:
20             all_entity_candidate_removed = np.delete(all_entity,
21                               list(all_entity).index(candidate))
22             N = np.random.choice(all_entity_candidate_removed,
23                                               C, replace=False)
24         temp_triple = np.copy(triple)
25         temp_matrix = np.tile(temp_triple, (C, 1))
26         for i in range(len(temp_matrix)):
27             temp_matrix[i][corrupt_index] = N[i]
28         for i in range(len(temp_matrix)):
29             negative_batch.append(temp_matrix[i])
30     return np.array(negative_batch)
```

FIGURE 5.1: PyTorch code for Indexed RNS

one of the indices is given as 50-50. Then the corresponding entity is extracted as *candidate* in line 7 of the Figure 5.1. As mentioned in Chapter 4, the strings of entities and relations are represented as numerical values. Thus, a set of all entities can be easily created with the number of the entities (line 8 of the code). Depending on replacing either the head or the tail, the available pool of the entities is calculated from lines 9 - 15 of the Figure 5.1. In line 13, a list comprehension is enacted, which returns the *Indexed Data* as a list without any *null* values. Before choosing the replacement entity randomly, the special case mentioned at the beginning of this chapter must be handled. For which a try-catch block is executed in the implementation. If an error is thrown, the available pool will be calculated by subtracting the *candidate* from the entire entity set *all_entity* as shown from lines 17 - 23. Method *np.random.choice* supports choosing multiple elements from a set at once. Thus *all_unique_entity* or *all_entity_candidate_removed* is set as one of the parameters of this method, as shown in line 18 and line 22 to indicate the available pool. The number of the chosen elements C and $replace = False$, which restricts the repetition of the chosen elements, are set as the parameters of the random choice method. Finally, the C numbers of negative triples are created and appended to the negative batch sample. This negative batch will be the C times the positive batch size because for each triple C numbers of negative samples are generated.

5.3 Indexed Distributional Negative Sampling (Indexed DNS)

Indexed Dataset is injected to DNS[9] to create another variant of the Negative Sampling Algorithm. This proposed algorithm will be named Indexed Distributional Negative Sampling or *Indexed DNS* as shown in Algorithm 7. The probability table explained in Chapter 4 which contains the fraction of head per tail and tail per head for each relation, is utilized in this algorithm. This table is set as the probability parameter inside the *Bernoulli distribution*[54] to select whether to corrupt head or tail per triple. Algorithm 7 examples when the tail is selected for triple corruption. The main component of *Indexed DNS* is calculating the $cosine - similarity$ [53] between the candidate, tail in this case, and the initialized embeddings of the available pool of entities. This will be stored as a matrix M. The probability of taking an entity will be calculated while enumerating the matrix M. One crucial factor of the original DNS algorithm is, to create C numbers of negative samples, the enumeration through the matrix M will be executed C times. This impacts the training time of the KGE models. However, *Indexed DNS* improves this training time by removing the chances of generated negative samples true for the train dataset. This performance issue is explained comprehensively in Chapter 7. There is a possibility that the entity set provided from following the Algorithm 5 might throw some exceptions for being empty. This special case is handled in the implementation.

Algorithm 7 Indexed Distributional Negative Sampling

Input Training set S = $\{(h, r, t)\}$, indexed training set $S_{indexed}$, entity and relation sets ε and $\mathcal{R}$, batch size b, number of negative samples C.
Returns Given a batch $B = \{(h, r, t)_k\}$, $1 \leq k \leq b$, return corrupted triples B' for each triple in the batch.

```
1: $B' = \phi$                                  ▷ Stores the negative triples for batch $B$
2: for $triple = (h, r, t) \in$ batch $B$ do
3:    $N = \phi$                                ▷ Stores negative triples for $triple$
4:    Use Bernoulli sampling to conclude that $t$ should be corrupted.
5:    $\varepsilon_{\text{indexed}} \leftarrow$ extract Set($t_{\text{r}}$) from $S_{\text{indexed}}$
                    ▷ $t_{\text{r}}$ denotes set of all the $t$'s w.r.t. $r$ from $S_{indexed}$
6:    $\varepsilon' \leftarrow \varepsilon - \varepsilon_{\text{indexed}}$
7:    $M \leftarrow$ cosine-sim$(t, \varepsilon')$
8:    for (i,m) $\in$ enumerate(M) do
9:       $p_{\text{accept}} \leftarrow max(0, m)$
10:      $p_{\text{reject}} \leftarrow 1 - p_{\text{accept}}$
11:      With probability $p_{\text{accept}}$ choose entity i.
12:      if entity i is chosen then
13:         $N \leftarrow N \cup (h, r, \varepsilon'[i])$ [Continue $C$ times]     ▷ $\varepsilon'$[i] denotes entity i
14:      end if
15:   end for
16:   $B' \leftarrow B' \cup N$
17: end for
18: return $B'$
```

Implementation: The PyTorch code shown in Figure 5.2 is the implementation of Algorithm 7. There are six parameters for this function. They are the positive batch of triples (h, r, t) and the number of negative triples C to be generated from each positive triple, the number of the entities, the probability table and the *Indexed Dataset* mentioned in Chapter 4. The last input parameter of the function *indexed_DNS* is the *model*, which gives access to the initialized embeddings of entities by the KGE models. A $[C\times$ 3] matrix N is created to store the negative triples. variable *current_relation* extracts the relation r of the triple. Head per tail (*hpt*) and tail per head (*tpt*) for the corresponding relation are extracted from the probability table. Using these *hpt* and *tpt*, the head corruption probability as shown in the Equation 4.3 and tail corruption probability as shown in Equation 4.4 are calculated from lines 7 - 10 in the Figure 5.2. Using these calculated probability *Bernoulli distribution* determines which entity should be replaced, head or tail of the triple. Once the *candidate* is selected, the corresponding vector embedding is extracted in line 15 from Figure 5.2 with the command

$model.get\_embedding(candidate)$, which gives access to the corresponding embeddings of entities or relations initialized by the KGE model. A check block is kept to investigate whether the available pool created from the *Indexed Dataset* is empty or not. If the pool's size is 0, the entire entity set minus the *candidate* is considered the available pool, and their corresponding vector embeddings are extracted as shown from lines 23 - 30. Afterward, the cosine similarity of the candidate with respect to the available pool - *all_entity_embedding* is calculated with the help of Torch's convolution functions as shown in line 31 and stored in a matrix M. The dimension of M is [size of the available pool $\times$ 1], which means there will be one scalar value from cosine similarity for each entity.

While enumerating through M, for each position i which denotes the entity ID is set in the *corrupt_index* in line 37. Probability for accepting and rejecting the entity i is calculated lines 38 - 40. Now, there is a possibility that the value of the cosine similarity is negative. For this cause, $max(0, cosinesimilarity)$ is taken as shown in lines 38 - 39. The matrix M is in Torch and *random.choice* is a *numpy* function. Thus M is converted by the function $M.detach().cpu().numpy$ to *numpy*. Thereafter, the probabilities are passed through the *random.choice* to decide whether to take or not take the replacement entity, as shown in lines 43 - 47. One drawback of both DNS and *Indexed_DNS* is that the entire process is rerun C times for generating C numbers of negative triple per positive triple. At the end of traversing, the negative triples are appended to a matrix, and the function returns the matrix.

```
def indexed_DNS(positive_batch, C, model, number_of_entities,
                probability_tabl, indexed_table): negative_batch = []
  t = positive_batch.copy()
  for triple in t:
    N = np.zeros((C, 3), dtype=int)
    current_relation = triple[1]
    tph = probability_tabl['tph'][current_relation]
    hpt = probability_tabl['hpt'][current_relation]
    head_corruption_probability = tph / (tph + hpt)
    tail_corruption_probability = hpt / (tph + hpt)
    corrupt_index = np.random.choice([0, 2],
                    p=[head_corruption_probability,
                       tail_corruption_probability])
    candidate = triple[corrupt_index]
    candidate_embedding = model.get_embedding(candidate).view(1, -1)
    all_entity = np.arange(0, number_of_entities, 1)
    if corrupt_index == 0:
      des_str = 'sub:' + str(triple[1])
    else:
      des_str = 'obj:' + str(triple[1])
    inter_data =[i for i in np.array(indexed_table[des_str])if~np.isnan(i)]
    all_unique_entity = np.setxor1d(all_entity, inter_data).astype(int)
    if all_unique_entity.size == 0:
      all_entity_candidate_removed = np.delete(all_entity,
                                  list(all_entity).index(candidate))
      all_entity_embedding = model.get_vectorized_embedding(
                                      all_entity_candidate_removed)
    else:
      all_entity_embedding = model.get_vectorized_embedding
                                     (all_unique_entity)
    M = F.cosine_similarity(all_entity_embedding, candidate_embedding)
    c = 0
    for i, m in enumerate(M):
      if c == C:
          break
      clone_triple = triple.copy()
      clone_triple[corrupt_index] = i
      p_accept = torch.max(torch.tensor(0.0).cuda(),
                      m.cuda()).detach().cpu().numpy()
      p_reject = 1 - p_accept
      take = np.random.choice(['take', 'not_take'], 1,
                          p=[p_accept, p_reject])
      if (take == 'take') and (c < C):
          N[c] = clone_triple
          c += 1
      else:  continue
      for index in range(len(N)):
          negative_batch.append(np.array(N[index]))
    return np.array(negative_batch)
```

Figure 5.2: PyTorch code for Indexed DNS

5.4 Indexed Affinity Dependent Negative Sampling (Indexed ADNS)

Algorithm 8 Indexed Affinity Dependent Negative Sampling

Input Training set $S_{(h,r,t)}$, indexed training set S$_{indexed}$, entity Set ε, relation set $\mathcal{R}$, batch size b, number of negatives C

Returns Given a batch $B = \{(h, r, t)_k\}$, $1 \leq k \leq b$, return corrupted triples B' for each triple in the batch.

```
1: $B' = \phi$                                   ▷ Stores the negative triples for batch $B$
2: for $triple = (h, r, t) \in$ batch $B$ do
3:     $N = \phi$                                 ▷ Stores negative triples for $triple$
4:     Use Bernoulli sampling to conclude that $t$ should be corrupted.
5:     $\varepsilon_{\text{indexed}} \leftarrow$ extract Set($t_{\text{r}}$) from $S_{\text{indexed}}$
                              ▷ $t_{\text{r}}$ denotes set of all the $t$'s w.r.t. $r$ from S$_{indexed}$
6:     $\varepsilon' = \varepsilon$ - $\varepsilon_{\text{indexed}}$            ▷ subtract $\varepsilon_{\text{indexed}}$ from total entity set
7:     $M_{\text{score}} = CosineSimilarity(candidate, \varepsilon')$
8:     $M_{\text{score}} = max(0, M_{\text{score}})$
9:     for $i \in length(M_{\text{score}})$ do
10:        $probability\_fitness_i = M_{\text{score}_\text{i}} \div \sum_j M_{\text{score}_\text{j}}$
                                              ▷generate the fitness vector
11:    end for
12:    $selected\_entities = random\_choice(\varepsilon', C, probability\_fitness)$
13:    $N =$
       $FormNegative(t,\ selected\_entities,\ candidate\_position)$
                                              ▷ $C$ negative triples per positive
14:    $B' \leftarrow B' \cup N$
15: end for
16: return $B'$
```

Indexed Affinity Dependent Negative Sampling or *Indexed ADNS* is a variant of ADNS [11]. The *Indexed Dataset* is injected into the original algorithm to improve the training time of the KGE models. ADNS itself is a variant of DNS, which includes an additional fitness function. *Indexed ADNS* preserves this additional function well by selecting multiple entities to create multiple negative triples per positive triple at once. Principally this fitness function is the probability of choosing each element among the available pool of entities. In *Indexed ADNS*, first, the cosine similarity is calculated with respect to the chosen replacement entity with *Bernoulli distribution* over the probability table similar to DNS explained in Section 5.3. The matrix dimension is [available pool size $\times$ 1] because for each entity in the available pool, one scalar value is set by the cosine similarity calculation. These scalar values can be negative, and in this case, calculating the fitness function will result in an error. Thus max of 0 and the value of cosine similarity is taken. The fitness function's use solves the problem for both *Indexed DNS* and DNS of repeating the same process depending on the number of negative triples since it allows to choose multiple entities with similar meaning at once. As well as injecting the indexed table shows a promising improvement of the training time for *Indexed ADNS*, which will be illustrated in Chapter 7.

Implementation: Input parameters of the function *indexed_ADNS* shown in Figure 5.3 are the positive batch, the number of negative triples C per positive triple, number of the entities, the probability table, the *Indexed Dataset*, and finally the *model* which is used to extract embeddings for cosine similarity calculation. Lines 4 - 33 of Figure 5.3 is similar to *Indexed DNS* explained in Section 5.3. This block of code basically selects if head or tail should be replaced with *Bernoulli distribution*. The available pool is created with respect to the replacement entity by using the *Indexed Dataset*. Also, the vector embedding of the *candidate* and the available pool is extracted. Finally, the cosine similarity using *numpy* library is calculated between the candidate and the available pool. Having negative values in the matrix containing the cosine similarity cannot be afforded. Thus, all the negative values are set to 0. The special case mentioned in Section 5.5 is handled from lines 28 - 33.

However, the fitness function can also throw some exceptions. Since the sum of the cosine similarity is the divisor of the fitness function, if all the values of the matrix M are 0 then an arithmetic exception will occur. In this scenario, another check block is kept from lines 36 - 42. This basically depicts that, if there is an arithmetic error, C numbers of entities will be chosen randomly from the entity set where the candidate is subtracted. Again, as mentioned previously, The function *random.choice* is a *numpy* function. Thus, the Torch value is converted into *numpy* as shown in lines 37 - 39. C numbers of negative triples per positive triple are appended in the *negative_batch*, and this will be the return value of the function.

```
def indexed_ADNS(positive_batch, C, model, number_of_entities,
                 probability_table, indexed_table):
    negative_batch = []
    t = positive_batch.copy()
    for triple in t:
        current_relation = triple[2]
        tph = probability_table['tph'][current_relation]
        hpt = probability_table['hpt'][current_relation]
        head_corruption_probability = tph / (tph + hpt)
        tail_corruption_probability = hpt / (tph + hpt)
        corrupt_index = np.random.choice([0, 1],
                                    p=[head_corruption_probability,
                                       tail_corruption_probability])
        candidate = triple[corrupt_index]
        candidate_embedding = model.get_embedding(candidate)
                                       .view(1, -1)
        all_entity = np.arange(0, number_of_entities, 1)
        if corrupt_index == 0:
            des_str = 'sub:' + str(triple[2])
        else:
            des_str = 'obj:' + str(triple[2])
        inter_data = [i for i in np.array(indexed_table[des_str]) if
                      ~np.isnan(i)]
        all_unique_entity = np.setxor1d(all_entity, inter_data)
                                  .astype(int)
        all_entity_candidate_removed = np.delete(all_entity,
                                     list(all_entity).index(candidate))
        if all_unique_entity.size == 0:
            all_entity_embedding = model.get_vectorized_embedding(
                                        all_entity_candidate_removed)
        else:
            all_entity_embedding = model.get_vectorized_embedding(
                                        all_unique_entity)
        M = F.cosine_similarity(all_entity_embedding, candidate_embedding)
        M[M < 0] = 0
        try:
            N = np.random.choice(all_unique_entity, C,
                    p=M.detach().cpu().numpy()
                      / sum(M.detach().cpu().numpy()),
                    replace=False)
        except:
            N = np.random.choice(all_entity_candidate_removed, C)
        temp_triple = np.copy(triple)
        temp_matrix = np.tile(temp_triple, (C, 1))
        for i in range(len(temp_matrix)):
            temp_matrix[i][corrupt_index] = N[i]
        for i in range(len(temp_matrix)):
            negative_batch.append(temp_matrix[i])
    return np.array(negative_batch)
```

FIGURE 5.3: PyTorch code for Indexed ADNS

5.5 Overall Architecture Of The Approach

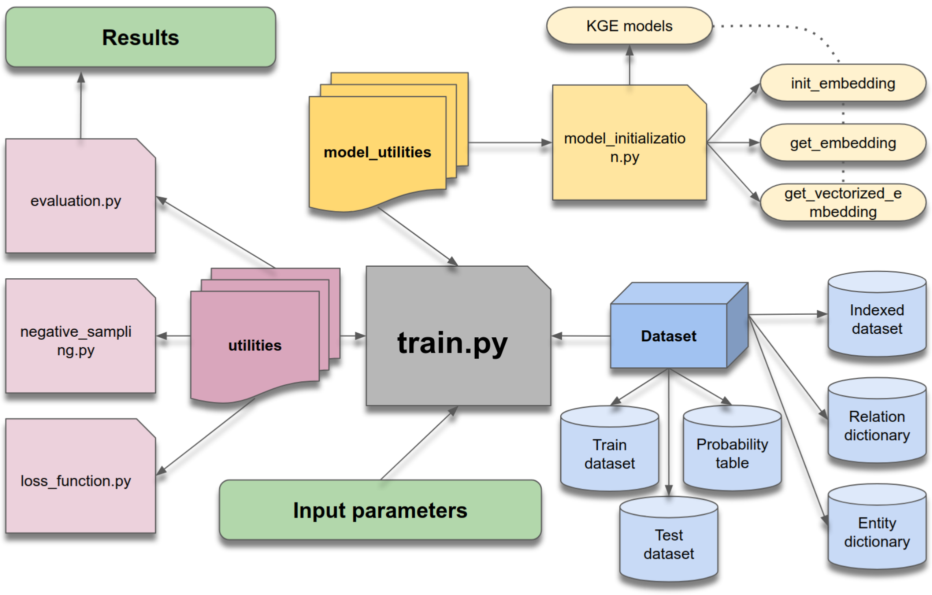


FIGURE 5.4: KGE Framework

The KGE framework, illustrated in Figure 5.4, adapted from another functioning framework [55], is used to test the efficiency of the three proposed Indexed Negative Sampling Algorithms. It has two packages, one main file and one folder for the datasets. The names of the two packages are *utilities* and *model_utiliteis*. The package *model_utilities* has a class file where the KGE models are implemented as shown in Figure 5.4. The package *utilities* has three classes: one for calculating the loss, one for creating the negative samples, and one for evaluating the trained models. The implemented algorithms explained in Section 5.2, Section 5.3 and Section 5.4 are set inside the *utilities* package's *negative_sampling.py* file. Alongside, the files generated from the pre-processing phase explained in Chapter 4 are kept inside the dataset folder. For results, the *train.py* file is executed with the proper parameter, which will be mentioned in Chapter 7.

Chapter 6

Distributed KGE Models

This chapter focuses on the distributed approach of three popular KGE models with Spark framework and BigDL library. This approach is inspired by the implementation of TransE [15] which is still in the development phase[1] by the SANSA-Stack [13] : an open-source structured data processing engine[2]. The approach of this thesis shows prospective aptitude compared to the SANSA team's version and solves a bug[3] of the current system which will be elaborated in Chapter 7. Not only that, this chapter illustrates the implementations of two other KGE models: DistMult and ComplEx, which are other contributions of this thesis.
Section 6.1 will describe the basic concepts of the selected KGE models and Section 6.2 explains the basic structure of the distributed models. Implementations of distributed TransE, DistMult and ComplEx are elaborated in Section 6.3, Section 6.4 and Section 6.5 respectively. The architecture of the distributed KGE model's framework is explained in Section 6.6.

[1]https://github.com/SANSA-Stack/Archived-SANSA-ML/tree/master/sansa-ml-spark
[2]https://github.com/SANSA-Stack
[3]https://github.com/SANSA-Stack/Archived-SANSA-ML/issues/18

6.1 Selected KGE Models

To test the distributed approach, three KGE models are selected. They are TransE [15], DistMult [16] and ComplEx [17]. RNS, as discussed in Chapter 3, is used as the selected Negative Sampling Algorithm. Margin Ranking Loss is used to calculate the loss function. AdaGrad [8] optimizer is employed to train the KGE models instead of the basic SGD [6] optimizer.

6.1.1 TransE

TrasnE [15] translates the embeddings of the head (h) towards the tail (t) with respect to the relation (r) of the triple (h, r, t) to a lower dimension. This model uses norm $L1$ or $L2$ ($||\ .\ ||$) as the scoring function which calculates the dissimilarity measure d, between translated head and tail, shown in Equation 6.1.

$$f_{\text{TransE}} = d(h + r, t) = ||h + r - t|| \tag{6.1}$$

This scoring function is then used on positive triple (h, r, t) and negative triple (h', r, t') inside the loss function as shown in Equation 6.2.

$$L = \sum_{(h,r,t)\in S} \sum_{(h',r,t')\in S'_{\text{(h, r, t)}}} [\lambda + d(h + r, t) - d(h' + r, t')]_{+} \tag{6.2}$$

Equation 6.2 is known as Margin-Based Ranking Criterion [56] where λ is the margin, h' and t' represent random replacement of the h or t for a particular triple.

6.1.2 DistMult

DistMult [16] is a bilinear diagonal model which uses the trilinear dot product by multiplying the head (h) and the tail (t) with their corresponding relational matrix W_{r} as the scoring function.

$$f_{\text{DistMult}} =< E_{\text{h}}, W_{\text{r}}, E_{\text{t}} > \tag{6.3}$$

Equation 6.3 is the scoring function of DistMult where E_{h} and E_{t} are the embeddings of head and tail of the triple and W_{r} is a diagonal matrix of the relation. DistMult uses margin-based ranking criterion shown in Equation 6.2, as the loss function similar to TransE. Regularization [16] is used to improve the model.

6.1.3 ComplEx

ComplEx [17] is an extension of DistMult This model has real R_{e} and imaginary I_{m} parts for both entities and relation embeddings. The ComplEx uses the trilinear Hermitian dot product as the scoring function as shown in Equation 6.4.

$$\begin{aligned} f_{\mathrm{ComplEx}} &= R_{\mathrm{e}} < E_{\mathrm{h}}, W_{\mathrm{r}}, \overline{E_{\mathrm{t}}} > \\ &=< R_{\mathrm{e}}(W_{\mathrm{r}}), R_{\mathrm{e}}(E_{\mathrm{h}}), R_{\mathrm{e}}(E_{\mathrm{t}}) > \\ &+ < R_{\mathrm{e}}(W_{\mathrm{r}}), I_{\mathrm{m}}(E_{\mathrm{h}}), I_{\mathrm{m}}(E_{\mathrm{t}}) > \\ &+ < I_{\mathrm{m}}(W_{\mathrm{r}}), R_{\mathrm{e}}(E_{\mathrm{h}}), I_{\mathrm{m}}(E_{\mathrm{t}}) > \\ &- < I_{\mathrm{m}}(W_{\mathrm{r}}), I_{\mathrm{m}}(E_{\mathrm{h}}), R_{\mathrm{e}}(E_{\mathrm{t}}) > \end{aligned} \tag{6.4}$$

In Equation 6.4 E_{h} and E_{t} are the embeddings of the head and the tail of the triple and W_{r} is the relation embedding space. ComplEx also uses the loss function and the regularization similar to DistMult.

6.2 General Structure Of The Distributed KGE Model

Algorithm 9 General training structure of the distributed KGE model

```
 1: Create Spark Session
 2: Initialize BigDL Engine
 3: Read data
 4: Initialize embeddings
 5: for each epoch do
 6:     Function $TrainModel$ :        ▷Function that returns the loss of the batch
 7:         Normalize entity embedding
 8:         Get positive sample
 9:         Set negative sample
10:         Calculate Score
11:         Calculate Loss
12:     Optimize $TrainModel$
13: end for
```

The general structure of the distributed KGE model's training is shown in Algorithm 9. The differences among TransE, DistMult, and ComplEx models are calculating the score function and the loss function. The rest of the procedure is the same for all. The common processes will be discussed in this Section and the score loss calculation of the different KGE models will be discussed in Section 6.3, Section 6.4 and Section 6.5.

1. Create Spark session and initialize BigDL engine: Initializing the Engine of BigDL with *Engine.init* can contribute to a good performance of the models by setting the environment variables correctly[4]. To do so, *SparkContext* needs to be created by using the *SparkConf* returned by *Engine* as shown in Figure 6.1. In the spark configuration (*SparkConf*[5]), the app name is set with respect to the model's name, and to run the Spark in 4 cores of the machine locally, $setMaster(local[4])$ is set. *SparkSession* is also created with *SparkConf* for working with RDDs as shown in Figure 6.1.

[4]https://bigdl-project.github.io/0.10.0/APIGuide/Engine/
[5]https://spark.apache.org/docs/latest/configuration.htmlspark-properties

```
    val conf = Engine.createSparkConf().setAppName("TransE")
                     .setMaster("local[4]")
    val sc = new SparkContext(conf)
    val spark = SparkSession.builder.config(conf).getOrCreate()

    Engine.init

```

FIGURE 6.1: Create SparkSession and initialize Engine of BigDL

2. Read data: This step follows the basics of reading RDDs with Spark. Since the files are saved in *tsv* format, for reading the RDDs '$\backslash t$' is set as the separation settings. For training the model, a minimum of three datasets are required to be read. They are the train dataset, list of entities, and list of relations as demonstrated in Figure 6.2.

```
    val trainSet     = spark.read.option("sep", "\t")
                            .csv(resourceDir + "train.tsv").rdd
    val entityList   = spark.read.option("sep", "\t")
                            .csv(resourceDir + "entityToID.tsv")
                            .select("_c1").rdd
                            .map(r => r(0).asInstanceOf[Object])
                            .collect
    val relationList = spark.read.option("sep", "\t")
                            .csv(resourceDir + "relationToID.tsv")
                            .select("_c1").rdd
                            .map(r => r(0).asInstanceOf[Object])
                            .collect

```

FIGURE 6.2: Read data for Distributed KGE models

3. Initialize embeddings: As shown in the Algorithm 9, this stage is the beginning of executing the KGE models. And the first step is to initialize the embeddings of relation and entity. For the distributed approach of the KGE models, entity embeddings and relation embeddings are kept as one Tensor to benefit Optimization. The reason will be elaborated later in Chapter 7. As shown in Figure 6.3, embeddings for entity and relation are created as Tensors with the *entityListLength*, *relationListLength* and the dimension (k). Their values are set randomly within the range mentioned in TransE. Relation embeddings are normalized before starting the epochs of training. Thus, as shown in line 5 in the Figure 6.3, the relation Tensor is normalized with a built-in function $Normalize(2).forward(Tensor)$ where 2 indicates L_2 normalization. As mentioned before, for the benefit of optimizing the models with BigDL, the embeddings are kept in one Tensor. Therefore, a function *Join* as shown in Figure 6.4 is implemented, which combines the entity Tensors and the relation Tensor. Entity Tensor is set before Relation Tensor. This condition is important because the relation embedding's position will start from (index + length of entity list). In brief, the *Join* function creates a Tensor with the summation of the lengths of entity and relation list and the dimension (k). While traversing through it, the rows are updated with entity Tensor or relation Tensor depending on the position of the combined Tensor index, as shown in lines 6 - 11 of the Figure 6.4.

```
1    def CreateEmbedding (entityListLength:Int, relationListLength:Int,
2                         k:Int) : (Tensor[Float]) = {
3      val entityEmbedding   = Tensor(entityListLength, k)
4                                   .rand(-6/Math.sqrt(k), 6/Math.sqrt(k))
5      val relationEmbedding = Normalize(2).forward(
6                                  Tensor(relationListLength, k)
7                                  .rand(-6/Math.sqrt(k), 6/Math.sqrt(k)))
8      var embedding = Join(entityEmbedding, entityListLength,
9                           relationEmbedding, relationListLength, k)
10     (embedding)
11   }
```

FIGURE 6.3: Initialize embeddings

```
1  def Join (entityEmbedding:Tensor[Float], entityListLength:Int,
2              relationEmbedding:Tensor[Float], relationListLength:Int,
3              k:Int) : (Tensor[Float]) = {
4      var embedding = Tensor(entityListLength + relationListLength, k)
5      for(count <- 1 to (entityListLength + relationListLength)){
6        (count<=entityListLength) match {
7          case true  => embedding.update(count,
8                                         entityEmbedding.apply(count))
9          case false => embedding.update(count,
10                                        relationEmbedding
11                                        .apply(count-entityListLength))
12       }
13     }
14     (embedding)
15 }
```

FIGURE 6.4: Join embeddings

```
1     val optim = new Adagrad(learningRate = learningRate)
2     for(epoch <- 1 to nEpoch){
3       //Function for updating the embedding
4       def Update(x: Tensor[Float]) = {
5         (TrainModel(trainSet, entityList, entityListLength,
6                     relationListLength, embedding, L,
7                     gamma, k, trainSize), x)
8       }
9       optim.optimize(Update, embedding)
10    }
```

Figure 6.5: Run n numbers of training epochs

4. Run n number of epochs (nEpoch) for training the model: Once the embeddings are initialized, the epoch starts training the KGE models as shown in Figure 6.5. This part is very crucial. First the optimizer is initialized as *optim* with a *learningRate* as shown in line 1 of Figure 6.5. Inside the loop for running the epochs to train the models, *optim* is used to call the optimizer. *optim.optimizer* (*function*, *input tensor*) is the structure of calling the optimizers in BigDL[6]. The *function* defines how a value is calculated with respect to an equation over the *input tensor* and returns the results and the gradients of the equation as a Tensor. From the KGE model's perspective, the float value from the loss calculation can be considered as the loss and the embedding itself as the gradient. Therefore, inside the loop for running the epochs, a function called *Update* is created, which returns the model's loss value - the outcome of the *TrainModel* function, and the embeddings of the entities and relations. This approach is inspired by the SANSA team's current implementation[7]. However, a bug[8] has been reported for this implementation. Executing the training in this manner sorts out the issue and improves the approach which will be discussed in Chapter 7.

4.1 Function TrainModel: Function *TrainModel* is called in each epoch inside the optimizer as shown in Figure 6.5. This function basically takes the positive sample and creates the corresponding negative sample. Thereafter calculates the score and loss of the model. The loss is returned. However, before all these, the entity embeddings are normalized with the function name *Normalization* as shown in Figure 6.7.

[6]https://bigdl-project.github.io/0.10.0/APIGuide/Optimizers/Optim-Methods/adagrad
[7]See footnote 2
[8]See footnote 3

```scala
def TrainModel(trainSet: RDD[Row], entityList:Array[Object],
               entityListLength:Int, relationListLength:Int,
               embedding: Tensor[Float], L:Int, gamma:Float,
               k: Int, trainSize: Long) : (Float) = {
    embedding.copy(Normalization(embedding, entityListLength,
                                 relationListLength, k))
    val positiveSample = trainSet.collect
    val negativeSample = positiveSample
                          .map(row => {NegativeSample(row,entityList)})
    val positiveDistance = Score(positiveSample,
                                 entityListLength,
                                 embedding, L, gamma)
    val negativeDistance = Score(negativeSample,
                                 entityListLength,
                                 embedding, L, gamma)
    (LossFunction(positiveDistance,negativeDistance,gamma,k,trainSize))
}
```

FIGURE 6.6: Function to train the distributed model

4.1.1 Normalization of the embedding: This step aims to improve the models by imposing a unit embedding length after each gradient step. As indicated before, the distributed approach has one embedding combining entity and relation embeddings. Hence to normalize only the entity embedding, the total embedding must be split to extract the entity embeddings. The method $Tensor.split(size,\ dim)$ in BigDL basically splits the Tensor into a table of Tensors of the *size* along the dimension *dim*. As shown in line 4 of Figure 6.7, the total embedding is split by the rows with the entity list's length since entity embedding is put first in the total embedding. Then after calling the built-in L_2 normalize function from BigDL on the extracted entity embedding, the split Tensors are joined by the *Join* function shown in Figure 6.4.

```scala
1    def Normalization(embedding:Tensor[Float], entityListLength:Int,
2                      relationListLength:Int, k:Int) :
3                      (Tensor[Float]) = {
4      val temp = embedding.split(entityListLength, 1)
5      (Join(Normalize(2).forward(temp(0)), entityListLength, temp(1),
6                              relationListLength, k))
7    }
```

FIGURE 6.7: Normalization function for distributed KGE models

4.1.2 Negative sample generation: The distributed approach implements the RNS as shown in Figure 6.8. This function executes the RDDs, and since RDDs are immutable in Spark, a new set of RDD for the negative sample is created. RDD[Row] of the positive sample is mapped with a new Row by randomly corrupting the head or tail from the entity list where the replacement candidate is removed as shown in lines 7 - 11.

```
1   def NegativeSample(row:Row, entityList:Array[Object]):
2                     (Row) = {
3     val sub  = row.get(0).asInstanceOf[Object]
4     val pred = row.get(1).asInstanceOf[Object]
5     val obj  = row.get(2).asInstanceOf[Object]
6     if(Random.nextInt(2) == 0){ //change head
7       val subList = entityList.diff(Array(sub))
8       val negSub =  subList(Random.nextInt(subList.length))
9       (RowFactory.create(negSub, pred, obj))
10    }else{
11      val objList = entityList.diff(Array(obj))
12      val negObj =  objList(Random.nextInt(objList.length))
13      (RowFactory.create(sub, pred, negObj))
14    }
15  }
```

FIGURE 6.8: RNS for distributed KGE models

4.1.3 Score and Loss calculation: these two calculations are the unique features of each for different KGE models. Hence, they will be explained separately in Section 6.3, Section 6.4 and Section 6.5.

6.3 Distributed TransE

Adapting the basics explained in Section 6.2 following score and loss calculations are executed for TransE to complete the Algorithm 9.

Score function: The Equation 6.1 is implemented in Figure 6.9 and Figure 6.10. The function shown in Figure 6.9 takes the *sample* as the combination of Array[Row], the entity's length for extracting corresponding entity and relation embeddings from the total embedding, indication to which norm to calculate as L. To calculate the score, embeddings of the triple are extracted as shown from lines 7 - 10 of Figure 6.9. The values of the triples (h, r, t) are principally the

```
 1  def Score(sample:Array[Row], entityListLength:Integer,
 2            embedding:Tensor[Float], L:Int, gamma:Float):
 3            (Tensor[Float]) = {
 4    val sampleDistance = Tensor(sample.size, embedding.size(2))
 5    var count = 1
 6    sample.map(row => {
 7      val subTensor  = embedding.select(1, row.get(0).toString.toInt)
 8      val predTensor = embedding.select(1, row.get(1).toString.toInt +
 9                                                  entityListLength)
10      val objTensor  = embedding.select(1, row.get(2).toString.toInt)
11      val dist = L_p_norm(subTensor, predTensor, objTensor, L)
12      sampleDistance.update(count,dist)
13      count += 1
14    })
15    (sampleDistance)
16  }
```

FIGURE 6.9: Score function of distributed TransE

```
def L_p_norm(SubTensor:Tensor[Float], predTensor:Tensor[Float],
               objTensor:Tensor[Float],  L:Int): (Tensor[Float]) = {
  L match {
    case 1 => ((SubTensor + predTensor - objTensor).abs)//L1-norm
    case _ => ((SubTensor.pow(2) + predTensor.pow(2) -
                  objTensor.pow(2)).sqrt.abs)//L2-norm
  }
}
```

FIGURE 6.10: L_p norm of distributed TransE

corresponding embedding index due to the mapping nature in the earlier stages of KGE models. Embeddings of the relations are extracted with the (index + length of the entity) due to the embedding structure discussed previously. Once the embeddings are extracted $L1$, or $L2$ norm is calculated as shown in Figure 6.10 with the basic Tensor functionalities from BigDL.

Loss function: Figure 6.11 implements the Equation 6.2 for loss calculation. As shown in Figure 6.6, once the positive score and negative score are calculated, they are sent to the *LossFunction* as showed in Figure 6.11 which basically subtracts the summation of the positive score and gamma from the negative score. Now, Equation 6.2 takes the positive values only. Therefore, *ReLU*, a built-in function from BigDL, is used over the resulting Tensor of loss. Finally, the average loss of the entire Tensor is calculated as shown in line 4 of Figure 6.11.

```
1 def LossFunction(posDist:Tensor[Float], negDist:Tensor[Float],
2                  gamma:Float, k:Int, trainSize: Long):
3                  (Float) = {
4     val loss = (ReLU().forward(posDist  + gamma - negDist)).sum
5                 /trainSize
6     println(loss)
7     (loss)
8 }
```

FIGURE 6.11: Margin Ranking Loss for distributed KGE models

6.4 Distributed DistMult

Adapting the basics explained in Section 6.2 following score and loss calculations are executed for DistMult to complete the Algorithm 9.

Score function: The implementation of Equation 6.3 is shown in Figure 6.12 for DistMult. Similar to the score function of TransE, the embeddings of each triple are extracted. Since the score function of DistMult uses the trilinear dot product of the embeddings, the built-in *map* function is used to multiply the embedding values by the row indices of the embedding Tensors as illustrated from lines 12 - 13. This multiplication is executing two Tensors at a time since BigDL's Tensor operation allows to put two Tensors for the mapping.

```
 1  def Score(sample:Array[Row], entityListLength:Integer,
 2            embedding:Tensor[Float], L:Int, gamma:Float):
 3            (Tensor[Float]) = {
 4    val sampleDistance = Tensor(sample.size, embedding.size(2))
 5    var count = 1
 6    sample.map(row => {
 7      val subTensor  = embedding.select(1, row.get(0).toString.toInt)
 8      val predTensor = embedding.select(1, row.get(1).toString.toInt
 9                                              + entityListLength)
10      val objTensor  = embedding.select(1, row.get(2).toString.toInt)
11      val cloneSubTensor = subTensor.clone
12      val dist = cloneSubTensor.map(predTensor, (a, b) => a*b)
13                               .map(objTensor, (c, d) => c*d).sum
14      sampleDistance.update(count,dist)
15      count += 1
16    })
17    (sampleDistance)
18  }
```

FIGURE 6.12: Score function of distributed DistMult

Loss function: One specialty of DistMult's loss function is, it uses the regularization of the positive sample's embeddings. The distributed approach of DistMult also uses the margin-based ranking criterion as shown in Equation 6.2. The loss is added with the multiplication of regularized embeddings (denoted as *regul*) with the learning rate as shown in Figure 6.14. The implementation of regularization is

```
 1  def Regularization(sample:Array[Row], entityListLength:Integer,
 2                     embedding:Tensor[Float]) : (Float) = {
 3    val subTensor  = Tensor(sample.size, embedding.size(2))
 4    val predTendor = Tensor(sample.size, embedding.size(2))
 5    val objTensor  = Tensor(sample.size, embedding.size(2))
 6    var count = 1
 7    sample.map( row => {
 8      subTensor.update(count, embedding.select(1, row.get(0)
 9                                                     .toString.toInt))
10      predTendor.update(count, embedding.select(1, row.get(1)
11                                                      .toString.toInt
12                                                      + entityListLength))
13      objTensor.update(count, embedding.select(1, row.get(2)
14                                                     .toString.toInt))
15      count += 1
16    })
17    (subTensor.pow(2).mean+predTendor.pow(2).mean+objTensor.pow(2).mean)
18  }
```

FIGURE 6.13: Regularization for distributed KGE models

```
return (LossFunction(positiveDistance, negativeDistance, gamma, k,
        trainSize) + learningRate * regul)
```

FIGURE 6.14: Loss calculation with regularization for distributed KGE models

illustrated in Figure 6.13 where extraction of embeddings of the entity and relation of each triple is executed, and the summation of the mean squared values of the embeddings is calculated as shown in line 17.

6.5 Distributed ComplEx

Adapting the basics explained in Section 6.2 following score and loss calculations are executed for ComplEx to complete the Algorithm 9.

Score function: Embeddings of the ComplEx model are divided into two parts: real and imaginary. To implement the score function of ComplEx as shown in Equation 6.4, the embeddings are split into two sections as exhibited in Figure 6.15. One for the imaginary part and the other for the real part. For each triple in the sample, embeddings are extracted as shown in lines 7 - 9 of Figure 6.15. Corresponding embeddings are split into real and imaginary embedding, as shown in lines 13, 15, 17. The rest of the calculation is very trivial. The real score and imaginary score are calculated to get the model's score, as shown from lines 18 - 21.

```
1   def Score(sample:Array[Row], entityListLength:Integer,
2             embedding:Tensor[Float], L:Int, gamma:Float, k:Int):
3             (Tensor[Float]) = {
4     val sampleDistance = Tensor(sample.size, embedding.size(2))
5     var count = 1
6     sample.map(row => {
7       val subTensor  = embedding.select(1, row.get(0).toString.toInt)
8       val predTensor = embedding.select(1, row.get(1).toString.toInt
9                                                + entityListLength)
10      val objTensor  = embedding.select(1, row.get(2).toString.toInt)
11      val splitValue = k/2
12      var splitTensor = subTensor.split(splitValue,1)
13      val (realSub, imgSub) = (splitTensor(0), splitTensor(1))
14      splitTensor = predTensor.split(splitValue,1)
15      val (realPred, imgPred) = (splitTensor(0), splitTensor(1))
16      splitTensor = objTensor.split(splitValue,1)
17      val (realObj, imgObj) = (splitTensor(0), splitTensor(1))
18      val realScore = (realSub * realPred - imgSub * imgPred).sum
19      val imgScore = (realSub * imgPred  + imgSub * realPred).sum
20      val dis = ((realObj * realScore) + (imgObj * realScore)).sum
21      sampleDistance.update(count,dist)
22      count += 1
23    })
24    (sampleDistance)
25  }
```

FIGURE 6.15: Score function of distributed ComplEx

Loss function: Loss calculation of ComplEx is the same as DistMult explained in Section 6.4.

6.6 Overall Architecture Of The Approach

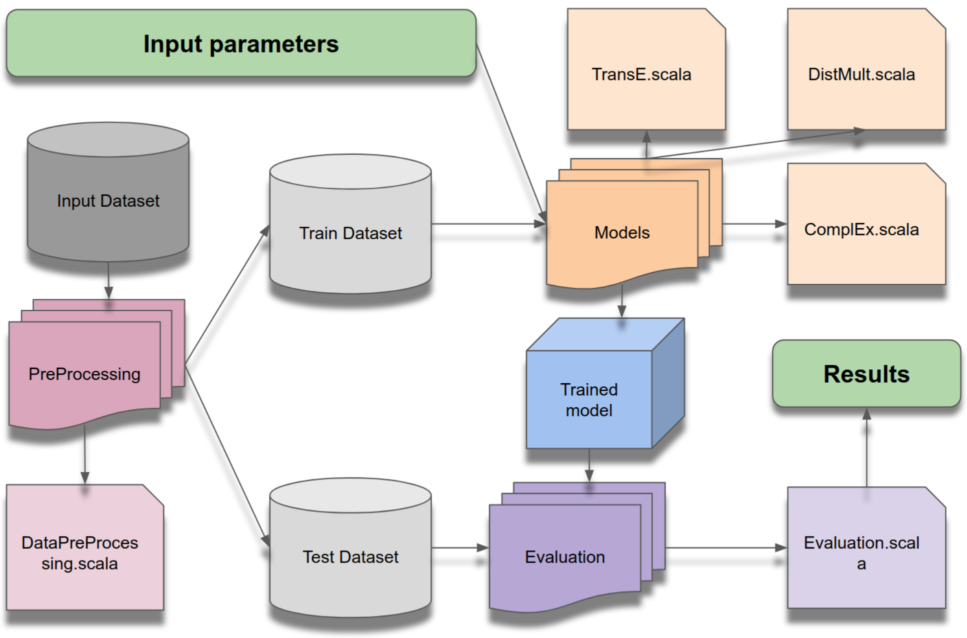


FIGURE 6.16: Distributed KGE framework

Distribution of the data is achieved by implementing the algorithm with Spark and BigDL. BigDL is also built on Spark. Optimization works in two ways with BigDL. Over direct Tensor manipulation, which is similar to general implementations of the KGE models. The second way of using optimizers with BigDL is by creating a neural network of the task. In this thesis, KGE models are optimized over the first process of updating the gradients. Creating a neural network of the KGE models is a future work that will be discussed in Chapter 8.
The distributed KGE framework is illustrated in Figure 6.16, consists of three packages: *PreProcessing*, *Models* and *Evaluation*. *PreProcessing* package takes the original dataset and generated the files explained in Chapter 4. With proper parameters and the input dataset, KGE models from the *Models* package is run to prepare the trained model. This trained model is used by *Evaluation.scala* to acquire the results, which will be illustrated in Chapter 7.

Chapter 7

Evaluation

This chapter evaluates the different Negative Sampling Algorithms proposed in Chapter 5 over different KGE models. Furthermore, implementing the KGE models in a distributed way with Spark is discussed in this chapter. The chapter starts with the description of the datasets in Section 7.1, metrics used to evaluate the approaches in Section 7.2, experimental setup in Section 7.3 and finally the results and findings in Section 7.4.

7.1 Dataset

Three datasets are used for the evaluation purpose. They are Kinship, UMLS, and Nations. The datasets are divided into three parts: train set, test set, and validation set with the ratio of 80:10:10. Table 7.1 shows the number of entities and relations and the triples in the train, test, and valid dataset. All the datasets contain only positive triples.

Dataset	**Number of**		**Number of triples in**			
	Entities (e)	**Relations (r)**	**Train set**	**Test set**	**Valid set**	**Total set**
Kinship	104	25	8544	1068	1074	10686
UMLS	135	46	5216	652	661	6529
Nations	14	55	1592	199	201	1992

Table 7.1: Statistics of the datasets

7.1.1 Kinship

Kinship dataset contains 'Aboriginal kinship of Alyawarra Tribe in Central Australia' [23]. The dataset is formed as triple (h, r, t) where relation (r) indicates the kinship among the people who act as entities (h, t). Table 7.1 indicates 25 kinship terms and 104 persons, making the total size of the dataset 10686.

7.1.2 UMLS

UMLS dataset consists of data from a Biomedical Domain called 'Unified Medical Language System' (UMLS) [25]. In this dataset, the entities (h, t) are different biomedical concepts like *hormone*, $cell\_functions$ etc. Relations (r) are the connection among them, for example: *complicates*, $affects$ etc. As mentioned in the Table 7.1, there are 135 entities and 46 relations and the size of the total dataset is 6529.

7.1.3 Nations

Nations dataset represents the paired relationships among 14 nations, for example economic, diplomatic, military etc [24]. All the nations are represented as entities (h, t) for example: Uk, $Netherlands$ etc. and the interactions among these nations are represented as relations (r) of the dataset. Sample relations are: *independence*, $conferences$ etc. There are 1992 numbers of triples (h, r, t) in the dataset from 14 entities and 55 relations as shown in the Table 7.1.

7.2 Evaluation Metrics

The evaluations of the models are based on the link prediction problem [57]. Standard evaluation metrics for these link predictions are used to evaluate the models. They are: Mean Rank (MR), Mean Reciprocal Rank (MRR) and $Hit@N$ (N = 1,3,5,10). Before testing with the evaluation metric following steps are executed:

- For each positive triple (h, r, t) in the test set, a set of a sample (S) is created. The head or the tail entity is replaced to create a set of corrupted triples (h', r, t') from the entire set where h' is not equaled to h and t' is not equaled to t.
- Triples in the sample set are removed, which are true for the training dataset. This setting is referred to as the filtered setting. Otherwise, the setting is considered a raw setting.
- Thereafter, triples from the sample set created in the previous steps are ranked against the trained model by calculating the score. The rank of true triple (h, r, t) is the position in the scored sample set (S), sorted in ascending order. Rank of the triple is denoted by $rank(h, r, t)$.
- Once the rank of the S (raw or filtered) is created, the following evaluation metrics are computed with the help of $|S|$ which denotes the size of the S.

7.2.1 Mean Rank (MR)

MR is the average rank of all the true test triples and calculated as shown in Equation 7.1.

$$MR = \frac{1}{|S|} \sum_{(h,r,t) \in S} rank(h,r,t) \tag{7.1}$$

7.2.2 Mean Reciprocal Rank (MRR)

MRR is the average inverse rank of the true test triples are calculated as shown in Equation 7.2.

$$MRR = \sum_{(h,r,t)} \frac{1}{rank(h,r,t)} \tag{7.2}$$

7.2.3 Hit@N

$Hit@N$ is the average number of times the rank of the true test triples are less than the value of N as shown in Equation 7.3 where the rank which is only added if it is less than or equals to N.

$$Hit@N = \frac{1}{|S|} \sum_{(h,r,t)} if(rank(h,r,t) \leq N) \tag{7.3}$$

A well-trained model indicates a higher MRR and $Hit@N$ and a lower MR.

7.3 Experiment Parameters

The following two sections contain all the experiment's input parameters with their brief description and possible ranges of the values used.

7.3.1 Experimental Setup For Indexed Negative Sampling Algorithms

Table 7.2 shows which values are used for which parameter for the KGE framework shown in Figure 5.4. Python 3.8.5, Pandas 1.1.5, and PyTorch 1.7.1 versions are used for this evaluation.

Parameter Name	**Description**	**Value Range**
name	Name of the KGE model	[transE, distmult, complEx]
data_ dir	Directory of the datasets	Path of the data folder
dim	Dimension of the embeddings	[20, 50, 100, 200]
batch_ size	Batch size of the samples	[500 ∼2750]
lr	Learning rate	[0.001, 0.01, 0.1]
max_ epoch	Number of max epochs	[100 ∼1000]
gamma	Gamma	[1, 2, 10]
negsample_ num	Number of negative triple per positive triple	[1, 2, 3, 5]
regul	Regularization	[True, False]
neg_ sampling	Name of the Negative Sampling Algorithms	[rand, indexed_rand, dns, indexed_dns, adns, indexed_adns]
optim	Optimizer	AdaGrad

Table 7.2: Name, description and value range of the parameters of train.py from KGE framework

7.3.2 Experimental Setup For Distributed KGE Models

Table 7.3 shows which values are used for which parameter for the distributed KGE framework proposed in this thesis. Spark 2.4.4, Scala 2.11.11, and BigDL 0.10.0 versions are used for this evaluation.

Parameter Name	**Description**	**Value Range**
resourceDir	Directory of the datasets	Path of the data folder
K	Dimension of the embeddings	[20, 50, 100, 200]
gamma	Gamma	[1, 2, 10]
learningRate	Learning rate	[0.001, 0.01, 0.1]
L	L_1 or L_2 norm	[1, 2]
nEpoch	Number of maximum epochs	[100 ∼1000]
optim	Optimizers	[SGD, Adagrad, Adam]

Table 7.3: Name, description and value range of the parameters of distributed KGE framework

7.4 Results And Findings

The following sections of this chapter explain the results and findings of the work that went behind this thesis. The evaluations of this thesis are illustrated in two sections. The first one focuses on the performances of the proposed Negative Sampling Algorithms. The other one is for the implementation of different KGE models in a distributed way. The parameters for each of these approaches are exhibited in Section 7.3.

7.4.1 Results And Findings of Indexed Negative Sampling Algorithms

This section exclusively focuses on the training time and performances (mostly for the filtered setting) of different variants of Negative Sampling Algorithms (NSA) for each dataset mentioned in Section 7.1.

KGE model	**NSA**	**Training time (s)**	**MR**		**MRR**		**hit@1 (%)**		**hit@3 (%)**		**hit@5 (%)**		**hit@10 (%)**	
			raw	**filter**	**raw**	**filter**	**raw**	**filter**	**raw**	**filter**	**raw**	**filter**	**raw**	**filter**
ComplEx	RNS	71	15	9	0.1718	0.4368	3.26	27.7	14.06	50.42	25.84	63.08	51.91	78.54
	Indexed RNS	387.16	11	3	0.1924	0.6894	5.87	55.45	18.25	78.45	30.17	86.55	56.75	94.55
	DNS	7577.51	8	2	0.2588	0.8909	8.24	76.16	27.19	91.39	45.34	93.90	76.68	96.79
	Indexed DNS	3664.23	11	5	0.2273	0.8324	7.96	76.82	22.86	86.71	35.80	84.26	62.24	94.15
	ADNS	992.18	9	2	0.2205	0.84	14.84	85.01	18.2	91.11	32.26	94.23	64.39	96.88
	Indexed ADNS	800.72	13	7	0.2008	0.8102	7.91	73.43	22.25	88.84	33.8	88.76	57.45	92.22
DistMult	RNS	41.16	16	9	0.1529	0.4007	3.54	20.58	12.24	42.04	21.23	55.91	42.32	74.02
	Indexed RNS	367.39	13	6	0.1786	0.4404	5.59	28.17	16.62	49.53	25.56	57.77	47.21	76.54
	DNS	8639.2	12	6	0.1921	0.5261	6.28	38.87	18.02	57.87	28.82	67.32	50.33	83.75
	Indexed DNS	3831.29	14	8	0.1989	0.3688	7.03	24.3	18.81	45.34	29.42	62.29	51.49	83.10
	ADNS	1039.47	13	6	0.1812	0.4827	6.19	33.89	16.67	53.45	26.35	63.92	45.25	82.82
	Indexed ADNS	902.95	15	7	0.1621	0.3806	3.91	22.02	14.2	43.58	23.74	55.26	43.76	74.26
TransE	RNS	39.42	15	8	0.1249	0.4031	6.21	58.7	11.86	50.01	24.19	55.07	58.05	70.95
	Indexed RNS	358.54	9	3	0.1508	0.5774	7.65	61.29	13.07	64.05	26.1	66.98	61.22	81.45
	DNS	2871.61	14	7	0.1385	0.427	6.21	59.8	11.3	51.63	23.35	45.40	51.03	81.19
	Indexed DNS	2030.37	12	6	0.1402	0.4433	8.044	63.95	11.72	51.82	23.77	54.1	45.88	75.98
	ADNS	956.67	13	7	0.1449	0.496	7.99	64.89	12.65	62.93	24.47	55.19	51.02	76.15
	Indexed ADNS	820.48	8	2	0.1484	0.5382	8.39	60.16	12.51	62.79	24.98	56.1	51.103	81.27

Table 7.4: Evaluations of KGE models with Kinship dataset

Kinship dataset: The performances for Negative Sampling Algorithms for Kinship dataset are elaborated in Table 7.4 . This table is generated with dimension, k

$= 50$, *batch size* $= 2750$, *learning rate* $= 0.1$, *gamma* $= 10$ and *number of negative samples* $= 3$ for 1000 *epochs*.

To address the first research question mentioned in Chapter 1, it is important to know the significance of creating a meaningful negative sample for the positive sample. Figure 7.1 shows that major times, all the Negative Sampling Algorithms work better than RNS with respect to different KGE models.

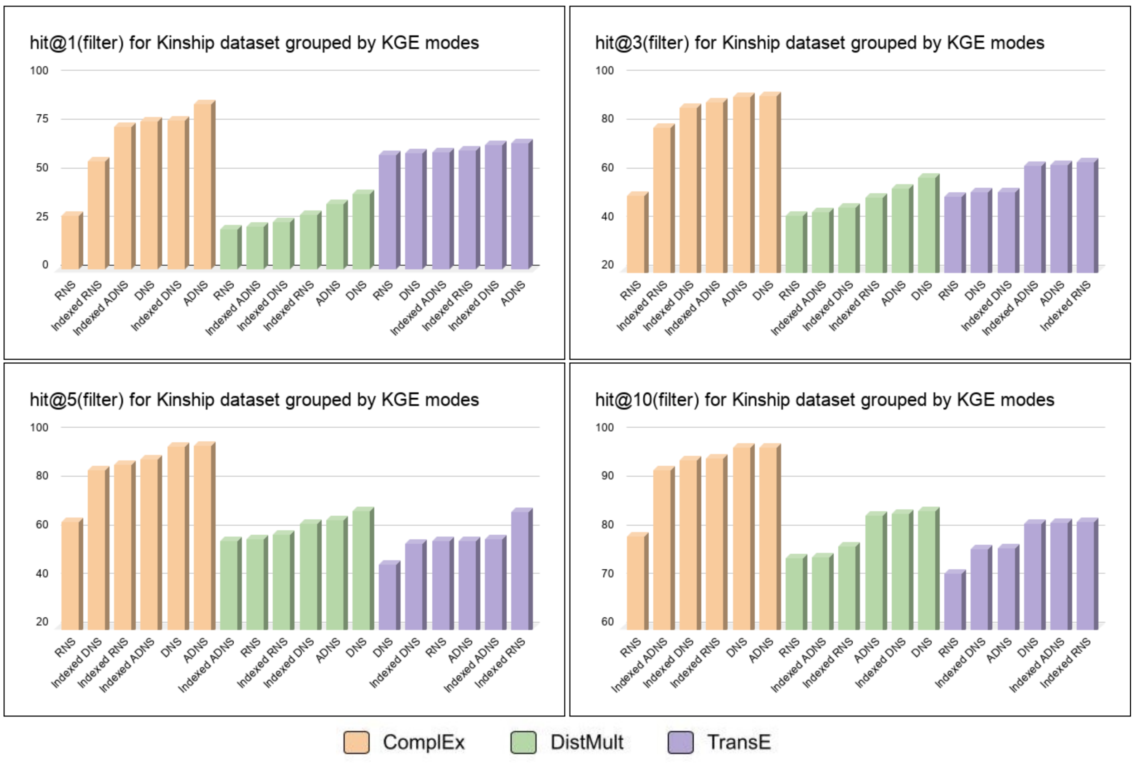


FIGURE 7.1: Performances of Negative Sampling Algorithm for each KGE models for Kinship dataset

DNS and ADNS algorithms aim to create more meaningful negative samplings [9][11], although the training time of these algorithms is much higher than RNS. Figure 7.2 shows the training time comparison between DNS and Indexed DNS among all three KGE models for the Kinship dataset. The processing time for Indexed DNS with TransE algorithm decreases by 17%. For DistMult and ComplEx, the time decreases by 38.6% and 34.8% correspondingly for *Indexed DNS*.

From Figure 7.1 and Table 7.4, it can be perceived that for TransE model, *Indexed DNS* performs better than DNS with respect to *Hit*@1 (increased from 59.8% to 63.95%), *Hit*@3 (increased from 51.63% to 51.82%) and *Hit*@5 (increased from 45.40% to 54.1%). For the DistMult model, *Indexed DNS* performs closely to DNS with respect to *Hit*@10 since for *Indexed DNS* it is 83.10% and for DNS it is 83.75%.

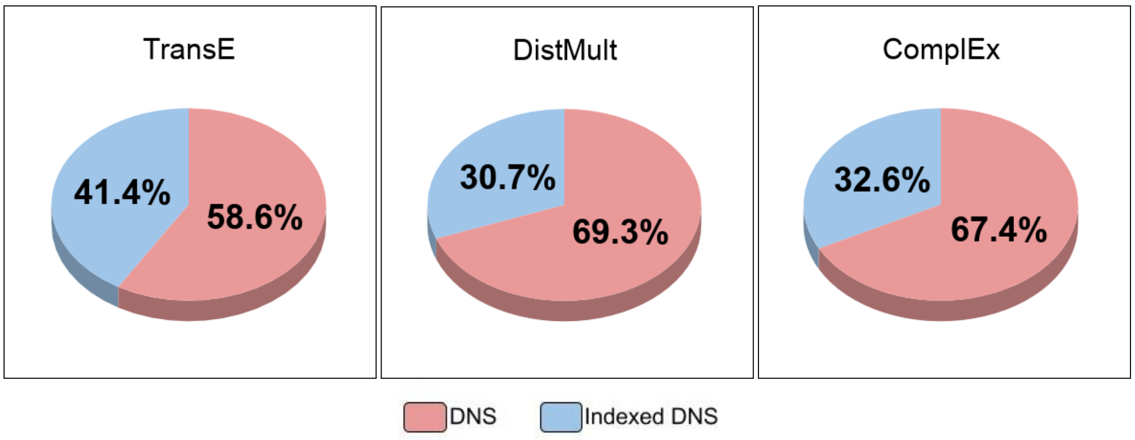


FIGURE 7.2: Training time comparison between DNS and *Indexed DNS* for Kinship

Lastly, for the ComplEx model, *Indexed DNS* is performing better than DNS at *Hit*@1 (increased from 76.16% to 76.82%).

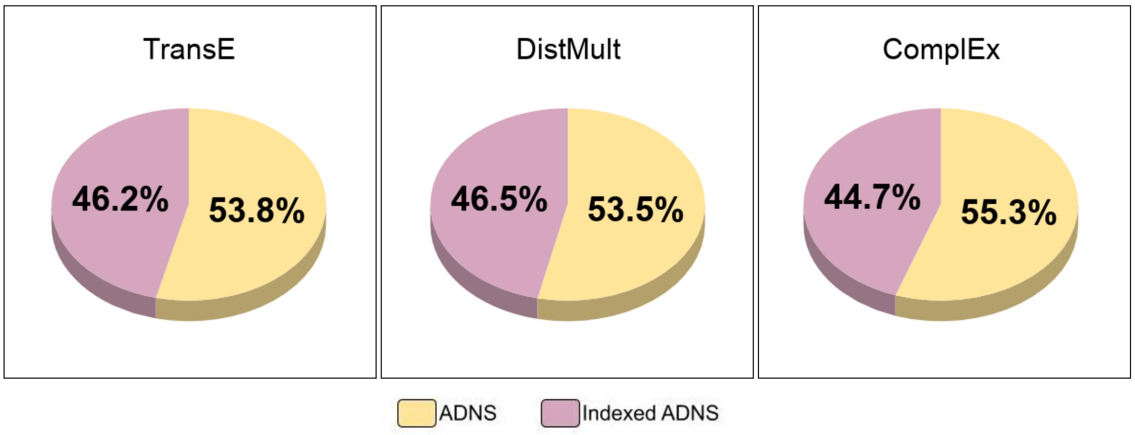


FIGURE 7.3: Training time comparison between ADNS and *Indexed ADNS* for Kinship

Training time for *Indexed ADNS* with respect to ADNS is decreased by 7.6% for TransE, 7% for DistMult and 10.6% for ComplEx, shown in Figure 7.3. Figure 7.1 and Table 7.4 shows that, for TransE model, *Hit*@5 increases to 56.1% with *Indexed ADNS* from 55.19% with ADNS. Both *Indexed ADNS* and ADNS perform closely for TransE by having *Hit*@3 as 62.79% and as 62.93%, respectively. *MR* of ADNS for TransE is 7, higher than *MR* of *Indexed ADNS*, which is 2. For the ComplEx model, *Indexed ADNS* (88.84%) and ADNS (91.11%) perform closely with respect to *Hit*@3. Also, *MRR* of *Indexed ADNS* is 0.8102, close to 0.84, the *MRR* of ADNS for ComplEx.

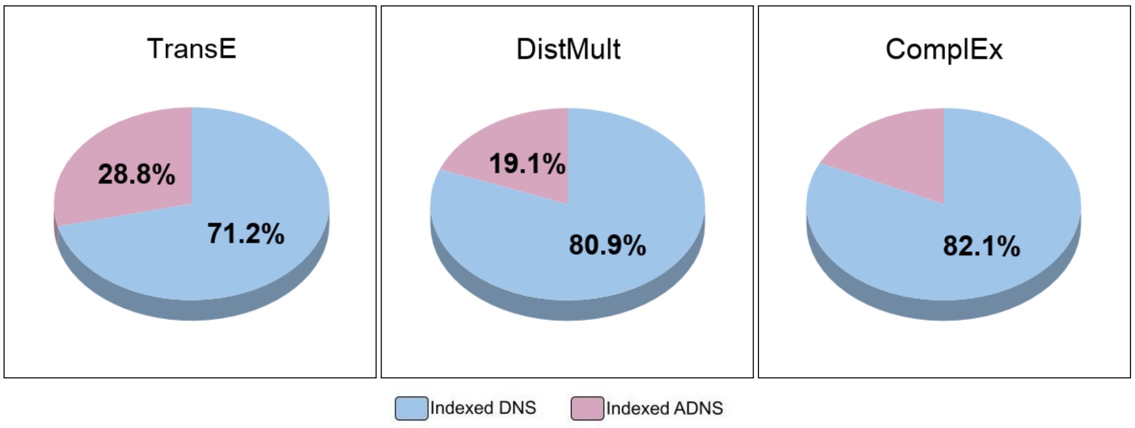


FIGURE 7.4: Training time comparison between *Indexed DNS* and *Indexed ADNS* for Kinship

To compare the performances of *Indexed DNS* and *Indexed ADNS*, Table 7.4 shows that for ComplEx, *Hit*@3 for *Indexed DNS* is 86.71% whereas, for *Indexed ADNS*, it is 88.84%. Figure 7.4 shows that the training time for *Indexed ADNS* is around 64% which is less than *Indexed DNS*. Similarly, for TransE, *Hit*@5 increases by 3.5% for *Indexed ADNS* with a 42% reduced training time. For TransE *MR* of *Indexed ADNS* is 2, and *MR* of *Indexed DNS* is 6, which is higher in value. This pattern can be seen in Table 7.4 for DistMult as well. Similarly, *MRR* shows the potentiality of *Indexed ADNS* over *Indexed DNS*. For DistMult, *MRR* of *Indexed DNS* is 0.3688. This is lower than *MRR* of *Indexed ADNS*, which is 0.3806.

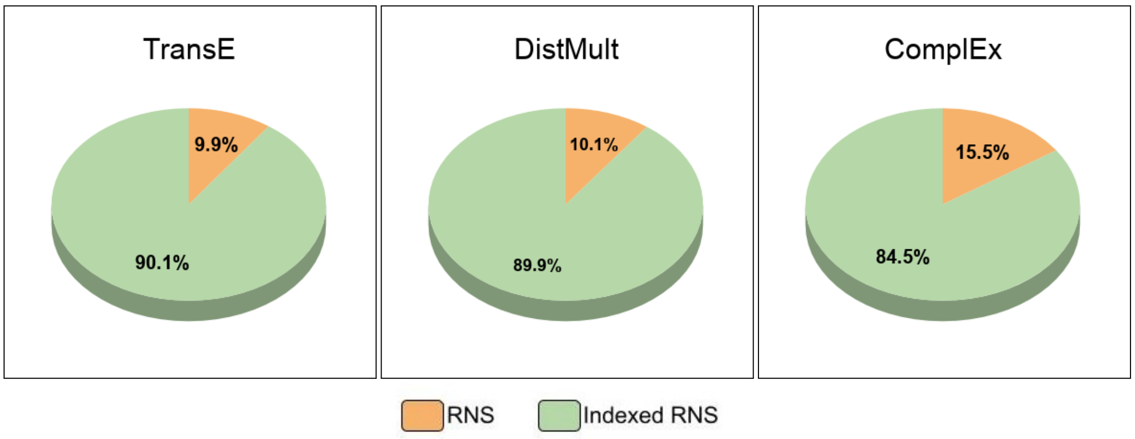


FIGURE 7.5: Training time comparison between RNS and *Indexed RNS* for Kinship

Training time for *Indexed RNS* is more than RNS as shown in Figure 7.5. However, Table 7.5 shows most of the time, *Indexed RNS* creates more meaningful negative samples to elevate the KGE model's performances than RNS does. Not

Hit@1		**Hit@3**		**Hit@5**		**Hit@10**	
KGE_NSA	%	**KGE_NSA**	%	**KGE_NSA**	%	**KGE_NSA**	%
DistMult_RNS	20.58	DistMult_RNS	42.04	TransE_DNS	45.396	TransE_RNS	70.954
DistMult_*Indexed ADNS*	22.02	DistMult_*Indexed ADNS*	43.58	TransE_*Indexed DNS*	54.1	DistMult_RNS	74.02
DistMult_*Indexed DNS*	24.3	DistMult_*Indexed DNS*	45.34	TransE_RNS	55.07	DistMult_*Indexed ADNS*	74.26
ComplEx_RNS	27.7	**DistMult**_*Indexed RNS*	**49.53**	TransE_ADNS	55.19	TransE_*Indexed DNS*	75.982
DistMult_*Indexed RNS*	**28.17**	TransE_RNS	50.01	DistMult_*Indexed ADNS*	55.26	TransE_ADNS	76.145
DistMult_ADNS	33.89	ComplEx_RNS	50.42	DistMult_RNS	55.91	**DistMult**_*Indexed RNS*	**76.54**
DistMult_DNS	38.87	TransE_DNS	51.63	TransE_*Indexed ADNS*	56.1	ComplEx_RNS	78.54
ComplEx_*Indexed RNS*	**55.45**	TransE_*Indexed DNS*	51.82	**DistMult**_*Indexed RNS*	**57.77**	TransE_DNS	81.192
TransE_RNS	58.7	DistMult_ADNS	53.45	DistMult_*Indexed DNS*	62.29	TransE_*Indexed ADNS*	81.266
TransE_DNS	59.8	DistMult_DNS	57.87	ComplEx_RNS	63.08	**TransE**_*Indexed RNS*	**81.448**
TransE_*Indexed ADNS*	60.16	TransE_*Indexed ADNS*	62.79	DistMult_ADNS	63.92	DistMult_ADNS	82.82
TransE_*Indexed RNS*	**61.29**	TransE_ADNS	62.93	**TransE**_*Indexed RNS*	**66.98**	DistMult_*Indexed DNS*	83.1
TransE_*Indexed DNS*	63.95	**TransE**_*Indexed RNS*	**64.05**	DistMult_DNS	67.32	DistMult_DNS	83.75
TransE_ADNS	64.89	**ComplEx**_*Indexed RNS*	**78.45**	ComplEx_*Indexed DNS*	84.26	ComplEx_*Indexed ADNS*	92.22
ComplEx_*Indexed ADNS*	73.43	ComplEx_*Indexed DNS*	86.71	**ComplEx**_*Indexed RNS*	**86.55**	ComplEx_*Indexed DNS*	94.15
ComplEx_DNS	76.16	ComplEx_*Indexed ADNS*	88.84	ComplEx_*Indexed ADNS*	88.76	**ComplEx**_*Indexed RNS*	**94.55**
ComplEx_*Indexed DNS*	76.82	ComplEx_ADNS	91.11	ComplEx_DNS	93.9	ComplEx_DNS	96.79
ComplEx_ADNS	85.01	ComplEx_DNS	91.39	ComplEx_ADNS	94.23	ComplEx_ADNS	96.88

Table 7.5: Comparison of performances among all the Negative Sampling Algorithms (NSA) over different KGE models for Kinship

only that, sometimes *Indexed RNS* outperforms other Negative Sampling Algorithms as well. Table 7.5 shows *Hit*@10 with *Indexed RNS* for ComplEx model works better than DNS and ADNS with TransE and DistMult models for Kinship dataset. *Indexed RNS* sometimes is one of the top performers regardless of the model.

UMLS dataset: Table 7.6 is generated for UMLS datasets for 1000 *epochs* with dimension, $k = 50$, *batch size* $= 2750$, *learning rate* $= 0.1$, *gamma* $= 10$ and *number of negative samples* $= 3$ as the parameters that illustrate the performances among the Negative Sampling Algorithms (NSA) and KGE models.

KGE model	**NSA**	**Training time (s)**	**MR**		**MRR**		**hit@1 (%)**		**hit@3 (%)**		**hit@5 (%)**		**hit@10 (%)**	
			raw	**filter**	**raw**	**filter**	**raw**	**filter**	**raw**	**filter**	**raw**	**filter**	**raw**	**filter**
ComplEx	RNS	46.63	20	4	0.1088	0.6438	1.13	48.87	6.81	74.96	13.46	84.87	33.43	93.12
	Indexed RNS	274	14	4	0.2037	0.7641	5.9	64.52	20.27	85.7	34.19	91.15	57.11	95.01
	DNS	3452.89	13	2	0.2298	0.9452	8.55	91.75	23.75	97.05	36.38	97.66	58.4	98.56
	Indexed DNS	1568.61	13	2	0.2072	0.7959	5.9	68	21.03	89.79	33.81	94.25	58.4	96.52
	ADNS	713.26	16	2	0.1683	0.861	4.92	79.27	15.28	91.6	23.75	94.55	45.46	97.81
	Indexed ADNS	653	14	4	0.2045	0.6871	6.66	54.31	20.65	80.11	32	87.97	55.3	94.55
DistMult	RNS	30.18	23	7	0.0977	0.4586	0.91	28.52	6.28	55.98	11.04	66.11	29.05	79.35
	Indexed RNS	253.36	18	7	0.1867	0.606	5.98	46.44	17.47	69.06	27	78.37	51.29	86.99
	DNS	4235.31	17	5	0.1914	0.5749	6.96	56.35	18	66.26	28.74	74.81	49.7	87.29
	Indexed DNS	2872.31	16	5	0.2014	0.6647	6.28	43.12	20.27	71.33	30.26	78.82	49.45	86.38
	ADNS	731.2	21	6	0.1239	0.6044	12.65	47.58	8.47	68.31	15.58	74.81	33.89	84.04
	Indexed ADNS	690.96	16	5	0.1963	0.632	6.73	49.92	18.84	71.26	29.43	78.72	52.27	88.65
TransE	RNS	22.64	20	9	0.1388	0.2999	1.51	6.35	13.92	44.4	22.39	55.6	39.94	74.36
	Indexed RNS	247.78	17	7	0.1661	0.3976	3.25	13.24	16.94	59.38	26.55	71.18	45.61	83.13
	DNS	2633.33	20	10	0.1547	0.345	2.87	10.06	15.66	51.13	25.72	62.63	44.25	75.95
	Indexed DNS	1694.97	19	9	0.1626	0.3839	3.56	16.11	16.26	53.56	26.4	62.93	45.54	77.69
	ADNS	620.36	17	7	0.1644	0.3923	3.1	12.48	16.26	58.55	26.55	70.65	48.49	84.11
	Indexed ADNS	559.73	16	6	0.1743	0.4424	3.78	17.47	17.1	65.51	28.37	77.84	49.55	88.12

TABLE 7.6: Evaluations of KGE models with UMLS dataset

From Table 7.6 and Figure 7.6, it can be perceived that for each KGE model, all other Negative Sampling Algorithms appeared to be performing better than RNS. For Instance, *Hit*@3 of ComplEx model with RNS is 74.96%, which increases to 80.11% and 89.79% with *Indexed ADNS* and *Indexed DNS*, respectively. *Indexed RNS* sometimes performs better than RNS. *Hit*@3 for ComplEx increases to 85.7% with *Indexed RNS*. Raw *MR* of RNS is as high as 20 as shown in Table 7.6.

For *Hit*@5, all the KGE models respond closely between DNS and *Indexed DNS*. For ComplEx, *Hit*@5 for DNS is 97.66%, and for *Indexed DNS*, it is 94.25%. However, the difference is only around 3%. *Hit*@5 for TransE with DNS is 62.63%, and with *Indexed DNS*, it is 62.93%, and the difference is less than 0.5%. Sometimes, *Indexed DNS* outperforms DNS. For instance, *Hit*@5 for DistMult, with *Indexed DNS*, increases to 78.82% from 74.81% which is the model's *Hit*@5 with DNS. For *Hit*@3 and *Hit*@10 of TransE, *Indexed DNS* achieves more accuracy than DNS as found in Table 7.6 and Figure 7.6. *MR* for *Indexed DNS* is 9, which

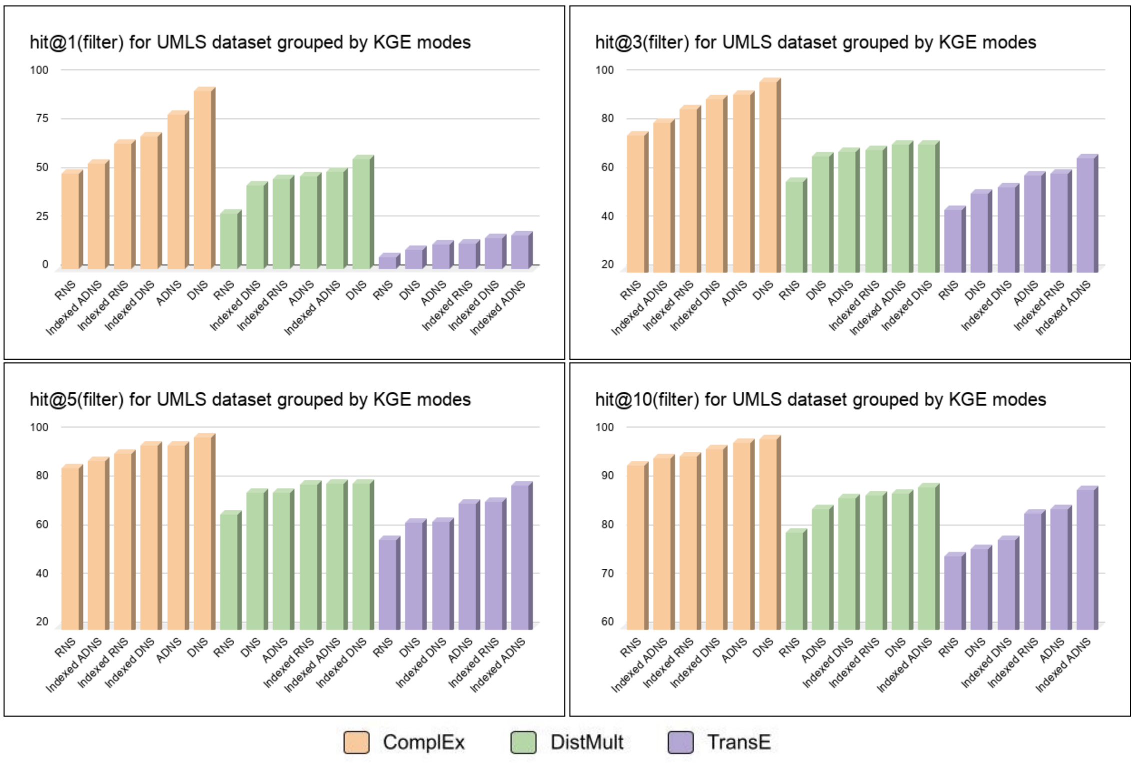


FIGURE 7.6: Performances of Negative Sampling Algorithm for each KGE models for UMLS dataset

is lower than 10, the MR of DNS for TransE. MRR of TransE also indicates that $Indexed\ DNS$ can be more effective than DNS as shown in Table 7.6. For example, MRR for $Indexed\ DNS$, which is 0.3839, is greater than MRR for DNS, which is 0.345.

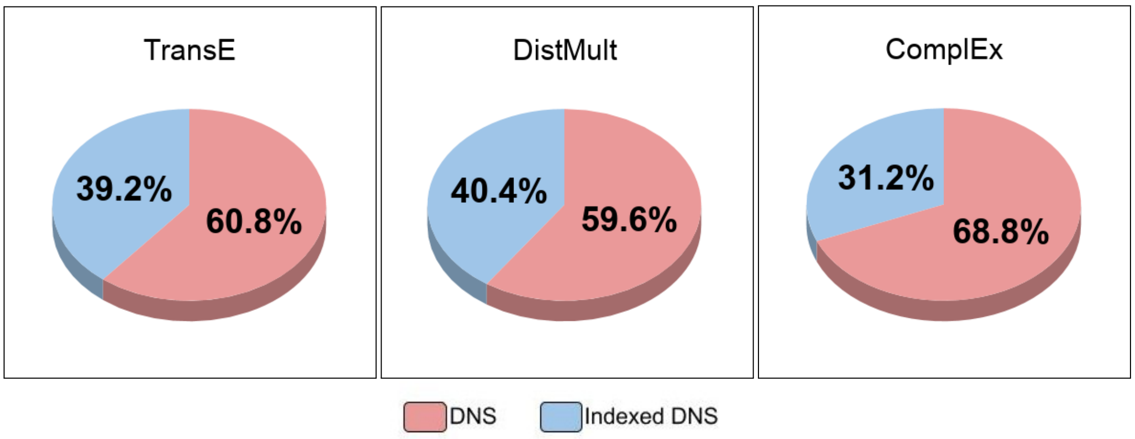


FIGURE 7.7: Training time comparison between DNS and $Indexed\ DNS$ for UMLS

Figure 7.7 shows the training time comparison between DNS and $Indexed\ DNS$ among all three KGE models for UMLS dataset. The processing time for $Indexed\ DNS$

with TransE algorithm decreases by 21.6%. For DistMult and ComplEx, the time decreases by 19.2% and 37.6% correspondingly.

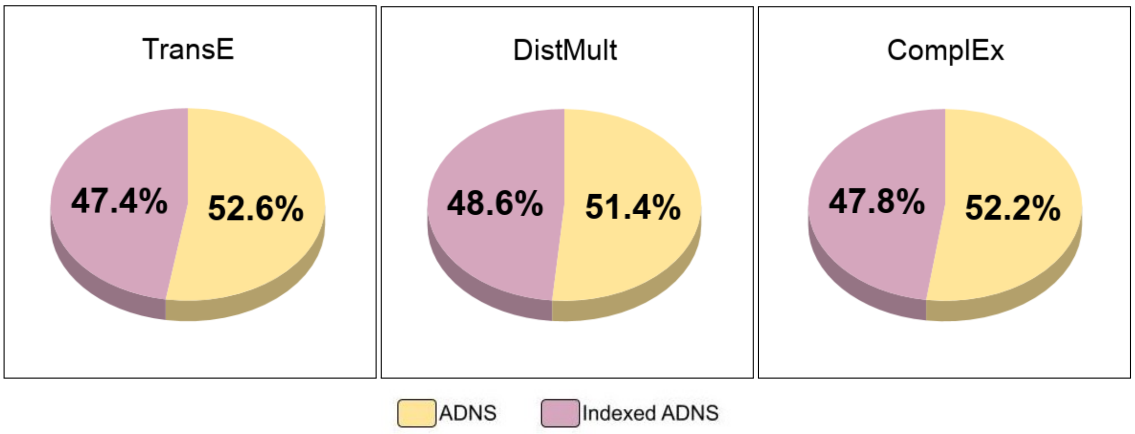


FIGURE 7.8: Training time comparison between ADNS and *Indexed ADNS* for UMLS

Indexed ADNS reduces the model's processing time as exhibited in Figure 7.8. This reduction in training time is 4.4%, 2.8%, and 5.2% correspondingly for ComplEx, DistMult, and TransE models.

The evaluation shows that for *Hit*@10 all the KGE models work well or closely between *Indexed ADNS* and ADNS. *Hit*@10 for TransE with *Indexed ADNS* is 88.12%, and with ADNS, it is 84.11%. Similarly, performance for ComplEx with *Indexed ADNS* is 94.55%, and with ADNS, it is 97.81%, and a difference is less than 5%. For DistMult model, the accuracy with *Indexed ADNS* (88.65%) is higher than ADNS (84.04%) by more than 4.5%. For TransE, *Hit*@1 for *Indexed ADNS* is 17.47%, and for ADNS, it is 12.48%, which depicts the better performance of *Indexed ADNS* by around 10%. *MRR* for *Indexed ADNS* is 0.2045, which is higher than *MRR* of ADNS valued as 0.1683. For DistMult, *MR* for *Indexed ADNS* is 5, and for ADNS, it is 6, so *MR* of *Indexed ADNS* is better.

Figure 7.6 and Table 7.6 indicates that *Indexed ADNS* works more satisfactorily than *Indexed DNS*. For example, *Hit*@10 for TransE model with *Indexed ADNS* is 88.12%, which is much higher than the *Hit*@10 with *Indexed DNS* (77.69%). Processing time for DistMult model with *Indexed ADNS* is reduced by 50% from *Indexed DNS* as illustrated in Figure 7.9 and the *Hit*@10 for the model with *Indexed ADNS* is increased to 88.65% from 86.38% (the *Hit*@10 with *Indexed DNS*). Figure 7.9 shows training time for *Indexed ADNS* is around

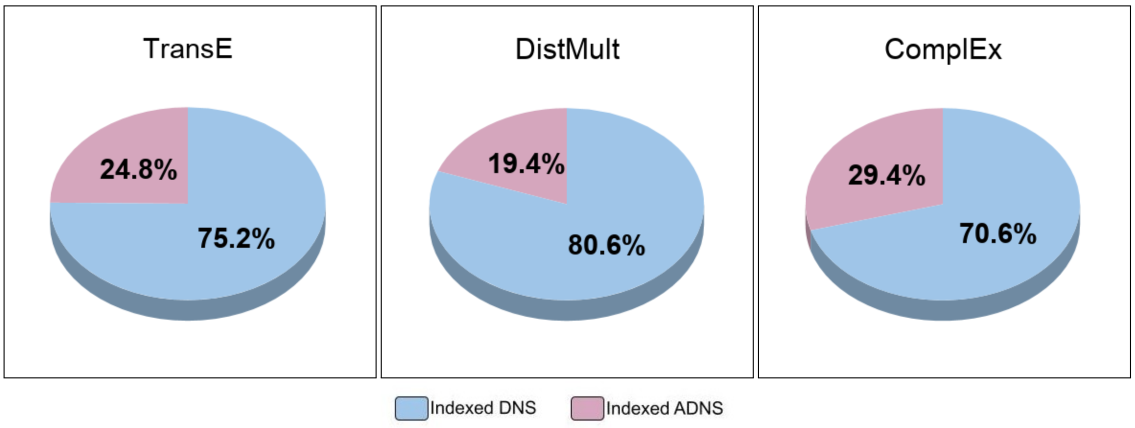


FIGURE 7.9: Training time comparison between *Indexed DNS* and *Indexed ADNS* for UMLS

60% less than *Indexed DNS* for ComplEx model. For the TransE model, *MR* of *Indexed ADNS* is 6, but *MR* of *Indexed DNS* is 9, which indicates a better performance of *Indexed ADNS*. On top of that, the processing time for TransE is reduced around 21% for *Indexed DNS* compared to *Indexed ADNS*. Moreover, *MRR* for TransE with *Indexed ADNS* is 0.1743, and *MRR* for *Indexed DNS* is 0.1626. This reveals that *Indexed ADNS* may show promising benefits than *Indexed DNS*.

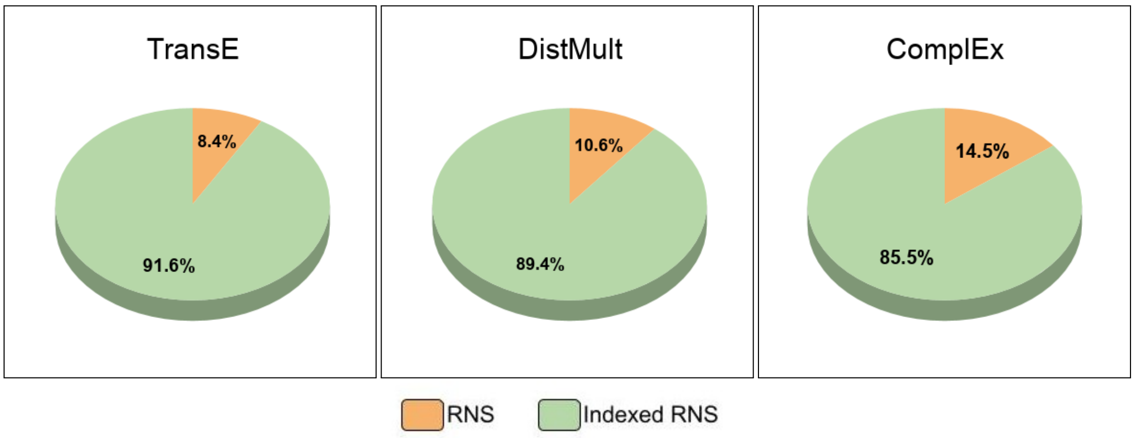


FIGURE 7.10: Training time comparison between RNS and *Indexed RNS* for UMLS

Even though ADNS and DNS are much better than RNS in theory, *Indexed RNS* sometimes performs better than other Negative Sampling Algorithms. Training time for *Indexed RNS* is more than RNS exhibited in Figure 7.10. However, it is less than all other variants of the Negative Sampling Algorithms. Table 7.7

shows that *Indexed RNS* is among top 4 performers. This shows the optimistic performances of the KGE models with an optimized time.

Hit@1		**Hit@3**		**Hit@5**		**Hit@10**	
KGE_NSA	**%**	**KGE_NSA**	**%**	**KGE_NSA**	**%**	**KGE_NSA**	**%**
TransE_RNS	6.35	TransE_RNS	44.4	TransE_RNS	55.6	TransE_RNS	74.36
TransE_DNS	10.06	TransE_DNS	51.13	TransE_DNS	62.63	TransE_DNS	75.95
TransE_ADNS	12.48	TransE_*Indexed DNS*	53.56	TransE_*Indexed DNS*	62.93	TransE_*Indexed DNS*	77.69
TransE_ *Indexed RNS*	**13.24**	DistMult_RNS	55.98	DistMult_RNS	66.11	DistMult_RNS	79.35
TransE_*Indexed DNS*	16.11	TransE_ADNS	58.55	TransE_ADNS	70.65	**TransE_** *Indexed RNS*	**83.13**
TransE_*Indexed ADNS*	17.47	**TransE_** *Indexed RNS*	**59.38**	**TransE_** *Indexed RNS*	**71.18**	DistMult_ADNS	84.04
DistMult_RNS	28.52	TransE_*Indexed ADNS*	65.51	DistMult_DNS	74.81	TransE_ADNS	84.11
DistMult_*Indexed DNS*	43.12	DistMult_DNS	66.26	DistMult_ADNS	74.81	DistMult_*Indexed DNS*	86.38
DistMult_ *Indexed RNS*	**46.44**	DistMult_ADNS	68.31	TransE_*Indexed ADNS*	77.84	**DistMult_** *Indexed RNS*	**86.99**
DistMult_ADNS	47.58	**DistMult_** *Indexed RNS*	**69.06**	**DistMult_** *Indexed RNS*	**78.37**	DistMult_DNS	87.29
ComplEx_RNS	48.87	DistMult_*Indexed ADNS*	71.26	DistMult_*Indexed ADNS*	78.72	TransE_*Indexed ADNS*	88.12
DistMult_*Indexed ADNS*	49.92	DistMult_*Indexed DNS*	71.33	**DistMult_** *Indexed DNS*	**78.82**	DistMult_*Indexed ADNS*	88.65
ComplEx_*Indexed ADNS*	54.31	ComplEx_RNS	74.96	ComplEx_RNS	84.87	ComplEx_RNS	93.12
DistMult_DNS	56.35	ComplEx_*Indexed ADNS*	80.11	ComplEx_*Indexed ADNS*	87.97	ComplEx_*Indexed ADNS*	94.55
ComplEx_ *Indexed RNS*	**64.52**	**ComplEx_** *Indexed RNS*	**85.7**	**ComplEx_** *Indexed RNS*	**91.15**	**ComplEx_** *Indexed RNS*	**95.01**
ComplEx_*Indexed DNS*	68	ComplEx_*Indexed DNS*	89.79	ComplEx_*Indexed DNS*	94.25	ComplEx_*Indexed DNS*	96.52
ComplEx_ADNS	79.27	ComplEx_ADNS	91.6	ComplEx_ADNS	94.55	ComplEx_ADNS	97.81
ComplEx_DNS	91.75	ComplEx_DNS	97.05	ComplEx_DNS	97.66	ComplEx_DNS	98.56

Table 7.7: Comparison of performances among all the Negative Sampling Algorithms (NSA) over different KGE models for UMLS

Nations dataset: The performances for Negative Sampling Algorithms (NSA) for Nations dataset are exhibited in Table 7.8 . This table is generated with dimension, $k = 50$, $batch\ size = 2750$, $learning\ rate = 0.1$, $gamma = 10$ and $number\ of\ negative\ samples = 3$ for 1000 *epochs*.

KGE model	**NSA**	**Training time (s)**	**MR**		**MRR**		**hit@1 (%)**		**hit@3 (%)**		**hit@5 (%)**		**hit@10 (%)**	
			raw	**filter**	**raw**	**filter**	**raw**	**filter**	**raw**	**filter**	**raw**	**filter**	**raw**	**filter**
ComplEx	RNS	8.29	8	2	0.179	0.6377	2.49	45.02	12.19	78.11	23.88	90.8	76.62	98.1
	Indexed RNS	27.86	8	2	0.1992	0.7031	4.23	55.22	15.67	83.83	29.1	90.8	77.11	99.5
	DNS	303.95	6	1	0.2998	0.9324	9.95	90.8	33.58	94.53	50.25	96.27	92.29	99.5
	Indexed DNS	141.56	6	2	0.2756	0.6231	7.46	42.04	33.08	78.86	48.26	90.55	82.59	99.75
	ADNS	224.24	7	2	0.2024	0.8412	2.74	75.37	16.92	91.04	34.08	96.27	83.58	99.5
	Indexed ADNS	105.53	7	2	0.2117	0.7244	5.72	56.47	16.17	86.57	31.34	94.78	80.35	99.9
DistMult	RNS	6.43	7	2	0.2123	0.6704	4.73	38.06	18.41	77.61	33.58	87.52	79.85	97.51
	Indexed RNS	26.88	7	2	0.2444	0.7142	8.21	56.72	22.14	82.59	36.82	89.8	81.34	99.25
	DNS	220.4	6	2	0.2998	0.8	10.7	70.9	32.84	85.32	50	92.79	85.82	98.01
	Indexed DNS	138.47	6	3	0.2756	0.5853	9.45	51.49	29.6	72.39	46.27	89.05	81.84	98.76
	ADNS	189.78	6	2	0.2549	0.7581	7.46	63.43	24.63	83.33	43.53	95.52	84.33	99.25
	Indexed ADNS	125.34	7	2	0.2518	0.7212	8.71	57.96	21.89	81.84	39.8	92.29	81.09	99
TransE	RNS	6.59	8	4	0.1882	0.3463	0.75	1.99	17.66	57.96	35.82	77.1	73.88	95.27
	Indexed RNS	25.04	7	3	0.2203	0.405	2.74	6.97	23.38	66.92	41.29	84.58	79.6	97.26
	DNS	195.54	7	3	0.2356	0.482	4.23	20.15	24.13	69.9	45.02	85.07	79.6	97.01
	Indexed DNS	166.15	7	4	0.2423	0.4046	5.47	13.18	26.87	58.71	42.54	78.61	75.12	97.76
	ADNS	278.32	7	3	0.2462	0.477	5.72	18.41	26.62	70.9	43.78	84.08	80.6	97.51
	Indexed ADNS	116.7	7	4	0.2094	0.3923	1.74	6.22	22.89	65.92	38.56	82.59	80.1	96.77

Table 7.8: Evaluations of KGE models with Nations dataset

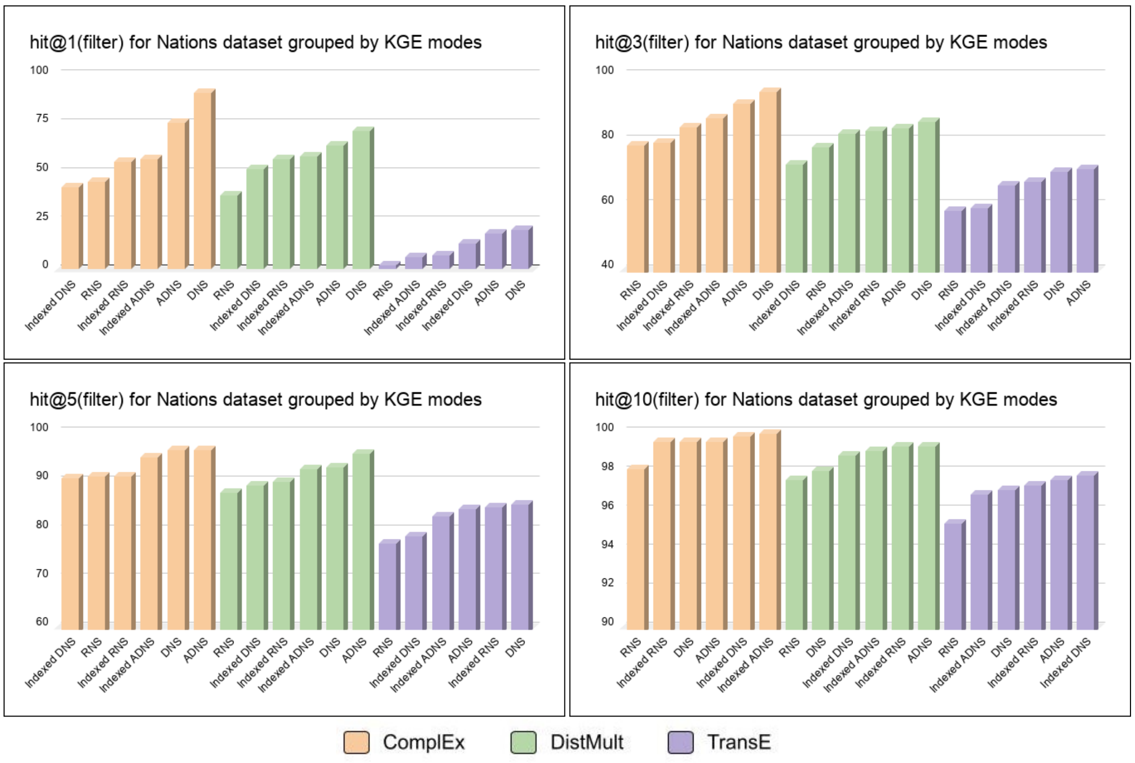


FIGURE 7.11: Performances of Negative Sampling Algorithm for each KGE models for Nations dataset

For Nations Dataset, most of the time, RNS performs less than other Negative Sampling Algorithms as illustrated in Table 7.8 and Figure 7.11. For instance, *Hit*@1 for TransE with RNS is 1.99%. However, at least 6.22% accuracy is achieved with *Indexed ADNS*. Figure 7.11 exhibits, sometimes the accuracy of *Indexed DNS* loses to RNS. For example, *Hit*@1 for ComplEx, *Hit*@3 for DistMult supports the statement. Nevertheless, other Negative Sampling Algorithms mostly work better than RNS.

Although Nations is a smaller dataset than Kinship and UMLS, supported by Section 7.1, training time does get optimized for *Indexed DNS* and *Indexed ADNS* as well. Training time for ComplEx with *Indexed DNS* is optimized by 36.4% with respect to DNS as shown in Figure 7.12. Similarly, training time for DistMult with *Indexed DNS* is reduced by 22.8%, and for TransE it is reduced by 8.2% compared to the training time of DNS.

Hit@10 for all the KGE models performs better with *Indexed DNS* than DNS. For TransE, with *Indexed DNS*, the *Hit*@10 is 97.76%, whereas, with DNS, it is 97.01%. For DistMult with *Indexed DNS*, the *Hit*@10 is 98.76%, which is higher

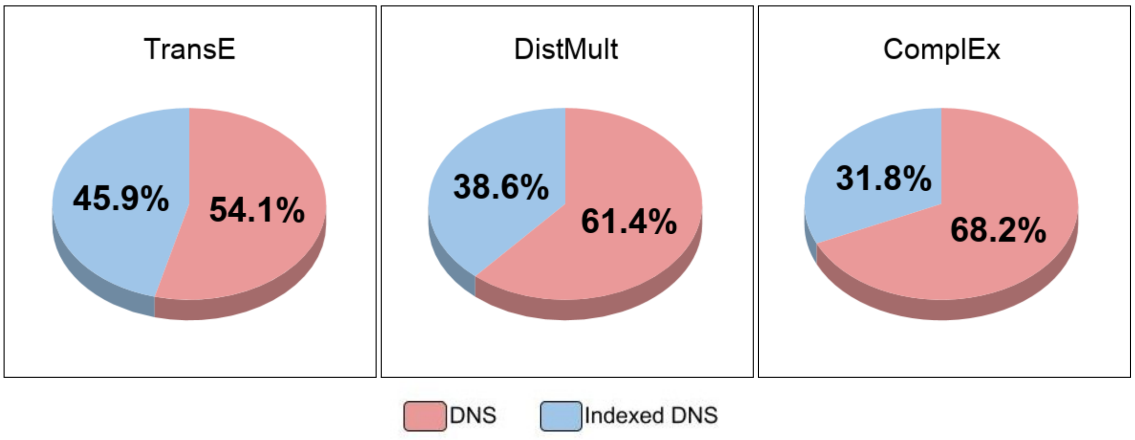


FIGURE 7.12: Training time comparison between DNS and *Indexed DNS* for Nations

than the accuracy with DNS (98.01%). For ComplEx, with *Indexed DNS* the *Hit*@10 is 99.75%, and with DNS it is less, valued at 99.5% as shown in Table 7.8.

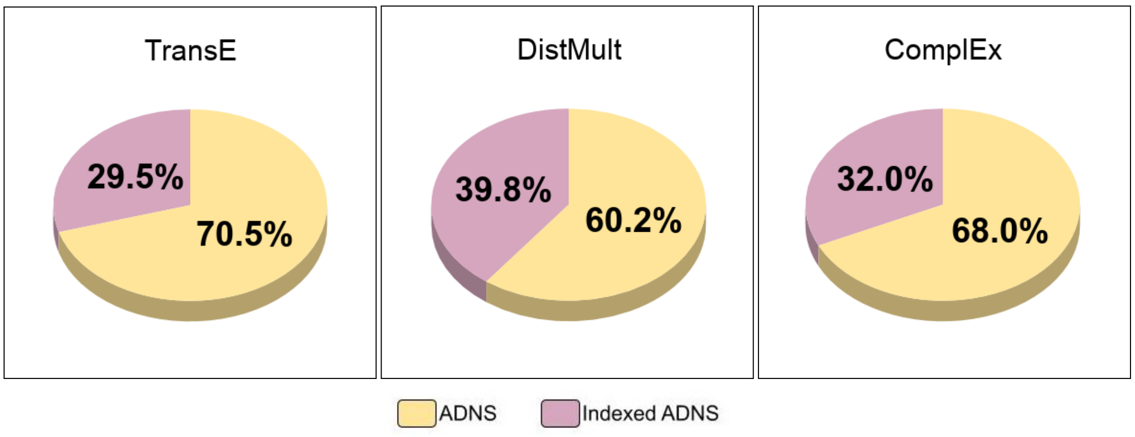


FIGURE 7.13: Training time comparison between ADNS and *Indexed ADNS* for Nations

On the other hand, *Indexed ADNS* too improves the training time with respect to ADNS. For ComplEx, the training time is reduced by 36%, for DistMult it is reduced by 20.4%, and for TransE it is reduced 41%. These improvements are illustrated in Figure 7.13.

Outcomes of *Indexed ADNS* and ADNS also indicate *Indexed ADNS* can work better as found in Figure 7.11. For ComplEx, *Hit*@10 with *Indexed ADNS* is 99.9% which is the highest performing Negative Sampling Algorithm as shown in Table 7.9. For other cases, *Indexed ADNS* and ADNS perform closely to each other. For example, as shown in Table 7.8 *Hit*@3 for DistMult with *Indexed ADNS* is 81.84% which is close to 83.33% of $Hit@3$ with ADNS. For raw MRR, $Indexed\ ADNS$ increased to 0.2117 from 0.2024, raw MRR of ADNS.

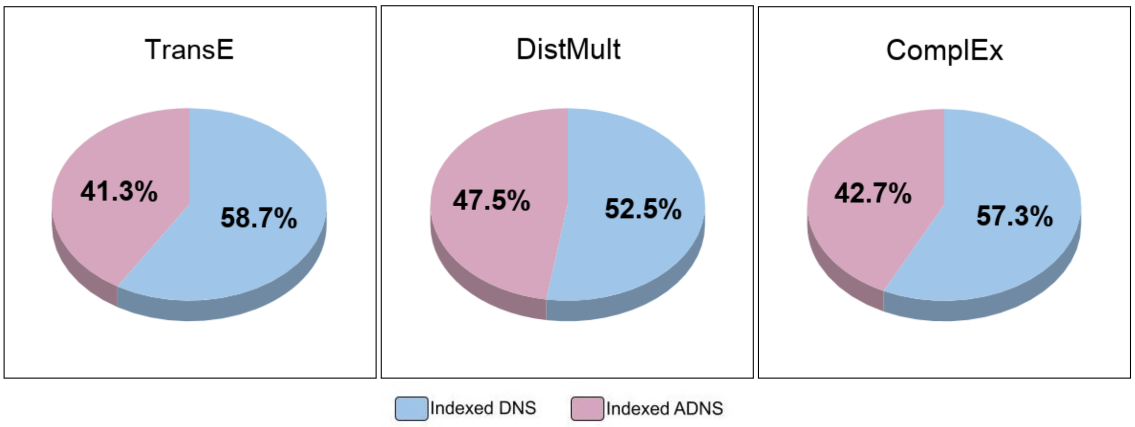


FIGURE 7.14: Training time comparison between $Indexed\ DNS$ and $Indexed\ ADNS$ for Nations

As shown in Figure 7.14 illustrates that the training time of ComplEx with $Indexed\ ADNS$ is around 14.6% less than the training time with $Indexed\ DNS$. As shown in Table 7.8 $Hit@3$ for ComplEx with $Indexed\ ADNS$ is 86.57% contrarily, $Hit@3$ with $Indexed\ DNS$ is 78.86%. This indicates a better performance of $Indexed\ ADNS$. For DistMult, $Indexed\ DNS$ takes 5% more training time than $Indexed\ ADNS$, and for TransE, $Indexed\ ADNS$ takes 17.4% less training time than $Indexed\ DNS$ as exhibited in Figure 7.14.

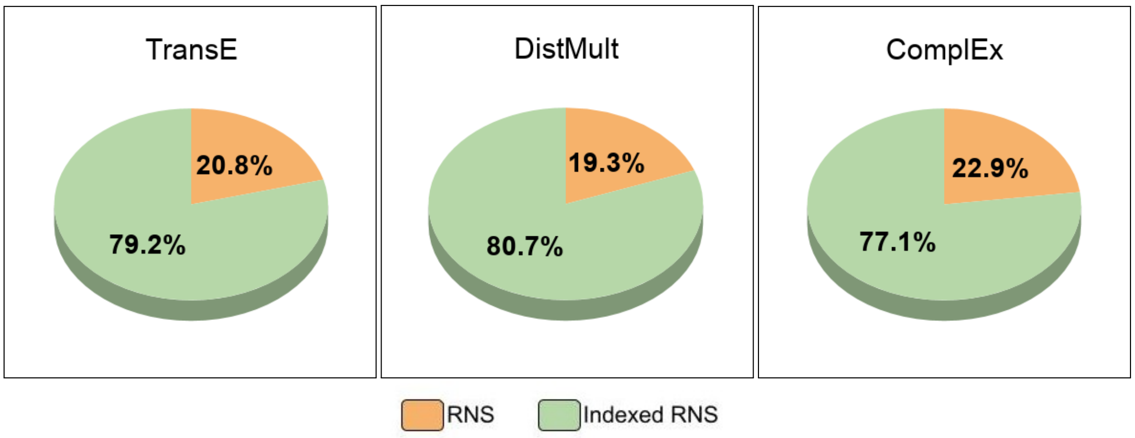


FIGURE 7.15: Training time comparison between RNS and $Indexed\ RNS$ for Nations

$Indexed\ RNS$ for Nations dataset contributes to creating meaningful negative samples. This can be found in Table 7.9. This table shows that among all the KGE models, $Indexed\ RNS$ can be one of the top performers. For example, with

Hit@3 for ComplEx, *Indexed RNS* outperforms ADNS for DistMult and TransE, DNS for TransE.

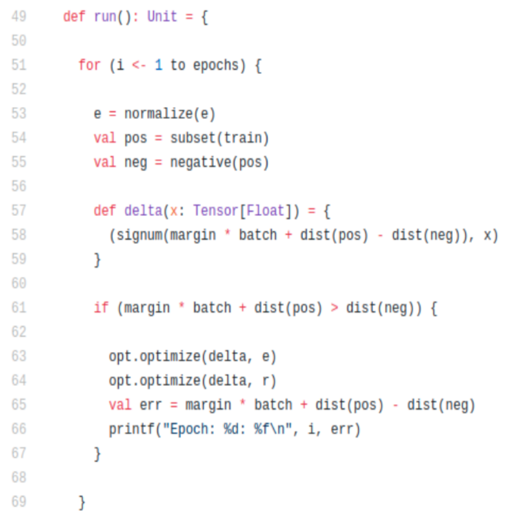
Hit@1		**Hit@3**		**Hit@5**		**Hit@10**	
KGE_NSA	%	**KGE_NSA**	%	**KGE_NSA**	%	**KGE_NSA**	%
TransE_RNS	1.99	TransE_RNS	57.96	TransE_RNS	77.1	TransE_RNS	95.27
TransE_*Indexed ADNS*	6.22	TransE_*Indexed DNS*	58.71	TransE_*Indexed DNS*	78.61	TransE_*Indexed ADNS*	96.77
TransE_*Indexed RNS*	**6.97**	TransE_*Indexed ADNS*	65.92	TransE_*Indexed ADNS*	82.59	TransE_DNS	97.01
TransE_*Indexed DNS*	13.18	**TransE**_*Indexed RNS*	**66.92**	TransE_ADNS	84.08	**TransE**_*Indexed RNS*	**97.26**
TransE_ADNS	18.41	TransE_DNS	69.9	**TransE**_*Indexed RNS*	**84.58**	DistMult_RNS	97.51
TransE_DNS	20.15	TransE_ADNS	70.9	TransE_DNS	85.07	TransE_ADNS	97.51
DistMult_RNS	38.06	DistMult_*Indexed DNS*	72.39	DistMult_RNS	87.52	TransE_*Indexed DNS*	97.76
ComplEx_*Indexed DNS*	42.04	DistMult_RNS	77.61	DistMult_*Indexed DNS*	89.05	DistMult_DNS	98.01
ComplEx_RNS	45.02	ComplEx_RNS	78.11	**DistMult**_*Indexed RNS*	**89.8**	ComplEx_RNS	98.1
DistMult_*Indexed DNS*	51.49	ComplEx_*Indexed DNS*	78.86	ComplEx_*Indexed DNS*	90.55	DistMult_*Indexed DNS*	98.76
ComplEx_*Indexed RNS*	**55.22**	DistMult_*Indexed ADNS*	81.84	ComplEx_RNS	90.8	DistMult_*Indexed ADNS*	99
ComplEx_*Indexed ADNS*	56.47	**DistMult**_*Indexed RNS*	**82.59**	**ComplEx**_*Indexed RNS*	**90.8**	**DistMult**_*Indexed RNS*	**99.25**
DistMult_*Indexed RNS*	**56.72**	DistMult_ADNS	83.33	DistMult_*Indexed ADNS*	92.29	DistMult_ADNS	99.25
DistMult_*Indexed ADNS*	57.96	**ComplEx**_*Indexed RNS*	**83.83**	DistMult_DNS	92.79	**ComplEx**_*Indexed RNS*	**99.5**
DistMult_ADNS	63.43	DistMult_DNS	85.32	ComplEx_*Indexed ADNS*	94.78	ComplEx_DNS	99.5
DistMult_DNS	70.9	ComplEx_*Indexed ADNS*	86.57	DistMult_ADNS	95.52	ComplEx_ADNS	99.5
ComplEx_ADNS	75.37	ComplEx_ADNS	91.04	ComplEx_DNS	96.27	ComplEx_*Indexed DNS*	99.75
ComplEx_DNS	90.8	ComplEx_DNS	94.53	ComplEx_ADNS	96.27	ComplEx_*Indexed ADNS*	99.9

Table 7.9: Comparison of performances among all the Negative Sampling Algorithms (NSA) over different KGE models for Nations

Findings: Overall, it can be qualified that *Indexed DNS* and *Indexed ADNS* positively reduces the training time with respect to DNS and ADNS. And this training time reduction formula can direct to the acceleration of the KGE model's performances or may at least reside at a non-threatening level. Even though the training time of *Indexed RNS* is higher than RNS, it is much lower than the training times of *Indexed DNS*, *Indexed ADNS*, DNS, and ADNS. As a bonus, sometimes *Indexed RNS* is found to be performing outstanding with respect to other Negative Sampling Algorithms.

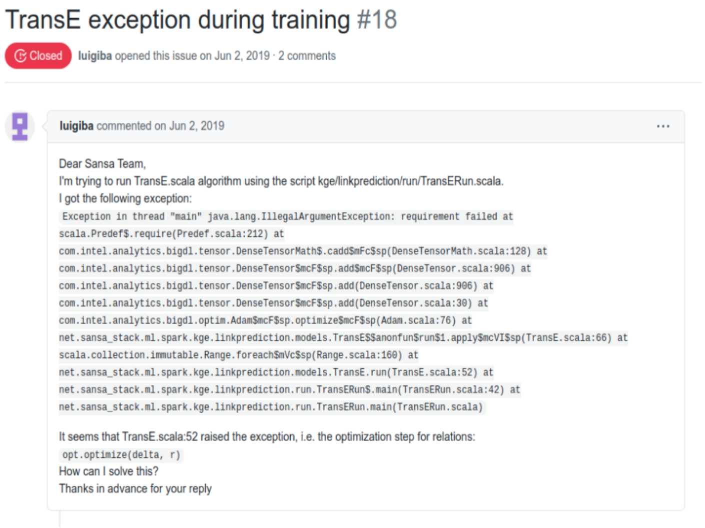
7.4.2 Results And Findings Of Distributed KGE Models

This section distinctively elaborates the results and findings of distributed KGE model's implementations with Spark framework and BigDL library. As mentioned in Chapter 6, distributed approaches of different KGE models are inspired by the implementation of TransE by SANSA-Stack [13]. Nevertheless, the application is still incomplete[1]. The following sections of this chapter will address the wellness of the solution to this issue as proposed in Chapter 6. Simultaneously this section will focus on how distributed TransE, DistMult, and ComplEx perform over three datasets and discuss the approach's impediments.



```
49  def run(): Unit = {
50
51    for (i <- 1 to epochs) {
52
53      e = normalize(e)
54      val pos = subset(train)
55      val neg = negative(pos)
56
57      def delta(x: Tensor[Float]) = {
58        (signum(margin * batch + dist(pos) - dist(neg)), x)
59      }
60
61      if (margin * batch + dist(pos) > dist(neg)) {
62
63        opt.optimize(delta, e)
64        opt.optimize(delta, r)
65        val err = margin * batch + dist(pos) - dist(neg)
66        printf("Epoch: %d: %f\n", i, err)
67      }
68
69    }
```

FIGURE 7.16: Code snippet from SANSA team's implementation



TransE exception during training #18

Closed luigiba opened this issue on Jun 2, 2019 · 2 comments

luigiba commented on Jun 2, 2019

Dear Sansa Team,
I'm trying to run TransE.scala algorithm using the script kge/linkprediction/run/TransERun.scala.
I got the following exception:

```
Exception in thread "main" java.lang.IllegalArgumentException: requirement failed at
scala.Predef$.require(Predef.scala:212) at
com.intel.analytics.bigdl.tensor.DenseTensorMath$.cadd$mFc$sp(DenseTensorMath.scala:128) at
com.intel.analytics.bigdl.tensor.DenseTensor$mcF$sp.add$mcF$sp(DenseTensor.scala:906) at
com.intel.analytics.bigdl.tensor.DenseTensor$mcF$sp.add(DenseTensor.scala:906) at
com.intel.analytics.bigdl.tensor.DenseTensor$mcF$sp.add(DenseTensor.scala:30) at
com.intel.analytics.bigdl.optim.Adam$mcF$sp.optimize$mcF$sp(Adam.scala:76) at
net.sansa_stack.ml.spark.kge.linkprediction.models.TransE$$anonfun$run$1.apply$mcVI$sp(TransE.scala:66) at
scala.collection.immutable.Range.foreach$mVc$sp(Range.scala:160) at
net.sansa_stack.ml.spark.kge.linkprediction.models.TransE.run(TransE.scala:52) at
net.sansa_stack.ml.spark.kge.linkprediction.run.TransERun$.main(TransERun.scala:42) at
net.sansa_stack.ml.spark.kge.linkprediction.run.TransERun.main(TransERun.scala)
```

It seems that TransE.scala:52 raised the exception, i.e. the optimization step for relations:
`opt.optimize(delta, r)`
How can I solve this?
Thanks in advance for your reply

FIGURE 7.17: Reported bug from SANSA team's implementation

Solution to the reported bug in SANSA-Stack:
By the time this thesis work has been in progress, a bug is reported in the GitHub repository of SANSA-Stack[2] on the TransE implementation as shown in Figure 7.17. Upon investigating thoroughly, the apparent reason is found due to different sizes of relation and entity embeddings. In their implementation, as shown in Figure 7.16, two different embeddings are maintained: one for the entity and the other for relation, which is the general scenario of the KGE models the optimizers are called twice, as shown in line 63 and line 64 of Figure 7.16[3]. However, calling

[1]https://github.com/SANSA-Stack/Archived-SANSA-ML/issues/18.
[2]https://github.com/SANSA-Stack
[3]This screenshot from the git repository is taken from the author's local machine

the optimizer on the loss for relation embedding is producing an error as shown in Figure 7.17[4].

The optimizers in BigDL expect two parameters - the loss of the function and all the gradients of the function combined in one tensor[5] Probably, due to this built-in nature, when first the entity embeddings are passed as the gradients, the optimizers may consider it's size as the standard. Numbers of entities and relations may differ, so as their embeddings since the respective embedding's size depends on the number of entities or relations and the dimension given. Thus, when relation embedding with a different size is passed through the same optimizer on the same loss, the dense tensors are not recognized by the optimizers. To solve this, the distributed approach of this thesis proposed maintaining one embedding to hold both relation and entity embeddings.

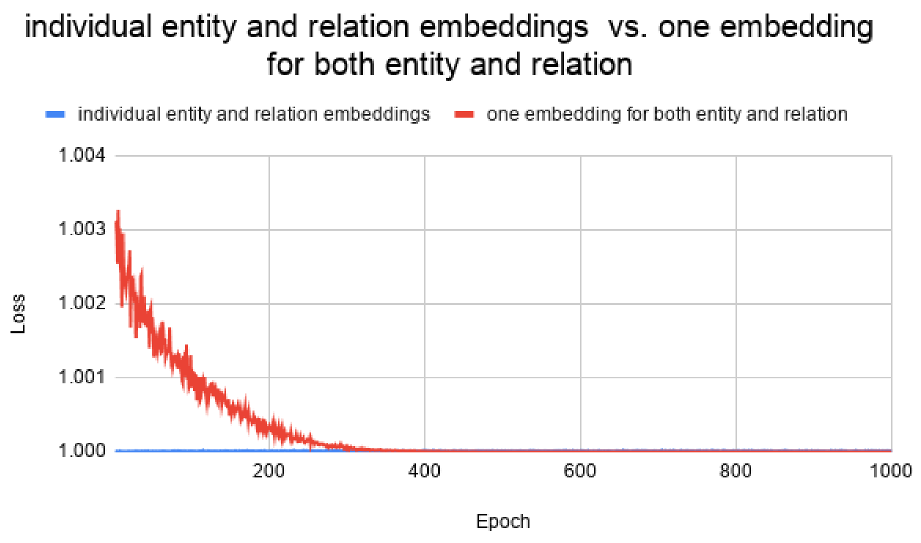


FIGURE 7.18: Loss vs epoch comparison between SANSA team's implementation and the proposed implementation of this thesis

Improved loss:

In the SANSA team's implementation, the optimizer is called over two different tensors, which may often throw an exception. If not, then the loss does not decrease per epoch rather fluctuates. Generally, good training should include a decline in loss after each epoch. Figure 7.18 shows a comparison of loss per epoch for the SANSA team's implementation vs. distributed approach followed in this thesis.

[4]See footnote 3

[5]https://bigdl-project.github.io/0.10.0/APIGuide/Optimizers/Optim-Methods/adagrad

Evidently, the loss does not fluctuate with the distributed approach for having one embedding.

Dataset	**KGE model**	**MR**		**MRR**		**Hit@1 (%)**		**Hit@5 (%)**		**Hit@10 (%)**		**Hit@50 (%)**	
		Raw	**Filter**	**Raw**	**Filter**	**Raw**	**Filter**	**Raw**	**Filter**	**Raw**	**Filter**	**Raw**	**Filter**
Kinship	**TransE**	51	47	1.82E^{-05}	1.97E^{-05}	0.93	0.93	4.93	5.12	9.86	10.52	50.18	53.72
	DistMult	52	48	1.79E^{-05}	1.93E^{-05}	0.74	0.74	2.88	3.16	6.42	7.16	45.62	51.39
	ComplEx	51	48	1.80E^{-05}	1.94E^{-05}	1.58	1.58	5.02	5.21	9.86	10.42	46.18	50
UMLS	**TransE**	68	60	2.21E^{-05}	2.49E^{-05}	0.75	0.9	2.72	3.32	5.9	6.8	35.24	40.99
	DistMult	60	53	2.48E^{-05}	2.81E^{-05}	0.15	0.3	2.26	2.87	5.44	7.41	42.51	50.07
	ComplEx	65	57	2.31E^{-05}	2.63E^{-05}	0.75	0.75	2.57	3.47	6.95	7.71	38.12	43.72
Nations	**TransE**	7	4	7.03E^{-04}	1.08E^{-03}	4.97	7.46	36.31	65.17	84.07	99.5	100	100
	DistMult	6	4	7.71E^{-04}	1.18E^{-03}	6.46	13.93	39.8	71.64	86.56	98.5	100	100
	ComplEx	6	3	8.07E^{-04}	1.25E^{-03}	7.96	18.4	41.79	73.63	87.06	99.5	100	100

Table 7.10: Evaluations of distributed KGE models for different datasets

Table 7.10 shows the results of different distributed KGE models for RNS with dimension, $k = 20$, *learning rate* $= 0.001$, *gamma* $= 2$ and *optimizer* $=$ Adam for 100 epochs. Distributed approaches of these KGE models need more research to figure out why they are performing unexpectedly, as shown in Table 7.10. The MR for Kinship and UMLS datasets are higher, which stipulates that the models are not training immaculately. Upon further inquiry, it is found that the mechanism of the optimizers in BigDL needs more thorough research to fix this issue. A general comparison of Tensor calculation between BigDL and PyTorch has been made to check if the problem resides in the implementation's architecture. However, it is found to be working correctly. Thus, the optimizer is the suspected part to be different in BigDL.

To get into more detail on the probable cause of the non-optimized model, Figure 7.19 can help. Since the nature of loss per epoch in Figure 7.18 implied the unfitting architecture of the KGE models, Figure 7.19 shows a similar pattern. This points out an issue with normalizing the entity embedding. Following the process of normalizing the entity embedding mentioned in Chapter 6, the loss is seen to be fluctuating in Figure 7.19. On the contrary, when the normalization of the embedding is restricted, the loss declines per epoch. One probable cause for such a situation can be, the optimizer may be unable to backtrack the normalization by using a built-in layered function of BigDL as elaborated in Chapter 6. This inspires more into the idea of researching the mechanism of BigDL's optimizer in depth.

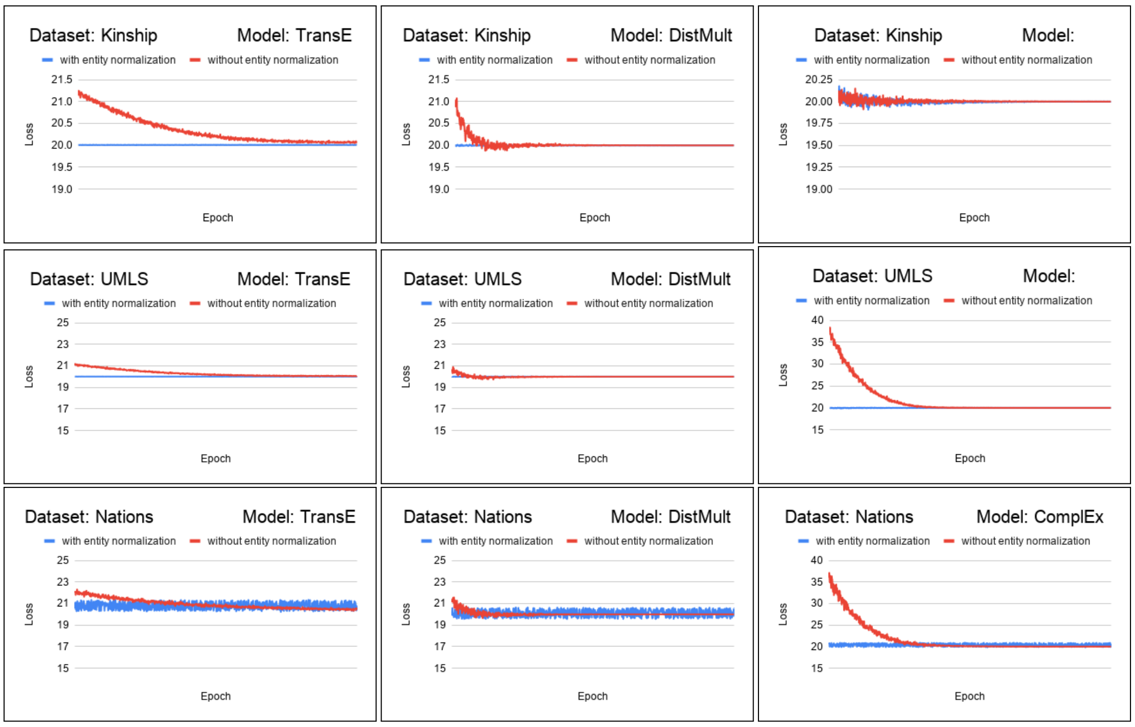


FIGURE 7.19: Loss vs epoch for all KGE models and datasets

Another cause can be constructed from both Figure 7.18 and Figure 7.19. After certain epochs, the loss gets to a plateau by the value of the margin of the margin-based rank criterion. Suppose the margin's value is 1, then the loss per epoch graph gets to a plateau at 1. This can be a case of vanishing or exploding gradients. An issue[6] is put in the community for any experts opinions. By the time of writing this thesis, no one responded. Even by changing parameters, this matter remains unsorted, which compels studying the infrastructure of the optimizer of BigDL even more.

An interesting factor can be seen from Table 7.10. Even though the models are behaving unexpectedly for the Kinship and the UMLS dataset, the evaluation metric's results show impressive behavior for Nations dataset. $Hit@10$ for all the KGE models are more than 99%. MR for Nations dataset, as shown in Table 7.8, are 8 (raw) and 4 (filter) for TransE, 7 (raw) and 2 (filter) for DistMult, and 8 (raw) and 2 (filter) for ComplEx implemented with PyTorch. Table 7.10 shows that the MR for the distributed approach is comparatively close to the sequential approach. The nature of Nations dataset is different from Kinship and UMLS dataset, which

[6]https://stackoverflow.com/questions/61714137/loss-of-transe-gets-plateau-at-the-margin-value-with-bigdl-library

is the ratio of entity and relation numbers as illustrated in Table 4.1. For Nations dataset, the number of entities is higher than the number of relations. Most of the entities are connected by multiple relations. Perhaps the modest training of the distributed KGE models with BigDL results in an outstanding performance for Nations datasets due to these characteristics.

Findings: Considering all the distributed approach's findings comprehensively, it is safe to conclude that studying the optimizer's mechanism from BigDL will enhance future work prospects based on this thesis's findings. For some datasets, the proposed distributed KGE models are working well; however, resolving the current issues with all the acquired insights will help achieve the KGE model's scalability for other datasets.

Chapter 8

Conclusion And Future Work

A part of this thesis considered the Negative Sampling Algorithm, an important but often overlooked part of the Knowledge Graph Embedding models. Creating meaningful negative samples is important for better training. Better trained Knowledge Graph Embedding models mean better applications of the Knowledge Graph can be achieved. This thesis has been interested in optimizing the Knowledge Graph Embedding model's overall training time, which is another important aspect concerning the growing Knowledge Graphs in the Big Data scenario. Implementing the models with the proposed Indexed Negative Sampling Algorithm can reduce a significant amount of processing time without compromising the performance much. Another part of this thesis is focused on the distributed approach of the Knowledge Graph Embedding models. Again, with the exponentially growing Knowledge Graphs, upgrading the computing machines frequently to execute the models is a big obstacle. Thus, accomplishing the distributed Knowledge Graph Embedding models can help with this overhead.

8.1 Conclusion

Three Negative Sampling Algorithms: RNS, DNS, and ADNS have been chosen to inject the proposed concept named *Indexed Dataset* to create meaning negative samples with an optimized training time. The three proposed variants are named *Indexed RNS*, *Indexed DNS*, and *Indexed ADNS*. *Indexed Dataset* fundamentally arranges the train dataset's indexes in a new table so that the KGE model can know which entities should not be chosen to corrupt a triple. This excludes the general necessity of checking if the generated corrupted triple is true for the train data. This helps to reduce the processing time of original algorithms of DNS and ADNS notably. Not only that, *Indexed RNS*, *Indexed DNS*, and *Indexed ADNS* can create more meaningful negative samples compared to RNS. Moreover, *Indexed RNS*, which is a variation of RNS, seldom performs exceptionally well compared to other Negative Sampling Algorithms. The training time with *Indexed RNS* is the lowest among DNS, ADNS, *Indexed DNS*, and *Indexed ADNS*. Three KGE models: TransE, DistMult, and ComplEx are used to evaluate the performances of the Negative Sampling Algorithms with Kinship, UMLS, and Nations datasets. For all three datasets, often ComplEx with Indexed Negative Sampling Algorithm outperforms other models with other Negative Sampling Algorithms. The training time's trimming is also among the highest ones for ComplEx and Indexed Negative Sampling Algorithms.

Although distributed KGE models should be continued for further research purposes, much deeper insights into the BigDL library are achieved in this thesis. The learning curve of the library is found to be steep for a beginner. Optimization with BigDL still needs some exploration to properly implement the distributed KGE models, although the basic structure is proposed in this thesis. The data preparation and the evaluation implemented in this thesis with Spark are running at full throttle. Even the data preparation and the testing of the KGE models with big datasets may impose an overhead. The codes prepared for these purposes can be parallelly executed by multiple worker nodes with Spark easily. So far, even the training is running with multiple cores of the local machine. Kinship, UMLS, and Nations datasets are used for the evaluation. Surprisingly, Nations dataset outperforms all three distributed KGE models.

8.2 Future Work

Firstly, the proposed *Indexed Negative Sampling Algorithm's* evaluations need to be conducted in the future with bigger datasets. For example, FB15K, WN18, etc. Another future work with these algorithms is to create deeper clusters of the *Indexed Dataset* proposed in this thesis. For example, the *Indexed Dataset* comprises a list of heads and tails with respect to a specific relation. However, some heads may only pair with certain tails and vice versa. For example, "*lives in*" relation in the KG has three triples: (*Tasneem*, *lives in*, *Bonn*), (*Drea*, *lives in*, *Bonn*) and (*Faheem*, *lives in*, *Bangladesh*). Now, for the relation, the list of heads will include {*Tasneem*, *Drea*, *Faheem*} and the list of tails will be {*Bonn*, *Bangladesh*}. While corrupting the head of the triple (*Tasneem*, *lives in*, *Bonn*), entity "*Faheem*" can be a notable option. However, the current *Indexed Dataset* will not allow that because it is subtracted from the entire entity list. If the *Indexed Dataset* can separate the list of heads with "*Tasneem*" and "*Drea*" from "*Faheem*" as they both live in "*Bonn*" then it can be checked if the results of the models increase by selecting "*Faheem*".

Distributed KGE models have much in-depth future work. As mentioned already, the optimizer's mechanism with BigDL needs more research. Solving the obstacles that have appeared in this thesis can contribute to achieving the distributed KGE model's *scalability*. BigDL is more famous for training neural network models in parallel mode over batches. This thesis's further future work is to create a layered model of the KGE models to exploit this benefit provided by the library.

Combining parallel optimization with meaningful Indexed Negative Sampling Algorithm shows great future aspects of the processing time and the KGE model's performance.

Bibliography

[1] D Lenat and E Feigenbaum. On the thresholds of knowledge. *Foundations of Artificial Intelligence, MIT Press, Cambridge, MA*, pages 185–250, 1992.

[2] David Parkins. The world's most valuable resource is no longer oil, but data. *The economist*, 6, 2017.

[3] Yankai Lin, Zhiyuan Liu, Maosong Sun, Yang Liu, and Xuan Zhu. Learning entity and relation embeddings for knowledge graph completion. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 29, 2015.

[4] Maximilian Nickel, Volker Tresp, and Hans-Peter Kriegel. A three-way model for collective learning on multi-relational data. In *Icml*, 2011.

[5] Yongqi Zhang, Quanming Yao, Yingxia Shao, and Lei Chen. Nscaching: simple and efficient negative sampling for knowledge graph embedding. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, pages 614–625. IEEE, 2019.

[6] Léon Bottou. Stochastic gradient descent tricks. In *Neural networks: Tricks of the trade*, pages 421–436. Springer, 2012.

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

[8] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of machine learning research*, 12(7), 2011.

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

[10] Zhen Wang, Jianwen Zhang, Jianlin Feng, and Zheng Chen. Knowledge graph embedding by translating on hyperplanes. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 28, 2014.

[11] Mirza Mohtashim Alam, Hajira Jabeen, Mehdi Ali, Karishma Mohiuddin, and Jens Lehmann. Affinity dependent negative sampling for knowledge graph embeddings. 2020.

[12] Joost Verbraeken, Matthijs Wolting, Jonathan Katzy, Jeroen Kloppenburg, Tim Verbelen, and Jan S Rellermeyer. A survey on distributed machine learning. *ACM Computing Surveys (CSUR)*, 53(2):1–33, 2020.

[13] Ivan Ermilov, Jens Lehmann, Gezim Sejdiu, Lorenz Bühmann, Patrick Westphal, Claus Stadler, Simon Bin, Nilesh Chakraborty, Henning Petzka, Muhammad Saleem, et al. The tale of sansa spark. In *International Semantic Web Conference (Posters, Demos & Industry Tracks)*, 2017.

[14] Branislav Holländer. Accelerating Deep Learning Using Distributed SGD — An Overview. `https://towardsdatascience.com/accelerating-deep-learning-using-distributed-sgd-an-overview-e66c4aee1a0c`, 2018. [Online; accessed 10-October-2020].

[15] Antoine Bordes, Nicolas Usunier, Alberto Garcia-Duran, Jason Weston, and Oksana Yakhnenko. Translating embeddings for modeling multi-relational data. *Advances in neural information processing systems*, 26:2787–2795, 2013.

[16] Bishan Yang, Wen-tau Yih, Xiaodong He, Jianfeng Gao, and Li Deng. Learning multi-relational semantics using neural-embedding models. *arXiv preprint arXiv:1411.4072*, 2014.

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

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

[19] 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.

[20] 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, 2008.

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

[22] 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.

[23] Woodrow W Denham. *The detection of patterns in Alyawara nonverbal behavior.* PhD thesis, University of Washington., 1973.

[24] Rudolph J Rummel. Dimensionality of nations project. Technical report, HAWAII UNIV HONOLULU DEPT OF POLITICAL SCIENCE, 1968.

[25] Alexa T McCray. An upper-level ontology for the biomedical domain. *Comparative and Functional Genomics*, 4(1):80–84, 2003.

[26] Amit Singhal. Introducing the knowledge graph: things, not strings. *Official google blog*, 5:16, 2012.

[27] 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.

[28] Georg Thimm and Emile Fiesler. Neural network initialization. In *International Workshop on Artificial Neural Networks*, pages 535–542. Springer, 1995.

[29] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pages 249–256. JMLR Workshop and Conference Proceedings, 2010.

[30] Bhushan Kotnis and Vivi Nastase. Analysis of the impact of negative sampling on link prediction in knowledge graphs. *arXiv preprint arXiv:1708.06816*, 2017.

[31] Xiaofei Zhou, Qiannan Zhu, Ping Liu, and Li Guo. Learning knowledge embeddings by combining limit-based scoring loss. In *Proceedings of the 2017 ACM on Conference on Information and Knowledge Management*, pages 1009–1018, 2017.

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

[33] Mojtaba Nayyeri, Sahar Vahdati, Jens Lehmann, and Hamed Shariat Yazdi. Soft marginal transe for scholarly knowledge graph completion. *arXiv preprint arXiv:1904.12211*, 2019.

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

[35] 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.

[36] Yuanfei Dai, Shiping Wang, Neal N Xiong, and Wenzhong Guo. A survey on knowledge graph embedding: Approaches, applications and benchmarks. *Electronics*, 9(5):750, 2020.

[37] Yankai Lin, Zhiyuan Liu, Huanbo Luan, Maosong Sun, Siwei Rao, and Song Liu. Modeling relation paths for representation learning of knowledge bases. *arXiv preprint arXiv:1506.00379*, 2015.

[38] X Let. Pattern classification. 2005.

[39] Antoine Bordes, Xavier Glorot, Jason Weston, and Yoshua Bengio. A semantic matching energy function for learning with multi-relational data. *Machine Learning*, 94(2):233–259, 2014.

[40] Shanchan Wu, Kai Fan, and Qiong Zhang. Improving distantly supervised relation extraction with neural noise converter and conditional optimal selector. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pages 7273–7280, 2019.

[41] Jason Weston, Antoine Bordes, Oksana Yakhnenko, and Nicolas Usunier. Connecting language and knowledge bases with embedding models for relation extraction. *arXiv preprint arXiv:1307.7973*, 2013.

[42] Antoine Bordes, Jason Weston, and Nicolas Usunier. Open question answering with weakly supervised embedding models. In *Joint European conference on machine learning and knowledge discovery in databases*, pages 165–180. Springer, 2014.

[43] Antoine Bordes, Sumit Chopra, and Jason Weston. Question answering with subgraph embeddings. *arXiv preprint arXiv:1406.3676*, 2014.

[44] Fuzheng Zhang, Nicholas Jing Yuan, Defu Lian, Xing Xie, and Wei-Ying Ma. Collaborative knowledge base embedding for recommender systems. In *Proceedings of the 22nd ACM SIGKDD international conference on knowledge discovery and data mining*, pages 353–362, 2016.

[45] Nikhil Ketkar. Introduction to pytorch. In *Deep learning with python*, pages 195–208. Springer, 2017.

[46] Wes McKinney et al. pandas: a foundational python library for data analysis and statistics. *Python for High Performance and Scientific Computing*, 14 (9):1–9, 2011.

[47] Guido Van Rossum et al. Python, 1991.

[48] Apache Spark. Apache spark. *Retrieved January*, 17:2018, 2018.

[49] Jason Jinquan Dai, Yiheng Wang, Xin Qiu, Ding Ding, Yao Zhang, Yanzhang Wang, Xianyan Jia, Cherry Li Zhang, Yan Wan, Zhichao Li, et al. Bigdl: A distributed deep learning framework for big data. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 50–60, 2019.

[50] Martin Odersky, Lex Spoon, and Bill Venners. *Programming in scala.* Artima Inc, 2008.

[51] Richard Socher, Danqi Chen, Christopher D Manning, and Andrew Ng. Reasoning with neural tensor networks for knowledge base completion. In *Advances in neural information processing systems*, pages 926–934. Citeseer, 2013.

[52] David L Tennenhouse and David J Wetherall. Towards an active network architecture. In *Proceedings DARPA Active Networks Conference and Exposition*, pages 2–15. IEEE, 2002.

[53] Faisal Rahutomo, Teruaki Kitasuka, and Masayoshi Aritsugi. Semantic cosine similarity. In *The 7th International Student Conference on Advanced Science and Technology ICAST*, volume 4, page 1, 2012.

[54] Eric W Weisstein. Bernoulli distribution. *https://mathworld. wolfram. com/*, 2002.

[55] Mirza Mohtashim Alam. Rulect: A system for extraction,representation and learning ofrelational patterns on knowledge graph embeddings. Master's thesis, Rheinische Friedrich-Wilhelms-Universität Bonn, 12 2020.

[56] Yanrong Wu and Zhichun Wang. Knowledge graph embedding with numeric attributes of entities. In *Proceedings of The Third Workshop on Representation Learning for NLP*, pages 132–136, 2018.

[57] Andrea Rossi, Denilson Barbosa, Donatella Firmani, Antonio Matinata, and Paolo Merialdo. Knowledge graph embedding for link prediction: A comparative analysis. *ACM Transactions on Knowledge Discovery from Data (TKDD)*, 15(2):1–49, 2021.

List of Figures

List of Tables