**KNOWLEDGE GRAPH EMBEDDING USING TRANSE**

A Project Progress Report

Submitted in partial fulfillment for the degree of

**BACHELOR OF TECHNOLOGY**

**in**

**COMPUTER SCIENCE ENGINEERING**

Submitted by

N VEMACHALAM (N180701)

N SHANTHA KUMARI (N180139)

S NIKHIL (N180978)

K DHANESH (N180049)

G INDAMMA (N180053)

*Under the Esteem Guidance of*

**Mr. Rayala Upendar Rao**



**DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING**

**Rajiv Gandhi University of Knowledge Technologies – Nuzvid**

**Nuzvid, Krishna, Andhra Pradesh – 521202.**



**DEPARTMENT OF COMPUTER SCIENCE ENGINEERING**

**Rajiv Gandhi University of Knowledge Technologies – Nuzvid**

**Nuzvid, Krishna, Andhra Pradesh – 521202.**

**CERTIFICATE OF COMPLETION**

This is to certify that the work entitled, **“Knowledge Graph Embedding using TransE”**is the bonafied work of **N.VEMACHALAM (ID No*:* N180701), N.SANTHA KUMARI (ID No*:* N180139)*,* S.NIKHIL *(*ID No*:* N180978)*,* K.DHANESH *(*ID No *:* N180049)*,* G.INDAMMA *(*ID No : N180053)** carried out under my guidance and supervision for 3dr year project of **Bachelor of Technology** in the department of Computer Science and Engineering under RGUKT IIIT Nuzvid. This work is done during the academic session February 2022 – July 2023, under our guidance.

---------------------------------------- -------------------------------------

**Mr. Rayala Upendar Rao Dr.Chiranjeevi Sadu**

Assistant professor, Assistant Professor,

Department of CSE, Head of the Department,

RGUKT Nuzvid. Department of CSE,

RGUKT Nuzvid



**DEPARTMENT OF COMPUTER SCIENCE ENGINEERING**

**Rajiv Gandhi University of Knowledge Technologies – Nuzvid**

**Nuzvid, Krishna, Andhra Pradesh – 521202.**

**CERTIFICATE OF EXAMINATION**

This is to certify that the work entitled, **“Knowledge Graph Embedding using TransE ”** is the bonafied work of **N.VEMACHALAM (ID No*:* N180701), N.SANTHA KUMARI (ID No*:* N180139)*,* S.NIKHIL *(*ID No*:* N180978)*,* K.DHANESH *(*ID No*:*N180049)*,* G.INDAMMA *(*ID No: N180053)** and here by accord our approval of it as a study carried out and presented in a manner required for its acceptance in 3rd year of **Bachelor of Technology** for which it has been submitted. This approval does not necessarily endorse or accept every statement made, opinion expressed or conclusion drawn, as a recorded in this thesis. It only signifies the acceptance of this thesis for the purpose for which it has been submitted.

------------------------------------------- ------------------------------

**Mr.Rayala Upendar Rao Project Examiner**

Assistant Professor, RGUKT-Nuzvid.

Department of CSE,

RGUKT-NUZVID



**DEPARTMENT OF COMPUTER SCIENCE ENGINEERING**

**Rajiv Gandhi University of Knowledge Technologies – Nuzvid**

**Nuzvid, Krishna, Andhra Pradesh – 521202.**

**DECLARATION**

**“N. VEMACHALAM (ID No*:* N180701), N.SANTHA KUMARI(ID No*:* N180139)*,* S.NIKHIL *(*ID No*:* N180978)*,* K.DHANESH *(*ID No*:*N180049)*,* G.INDAMMA *(*IDNo:N180053)** hereby declare that the project report entitled **“Knowledge Graph Embedding using TransE”** done by us under the guidance of **Mr. Upendar Rao**, Assistant Professor is submitted for the partial fulfillment for the award of degree of Bachelor of Technology in Computer Science and Engineering during the academic session November 2022-February 2023 at RGUKT-Nuzvid.

We also declare that this project is a result of our own effort and has not been copied or imitated from any source. Citations from any websites are mentioned in the references. The results embodied in this project report have not been submitted to any other university or institute for the award of any degree or diploma.

**Date:**

**Place: Nuzvid**

N VEMACHALAM (N180701)

N SHANTHA KUMARI (N180139)

S NIKHIL (N180978)

K DHANESH (N180049)

G INDAMMA (N180053)

**ACKNOWLEDGEMENT**

We would like to express our profound gratitude and deep regards to our guide **Mr. Rayala Upendar Rao** for his exemplary guidance, monitoring and constant encouragement to us throughout the B.Tech course. We shall always cherish the time spent with him during the course of this work due to the invaluable knowledge gained in the field of reliability engineering.

We are extremely grateful for the confidence bestowed in us and entrusting our project entitled **“Knowledge Graph Embedding using TransE”.**

We express gratitude to Mr. Chiranjeevi sadhu (HOD of CSE Dept) and other faculty members for being source of inspiration and constant encouragement which helped us in completing the project successfully.

Our sincere thanks to all the batchmates of 2018 CSE, who have made our stay at RGUKT-NUZVID, a memorable one.

Finally, yet importantly, we would like to express our heartfelt thanks to our beloved God and parents for their blessings, our friends for their help and wishes for the successful completion of this project.

# 

# **ABSTRACT**

Consider the problem of embedding entities and relationships of multi-relational data in low-dimensional vector spaces. Our objective is to propose a canonical model which is easy to train, contains a reduced number of parameters and can scale up to very large databases. Hence considering TransE, a method which models relationships by interpreting them as translations operating on the low-dimensional embeddings of the entities. Despite its simplicity, this assumption proves to be powerful since extensive experiments show that TransE significantly outperforms state-of-the-art methods in link prediction on two knowledge bases. Besides, it can be successfully trained on a large scale data set with 1M entities, 25k relationships and more than 17M training samples.

**Keywords:** TransE embeddings, multi-relations, Semantic Relationships, Negative Sampling, Scoring Function, Loss function, Positive triplets, Negative triplets, Link prediction.

**Table of Contents**

**ABSTRACT** 6

**CHAPTER 1** 9

**INTRODUCTION** 9

**1.1 Multi relational data** 9

**1.2 Knowledge Graph** 10

**1.3 TransE** 12

**1.4 Knowledge graph Embedding usig TransE**  14

**CHAPTER 2** 15

**REQUIREMENTS AND ANALYSIS** 15

**2.1 Hardware components** 15

**2.2 Software components** 15

**2.3 Functional requirements** 15

**2.4 Non-Functional requirements** 15

**CHAPTER 3** 16

**PROPOSED MODEL AND FLOW OF THE PROJECT** 16

**3.1 Proposed model : TransE Algorithm** 16

**3.2 Flow of the project** 19

**3.3 Methodology** 21

3.3.1 Modeling relations 22

**3.4 Applications** 24

**CHAPTER 4** 25

**IMPLEMENTATION** 25

**4.1 Importing the modules**  25

**4.2 Data Preparation** 26

**4.3 Model evalution** 31

**4.4 Model executing** 32

**4.5 Different databases** 41

**4.6 Example commands** 41

**4.4 Execution output** 42

**CHAPTER 5** 45

**OUTPUT** 45

**CHAPTER 6** 50

**CHAPTER 7** 51

**List of figures**

Figure 1 Multi relational data 10

Figure 2 Knowledge Graph 11

Figure 3 Simple illstration of TransE 12

Figure 4 Simple Algorithm of TransE 16

Figure 5 Datasets 41

Figure 6 run.py output 43

Figure 7 FB15K dataset 45

Figure 8 Prediction Example 45

Figure 9 FB15k result 46

Figure 10 WN18 dataset results 46

Figure 11 Assymetric relations in WN18 dataset - TransE 47

Figure 12 Assymetric relation in WN18 dataset - DistMult 48

Figure 13 Comaprison of Models 49

# **CHAPTER 1**

# **INTRODUCTION**

* 1. **Multi-relational data**

Multi-relational databases are large-scale databases that store structured information about various aspects of the world. Projects like DBPedia, Freebase, and Google Knowledge Vault have attempted to create such knowledge bases by extracting and organizing structured data from different sources. These knowledge bases provide a valuable resource for applications such as recommender systems, question answering, and automated personal agents. By leveraging the structured knowledge in these bases, developers can enhance recommendation algorithms, provide accurate answers to user queries, and create intelligent agents that understand user requests and make informed decisions.

**Characteristics of Multi-relational data:**

* **Multiple Types of Relations:** Multi-relational data involves diverse types of relations connecting entities. Each relation represents a distinct semantic or functional relationship between entities. For example, in a social network, relations can include "friendship," "follows," or "works with."
* **Heterogeneous Connections:** Entities in a multi-relational dataset can have varying types and degrees of connections. Different entities can be connected by different sets of relations, leading to a heterogeneous network structure. This complexity adds richness and diversity to the data.
* **Semantic Variability:** Relations in multi-relational data often exhibit semantic variability. A single relation can have different interpretations or meanings depending on the context or domain. For example, the relation "part of" can represent the membership of an entity in one context and the containment in another.
* **Granularity Levels**: Multi-relational data can involve relations at different granularity levels. Some relations may capture high-level connections between entities, while others may represent more detailed or specific relationships. This provides a multi-faceted view of the data and allows for analyzing the data at different levels of abstraction.
* **Complex Network Structure:** Multi-relational data typically forms complex network structures with interconnected entities and relations. The presence of multiple relations adds additional dimensions and layers of complexity to the relationships, making the data more intricate and challenging to model.
* **Data Sparsity:** Multi-relational datasets often suffer from data sparsity, meaning that not all possible relations or connections between entities are explicitly present in the dataset. There can be many missing links or unknown relationships, requiring models to infer or predict missing connections.

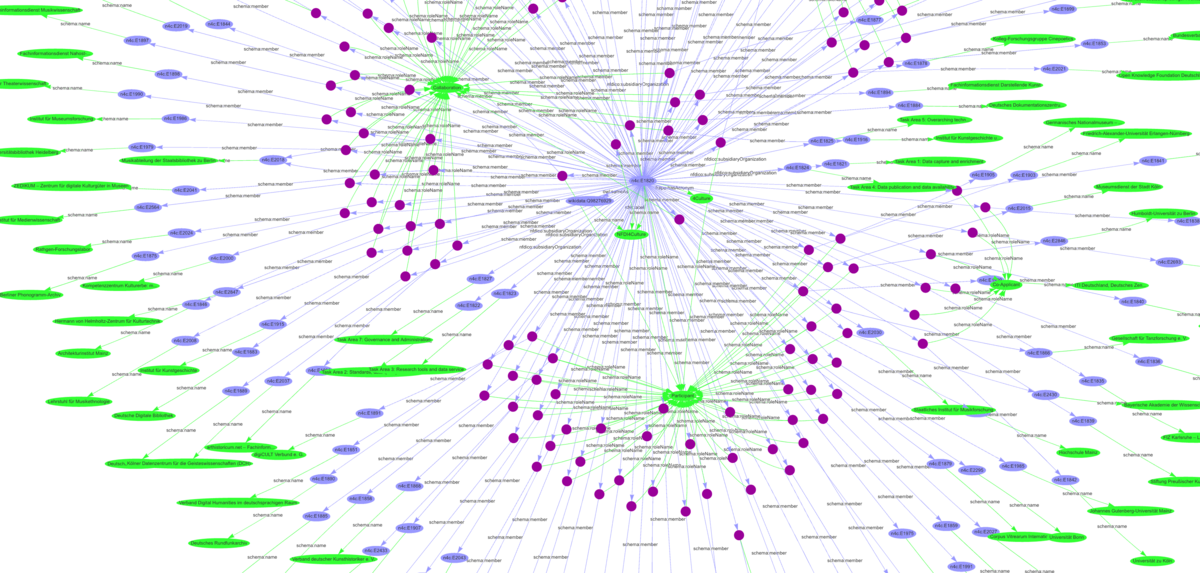


Figure 1 multi-relational data

## **Knowledge graphs :**

A knowledge graph is a structured representation of knowledge that captures information about entities, their attributes, and the relationships between them. It provides a way to organize and model knowledge in a graph-like structure, where nodes represent entities and edges represent relationships.

**Key aspects of knowledge graphs:**

**1. Entity-Relationship Representation:** Knowledge graphs represent real-world entities as nodes and the relationships between them as edges. Entities can be people, places, organizations, concepts, or any other object of interest. Relationships capture the connections, associations, or dependencies between entities.

**2. Rich Semantic Information:** Knowledge graphs often use ontologies, taxonomies, or schemas to define the types of entities and relationships and their properties. This enables more meaningful and precise representation of knowledge.

**3. Heterogeneous Data Integration:** They integrate diverse data types and formats, including text, numbers, images, and even temporal information. By unifying data from multiple sources, knowledge graphs create a unified view of the information.

**4. Scalability and Extensibility:** They can handle billions of nodes and edges and are designed to support efficient querying, navigation, and inference over the graph structure.

**5. Reasoning and Inference:** By leveraging the relationships and semantic information encoded in the graph, it becomes possible to infer new knowledge, make deductions, and answer complex queries that go beyond the explicitly stated facts.

**6. Applications:** Knowledge graphs have a wide range of applications, including semantic search, question answering systems, recommendation engines, information retrieval, data integration, knowledge management, and AI applications such as natural language understanding and reasoning.

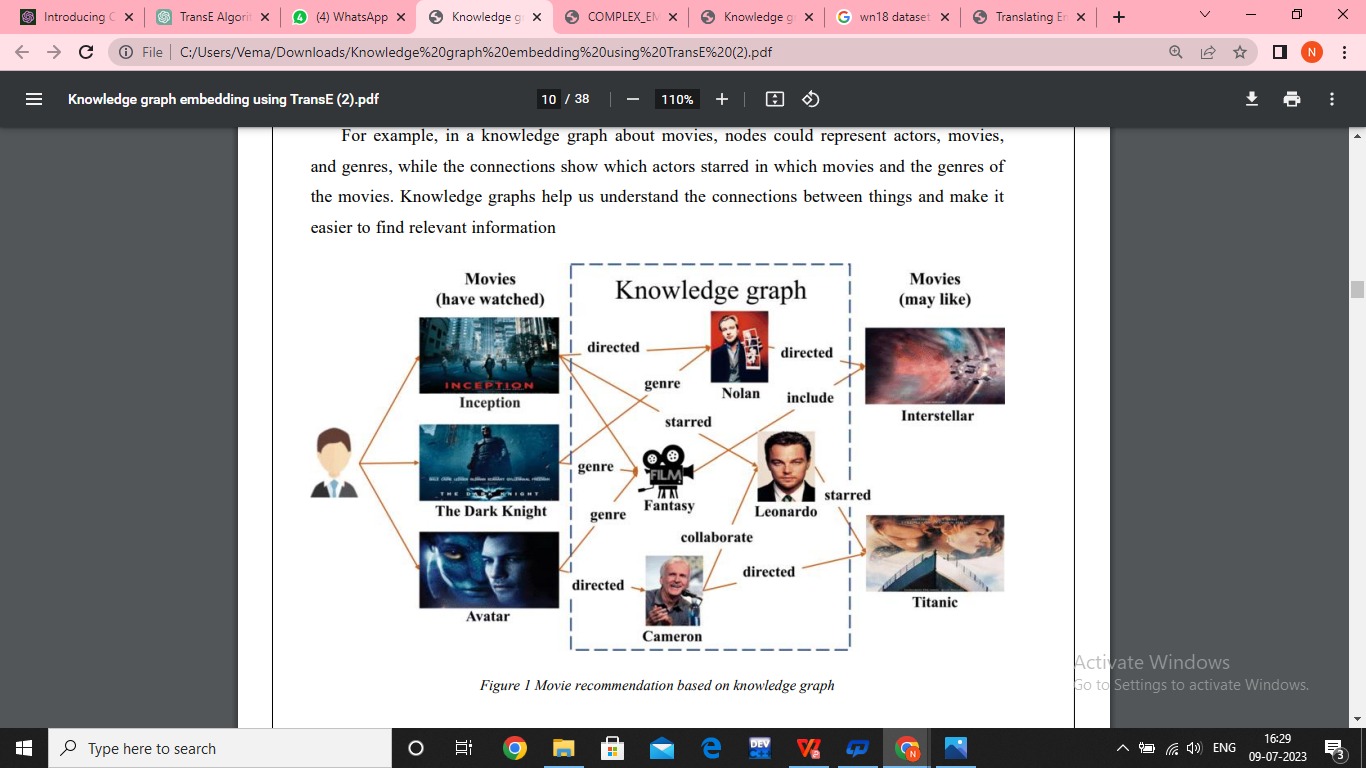


Figure 2 Knowledge Graph

## **1.3 TransE :**

TransE is a knowledge graph embedding model that represents entities and relations as vectors in a continuous vector space. It aims to capture semantic relationships between entities and relations by learning vector representations that preserve the translation property. The key idea behind TransE is that the relation vector can be obtained by adding the vector representation of the head entity to the vector representation of the relation, resulting in a vector close to the vector representation of the tail entity. TransE has been widely used for various knowledge graph tasks, such as link prediction and entity alignment.

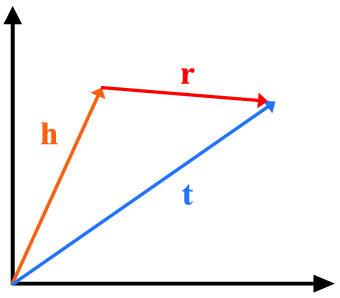
****

Figure 3 Simple illustration of TransE

like any modelhas certain limitations that should be considered. Some limitations of TransE include

**1. Inability to Capture Complex Relationships:** TransE assumes a simple translation-based relationship between entities and relations. It may struggle to capture more complex relationships, such as hierarchical or symmetric relationships, that exist in the knowledge graph.

**2. Lack of Explicit Semantics:** TransE does not explicitly model the semantics of relations. It treats relations as opaque translations in the latent space, which may limit its ability to capture nuanced relationships or handle polysemous relations

**3. Difficulty with One-to-Many or Many-to-One Relationships:** TransE assumes one-to-one relationships between entities, meaning each entity is connected to a unique tail entity through a specific relation. It may face challenges when dealing with one-to-many or many-to-one relationships, where multiple entities can be connected to a single entity or vice versa.

**4. Sensitivity to Imbalanced Data:** TransE's performance can be affected by imbalanced data, where certain relations or entities have significantly more or fewer instances than others. This can lead to biases in the learned embeddings and impact the accuracy of link prediction.

**5. Scalability for Large Knowledge Graphs:** As the size of the knowledge graph increases, the computational and memory requirements of TransE can become significant. Training and inference on large-scale knowledge graphs can be time-consuming and resource-intensive.

Implementation of Link prediction using TransE approach typically involves the following activities,

**1. Feature Extraction**: Various features are computed for each pair of nodes based on the network structure. These features can include common neighbours, node attributes, proximity measures, structural properties, and other network-based metrics. The goal is to capture information that can discriminate between connected and non-connected node pairs.

**2. Training Data Preparation:** A subset of known links and non-links (negative examples) from the existing network is used to create a training set. The known links serve as positive examples, while the negative examples are randomly sampled pairs of nodes that are not connected.

**3. Model Training:** Machine learning models, such as logistic regression, support vector machines, or graph neural networks, are trained on the training set using the extracted features. The models learn patterns and relationships from the available data to predict the likelihood of future connections.

1. **Prediction and Evaluation:** The trained model is then used to predict the likelihood of connections for unseen node pairs. These predictions are evaluated against the known ground truth (existing connections or future connections that become available later) to assess the accuracy and performance of the link prediction model.

## **1.4 Knowledge Graph embedding using TransE :**

In multi-relational data, we have a network of entities and relationships between them. These relationships can represent things like friendships in a social network, buying behaviors in recommender systems, or facts in a knowledge base.To make sense of this data, we need models that can capture the connections between entities and predict new relationships. One popular model for this task is TransE.

In TransE, relationships are represented as translations in the embedding space. If a triple (h, r, t) holds, the embedding of the tail entity t should be close to the embedding of the head entity h plus a vector specific to the relationship . TransE uses a reduced set of parameters, learning only one low-dimensional vector for each entity and relationship. The idea behind TransE is to capture the hierarchical structure often found in knowledge bases.

For example, if we think of a tree structure, siblings are close to each other, and parent- child relationships can be represented as translations. TransE focuses on representing these important relationships using a minimal number of parameters, making it both efficient and scalable. What's interesting is that TransE's translation-based approach has shown promising results in various applications. It has outperformed other complex models in predicting links in knowledge bases, even though it has a simpler architecture.

TransE can also handle large datasets with millions of entities, thousands of relationships, and millions of training samples. TransE's translation-based approach simplifies the modeling of relationships in multi- relational data, leading to better trade-offs between accuracy and scalability. TransE is introduced as a simple and efficient model for learning low-dimensional embeddings of entities in multi- relational data.

**Advantages:**

* Expressive Representation,
* Simplicity
* Improved prediction performances
* Scalable etc.,

**Disadvantages:**

* Increased complexity
* Sensitive
* Limited

**CHAPTER 2**

# **REQUIREMENTS AND ANALYSIS**

## **2.1 Hardware Requirements**

* Processor : Intel Xenon
* RAM : 16GB (approx.)
* Disk : 320GB

## **2.2 Software Requirements**

* Operating System : Windows 11 (64-bit)
* Programming Language : Python 3.7
* Library & Modules used : numPy 1.20.1, torch,dataloader,KGE model,random
* Platform used : Visual Studio Code

## **2.3 Functional Requirements**

* A theano-based Stochastic Gradient Descent implementation is required.
* Ensuring that the implementation can handle large-scale knowledge graphs efficiently.
* Supporting the completion of knowledge graphs by inferring missing triples or predicting potential relationships based on existing knowledge.

## **2.4 Non - functional Requirements**

* Scalable
* Performance
* Maintainability
* Compatibility

# 

# **CHAPTER 3**

# **TransE AND FLOW OF THE PROJECT**

## **3.1 TransE Algorithm**

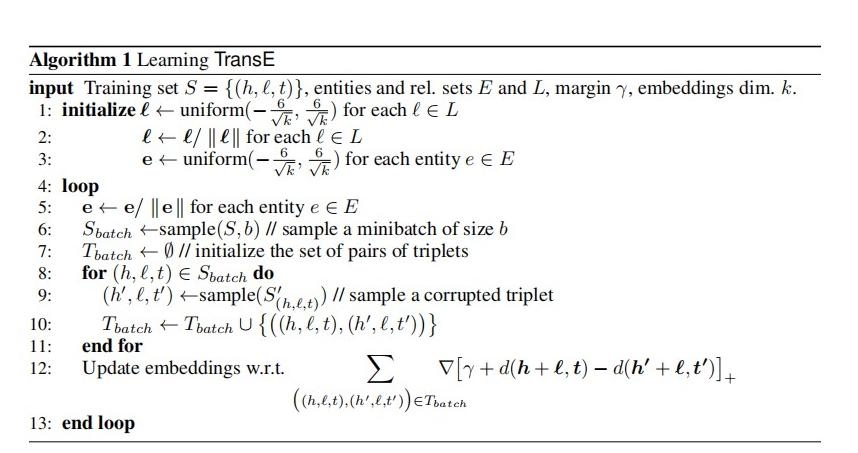


Figure 4 Simple Algorithm of TransE

**Initialization**

Initialize embeddings € for each relation l in the relation set L. The initial values can be sampled uniformly from a specified range. Initialize embeddings e for each entity e in the entity set E. The initial values can also be sampled uniformly from a specified range.

**Training loop:**

The algorithm enters a loop where it iterates over the following steps:

**Sampling:** Sbatch: A minibatch of size b is sampled from the training set S. It randomly selects b triplets (h, l, t) from the training set S.

**Tbatch:** A set is initialized to store pairs of triplets. This set will be used to hold the positive (original) triplets and their corresponding negative (corrupted) triplets.

**Negative triplet generation:** For each triplet (h, l, t) in Sbatch, a corrupted triplet (h', l, t') is sampled. The corruption process involves randomly replacing either the head entity h or the tail entity t with another entity from the entity set E. This helps create negative examples for training.

**Training pair generation:**

**Tbatch:** The set Thatch is updated by adding pairs of original and corrupted triplets, forming training pairs for the update step.

**Embedding update:** The embeddings (€ and e) are updated based on the training pairs Tbatch. The specific update rule or algorithm for updating the embeddings is not mentioned in the provided information.

The loop continues until convergence or a specified number of iterations.

**Key concepts in the algorithm**

**1. Training set S:** It is a set of triplets (h, l, t), where h represents the head entity, l represents the relation, and t represents the tail entity. These triplets serve as the training data for the algorithm.

**2. Entities set E:** It is a set that contains all the entities present in the training set S. Entities can be any real-world objects or concepts**.**

1. **Relations set L:** It is a set that contains all the relations present in the training set S. Relations represent the semantic connections between entities.
2. **Margin y:** It is a hyperparameter that defines the minimum distance that the algorithm aims to achieve between positive and negative triplets. It helps in creating a clear separation between correct and incorrect predictions.

**5. Embeddings dimension k:** It is a hyperparameter that determines the size of the embedding vectors generated for entities and relations. The embeddings capture the semantic representation of entities and relations in a continuous vector space.

**6. Initialize embeddings € for each relation l in the relation set L:** It refers to initializing the embedding vectors for each relation in the relation set L. The embeddings are typically initialized with random values from a uniform distribution.

**7. Initialize embeddings e for each entity e in the entity set E:** It refers to initializing the embedding vectors for each entity in the entity set E. Similar to relation embeddings, entity embeddings are usually initialized with random values from a uniform distribution.

1. **Sbatch:** It represents a minibatch of size b, which is a subset of the training set S. During each iteration, a minibatch is sampled from the training set to train the model in a more efficient manner.

**9. Adjust Learning Rate:** Indicates the adjustment of the learning rate using the AdaGrad algorithm to efficiently optimize the embeddings.

**10. Sample a corrupted triplet:** It involves randomly replacing either the head entity h or the tail entity t of a given triplet with another entity from the entity set E. This process creates a corrupted triplet, which serves as a negative example for training.

**11. Tbatch:** It represents the set of pairs of triplets, where each pair consists of an original triplet and its corresponding corrupted triplet. These pairs are used for training the model.

1. **Update embeddings:** It refers to the process of updating the embedding vectors of entities and relations based on the training pairs in Thatch. The specific update rule or algorithm for updating the embeddings is not provided in the given algorithm and would need to be implemented separately.

**3.2 Flow of the project**

Input Entities and Relations

Start

Creating Triples

Mapping triples with Id’s

Creating dictionaries for Entities and relations

Model Training

Evaluate on Test set

Hyperparameters

Creating Negative Samples

Calculation Score

Calculating Loss

Update Embeddings

Performance Evaluation of Validation set

Selecting Optimized model

Measure performance & Results

End

## 

## **Concepts in flowchart**

**Input Entities and relations :** Set of entities (E) and their corresponding relations (R) are provided as input to the algorithm.

**Creating Triples :** Creating triplets with entities and relations in the format(head, relation, tail).

**Mapping Triples with Id’s :** Assigning unique identifiers to entities and relationships for efficient representation.  
 **Creating Dictionaries for Entities and Relations :** Assigning unique numerical identifiers to entities and relations for efficient storage, retrieval, and representation in knowledge graphs.

**Model Training :** Refers to the training phase of the model. It involves initializing the hyperparameters (such as rank, regularization parameter, learning rate, and the number of negatives per positive sample), setting the maximum number of epochs, and implementing early stopping criteria to prevent overfitting. The training process involves repeatedly generating negative samples, computing the scoring function, updating embeddings using stochastic gradient descent and backpropagation, adjusting the learning rate, and evaluating the performance on the validation set to track the best model.

**Hyperparameters :** Adjustable settings that control the behavior and performance of a model, such as learning rate, batch size, and regularization.

**Creating Negative Samples :** Generating corrupted triplets by replacing entities to create examples of false relationships for training contrastive learning.

**Scoring Function :** Refers to the calculation of the scoring function using

the complex embeddings and scoring function equations. The scoring function measures the compatibility or likelihood of a triple (subject, relation, object) being true.

**Calcuting Loss :** Quantifying the discrepancy between predicted and actual values, guiding the optimization process by minimizing the error during training.

**Update Embeddings :** Represents the update step where the embeddings (entity and relation

embeddings) are updated using stochastic gradient descent and backpropagation. This step aims to

minimize the negative log-likelihood loss function with L2 regularization to optimize the model

parameters.

**Performance Evaluation of Validation set :** Represents the evaluation of the model's performance on the validation set. This involves measuring how well the model ranks subject and object substitutions for each test triple and computing evaluation metrics such as Mean Reciprocal Rank (MRR) and Hits at ‘m’. The performance on the validation set helps in selecting the best model for testing.

**Selecting best model :** Indicates the process of tracking the best-performing model based on the evaluation results on the validation set. This helps ensure that the final model selected for testing is the one that exhibits the highest performance on the validation set.

**Evaluate on Test set :** Represents the evaluation of the selected model on the test set. The model's performance on the test set is measured using metrics such as computing the probabilities of entries being true or false for targeted unobserved triples.

**Measure performance :** Refers to the calculation of performance metrics such as Mean

Reciprocal Rank (MRR) and Hits at m. These metrics provide an assessment of how well the model ranks and predicts true triples compared to false ones.

**Results :** Represents the output of the evaluation results, which may include the performance metrics obtained during the testing phase.

**3.3 Methodology**

### 

TransE, as originally formulated, is primarily designed for modeling single-relation data. However, it can still be applied to multi-relational data by considering each relation as a separate type of interaction between entities. To adapt TransE for multi-relational data, one common approach is to introduce a distinct relation embedding for each relation. This allows the model to capture the unique characteristics and semantics associated with different relations.

The **Scoring function** in TransE for multi-relational data can be defined as:

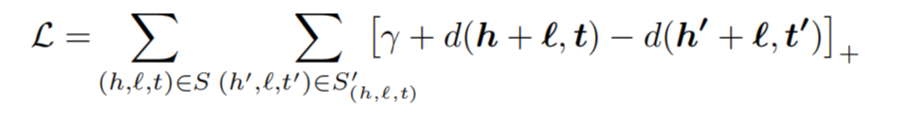
f(h, r, t) = ||h + r - t||

h is the embedding of the head entity.

r is the embedding of the relation.

t is the embedding of the tail entity.

The objective of TransE remains the same, which is to minimize the distance between the embeddings of the head entity, relation, and tail entity in the latent space. By treating each relation as a separate entity and incorporating relation embeddings, TransE can effectively model multi-relational data. It allows the model to capture the interactions and semantic relationships between entities under different relations, enhancing its capability for tasks such as link prediction, entity classification, or knowledge graph completion in multi-relational datasets.



### Modeling Relations

These embedding vectors capture the semantic meaning of the relation in a continuous vector space.TransE, relations are modeled as translation vectors in the embedding space. This translation-based modeling assumes that the relationship between entities can be represented as translations or shifts in the embedding space

**Relation Embeddings:**Each relation in the knowledge graph is associated with an embedding vector.These embedding vectors capture the semantic meaning of the relation in a continuous vector space.

**Translation-Based Assumption:**TransE assumes that the effect of a relation on an entity can be represented as a translation from the entity's embedding to the embedding of the related entity.In other words, the relation vector can be seen as the translation required to move from the embedding of the head entity to the embedding of the tail entity.

**Embedding Alignment:**The goal of TransE is to align the embeddings of entities and relations such that the translations between them correspond to the observed relationships in the knowledge graph.By aligning the embeddings, TransE aims to ensure that the embeddings of entities and their associated relations are compatible and consistent.

**Scoring Function:**To measure the compatibility or plausibility of a triplet (h, l, t), TransE uses a scoring function based on the distance or similarity between the embeddings of the head entity, the relation, and the tail entity.

The scoring function typically involves computing the distance between the sum of the embeddings of the head entity and the relation, and the embedding of the tail entity.

### 3.4. TransE Applications

**1. Knowledge Base Completion:** Link prediction is widely used to complete and refine knowledge bases. By this, knowledge bases can be populated with more comprehensive and accurate information which enhances the knowledge base's usability in various applications.

**2. Recommender Systems:** Link prediction plays a crucial role in improving the accuracy and effectiveness of recommender systems. This enhances user satisfaction and engagement in recommendation-based platforms, such as content streaming, or social media platforms.

**3. Social Network Analysis:** Link prediction helps in understanding and analysing social networks by predicting future links between individuals or entities. It aids in identifying potential friendships, collaborations, influence patterns, or communities within the network.

**4. Biomedical Research:** Link prediction has significant applications in biomedical research, particularly in biological networks and drug discovery. By predicting protein-protein interactions, gene-disease associations, or drug-target interactions, researchers can gain insights into complex biological processes, identify potential therapeutic targets, and accelerate drug discovery and development.

**5. Semantic Search and Question Answering**: Link prediction enables semantic search and improves question answering systems. This enhances the performance of search engines, virtual assistants, and natural language processing applications.

**6. Network Security and Fraud Detection:** Link prediction is applied in network security to detect and prevent fraudulent activities. By predicting potential links or relationships between entities involved in fraudulent behaviour, patterns and anomalies can be identified, enabling timely intervention and fraud prevention in financial transactions, cybersecurity, and online platforms.

**7. Entity Resolution:** Link prediction can aid in entity resolution, which involves identifying and merging duplicate or similar entities across multiple data sources. By predicting links between similar entities, the process of entity resolution becomes more accurate and efficient, improving data quality and integration.

**8. Social Influence Analysis:** Link prediction can assist in social influence analysis by predicting influential nodes or individuals in a social network. By identifying nodes that are likely to have a significant impact on the spread of information or behaviours, targeted strategies can be developed for marketing, opinion shaping, or social campaigns.

## **Examples**

* E-commerce
* Medical concepts
* Fraud detection systems
* Social interactions in a social network
* AI systems and Chatbots

# **CHAPTER 4**

# **IMPLEMENTATION**

**4.1 : Importing the module**

To run the files we need to import some python modules like below

from \_\_future\_\_ import absolute\_import

from \_\_future\_\_ import division

from \_\_future\_\_ import print\_function

import argparse

import json

import logging

import os

import random

import numpy as np

import torch

from torch.utils.data import DataLoader

from model import KGEModel

from dataloader import TrainDataset

from dataloader import BidirectionalOneShotIterator

**• Json :** Module for encoding and decoding Json data in python, lightweight human- readable data exchange.

**• Argparse:** ‘Argparse’ is a Python module that provides a convenient way to parse command-line arguments and generate user-friendly help messages and error handling, facilitating the creation of command-line interfaces for knowledge graph embedding programs. It simplifies the process of defining and accessing command-line arguments.

**• logging :** Logging can be used to track and record important information during the training and evaluation process.

**• Os:** Os module useful in knowledge graph embedding performing operations system related operations, such as creating dictionaries, reading files, or accessing system information providing functionalities that assist in managing and organinzing data.

**• Random:** Random module can be utilized in knowledge graph embedding for tasks, such as initializing embedding vectors with random values, creating random negative samples for training.

**• Numpy :** Numpy is widely used in knowledge graph embedding for efficient handling of large scale matrixes operations, such as embedding manipulation and loss function optimization.

**• Dataloader :** A dataloader is a component used in machine learning frameworks, such as PyTorch, to load and iterate over a dataset during training or evaluation. It provides an interface to efficiently retrieve and batch data samples from the dataset.

**4.2 Data Preparation**

Obtain a knowledge graph dataset in the form of triples (head entity, relation, tail entity). Preprocess the dataset to remove any noisy or inconsistent data. Create a set of all entities and relations present in the dataset. Embedding Model Selection: Choose a suitable embedding model for your task. Some popular models include TransE, TransR, ComplEx, and DistMult. Each model has its own assumptions and characteristics, so we choose TransE model.

from torch.utils.data import Dataset

class TrainDataset(Dataset):

    def \_\_init\_\_(self, triples, nentity, nrelation, negative\_sample\_size, mode):

        self.len = len(triples)

        self.triples = triples

        self.triple\_set = set(triples)

        self.nentity = nentity

        self.nrelation = nrelation

        self.negative\_sample\_size = negative\_sample\_size

        self.mode = mode

        self.count = self.count\_frequency(triples)

        self.true\_head, self.true\_tail = self.get\_true\_head\_and\_tail(self.triple)

    @staticmethod

    def count\_frequency(triples, start=4):

        '''

        Get frequency of a partial triple like (head, relation) or (relation, tail)

        The frequency will be used for subsampling like word2vec

        '''

        count = {}

        for head, relation, tail in triples:

            if (head, relation) not in count:

                count[(head, relation)] = start

            else:

                count[(head, relation)] += 1

            if (tail, -relation-1) not in count:

                count[(tail, -relation-1)] = start

            else:

                count[(tail, -relation-1)] += 1

        return count

    @staticmethod

    def get\_true\_head\_and\_tail(triples):

        '''

        Build a dictionary of true triples that will

        be used to filter these true triples for negative sampling

        '''

        true\_head = {}

        true\_tail = {}

        for head, relation, tail in triples:

            if (head, relation) not in true\_tail:

                true\_tail[(head, relation)] = []

            true\_tail[(head, relation)].append(tail)

            if (relation, tail) not in true\_head:

                true\_head[(relation, tail)] = []

            true\_head[(relation, tail)].append(head)

        for relation, tail in true\_head:

            true\_head[(relation, tail)] = np.array(list(set(true\_head[(relation, tail)])))

        for head, relation in true\_tail:

            true\_tail[(head, relation)] = np.array(list(set(true\_tail[(head, relation)])))

        return true\_head, true\_tail

class TestDataset(Dataset):

    def \_\_init\_\_(self, triples, all\_true\_triples, nentity, nrelation, mode):

        self.len = len(triples)

        self.triple\_set = set(all\_true\_triples)

        self.triples = triples

        self.nentity = nentity

        self.nrelation = nrelation

        self.mode = mode

    def \_\_len\_\_(self):

        return self.len

    def \_\_getitem\_\_(self, idx):

        head, relation, tail = self.triples[idx]

        if self.mode == 'head-batch':

            tmp = [(0, rand\_head) if (rand\_head, relation, tail) not in self.triple\_set

                   else (-1, head) for rand\_head in range(self.nentity)]

            tmp[head] = (0, head)

        elif self.mode == 'tail-batch':

            tmp = [(0, rand\_tail) if (head, relation, rand\_tail) not in self.triple\_set

                   else (-1, tail) for rand\_tail in range(self.nentity)]

            tmp[tail] = (0, tail)

        else:

            raise ValueError('negative batch mode %s not supported' % self.mode)

        tmp = torch.LongTensor(tmp)

        filter\_bias = tmp[:, 0].float()

        negative\_sample = tmp[:, 1]

**Main terms in the codes**

**Training Dataset**

The training dataset in knowledge graph embedding represents the data used to train the embedding model. It consists of positive triples, which are true statements or facts from the knowledge graph, and negative samples generated by corrupting the positive triples.

* **\_\_init\_\_ method :** The \_\_init\_\_ method to initialize the dataset with the triples and negative sample size.
* **\_\_getitem\_\_ method :** The \_\_getitem\_\_ method in the training phase of a KGE (Knowledge Graph Embedding) project is typically implemented within the TrainDataset class. It is responsible for returning a single training sample, which consists of a positive triple and a set of negative triples.
* **count\_frenquency :** The count\_frequency function in knowledge graph embedding is used to compute the frequency of occurrence for each partial triple, such as (head, relation) or (relation, tail), in the training dataset. It helps in subsampling by assigning higher weights to less frequent partial triples.
* **get\_true\_head\_and\_tail :** The get\_true\_head\_and\_tail function in knowledge graph embedding is used to build a dictionary of true triples that will be used for filtering negative samples during training. It helps ensure that negative samples generated for training do not include true triples.

**Test Dataset**

The test dataset in knowledge graph embedding is used to evaluate the performance of the embedding model on unseen data. It contains positive triples from the knowledge graph and generates negative samples for evaluation purposes.

* **\_\_init\_\_ method :** The \_\_init\_\_ method in the test dataset class initializes the test dataset object with the provided triples and all true triples.
* **\_\_len\_\_ method :** The \_\_len\_\_ method in the test dataset class returns the length of the test dataset, which corresponds to the number of test samples or triples.
* **\_\_getitem\_\_ method :** The \_\_getitem\_\_ method in the test dataset class is responsible for retrieving a specific test sample at the given index.

**Entity and Relation Indexing**

Assign a unique index to each entity and relation in the knowledge graph. This step allows you to refer to entities and relations by their numerical indices during training and inference.

**Negative Sampling :** Generate negative samples to create a balanced training set. Negative samples are corrupted triples where either the head or tail entity is replaced randomly. The number of negative samples generated per positive triple depends on the chosen sampling strategy.

**Model Training :** Initialize entity and relation embeddings as random vectors of a fixed dimensionality. Define the loss function that measures the compatibility between true and corrupted triples. Use gradient-based optimization methods (e.g., stochastic gradient descent) to update the embeddings and minimize the loss. Train the model by iteratively sampling positive and negative triples and updating the embeddings.

For this we need to write model.py file

der

from dataloader import TestDataset

class KGEModel(nn.Module):

    def \_\_init\_\_(self, model\_name, nentity, nrelation, hidden\_dim, gamma,

                 double\_entity\_embedding=False, double\_relation\_embedding=False):

        super(KGEModel, self).\_\_init\_\_()

        self.model\_name = model\_name

        self.nentity = nentity

        self.nrelation = nrelation

            torch.Tensor([gamma]),

            requires\_grad=False

        )

    @staticmethod

    def train\_step(model, optimizer, train\_iterator, args):

        '''

        A single train step. Apply back-propation and return the loss

        '''

        model.train()

        optimizer.zero\_grad()

        positive\_sample, negative\_sample, subsampling\_weight, mode = next(train\_iterator)

        if args.cuda:

            positive\_sample = positive\_sample.cuda()

            negative\_sample = negative\_sample.cuda()

            subsampling\_weight = subsampling\_weight.cuda()

        negative\_score = model((positive\_sample, negative\_sample), mode=mode)

                    'tail-batch'

                ),

                batch\_size=args.test\_batch\_size,

                num\_workers=max(1, args.cpu\_num//2),

                collate\_fn=TestDataset.collate\_fn

            )

            test\_dataset\_list = [test\_dataloader\_head, test\_dataloader\_tail]

            logs = []

            step = 0

            total\_steps = sum([len(dataset) for dataset in test\_dataset\_list])

            with torch.no\_grad():

                for test\_dataset in test\_dataset\_list:

                    for positive\_sample, negative\_sample, filter\_bias, mode in test\_dataset:

                        if args.cuda:

                            positive\_sample = positive\_sample.cuda()

                            negative\_sample = negative\_sample.cuda()

                            filter\_bias = filter\_bias.cuda()

                        batch\_size = positive\_sample.size(0)

                        score = model((positive\_sample, negative\_sample), mode)

                        score += filter\_bias

                        #Explicitly sort all the entities to ensure that there is no test exposure bias

                        argsort = torch.argsort(score, dim = 1, descending=True)

                        if mode == 'head-batch':

                            positive\_arg = positive\_sample[:, 0]

                        elif mode == 'tail-batch':

                            positive\_arg = positive\_sample[:, 2]

                        else:

                            raise ValueError('mode %s not supported' % mode)

                        for i in range(batch\_size):

                            #Notice that argsort is not ranking

                            ranking = (argsort[i, :] == positive\_arg[i]).nonzero()

                            assert ranking.size(0) == 1

                            #ranking + 1 is the true ranking used in evaluation metrics

                            ranking = 1 + ranking.item()

                            logs.append({

                                'MRR': 1.0/ranking,

                                'MR': float(ranking),

                                'HITS@1': 1.0 if ranking <= 1 else 0.0,

                                'HITS@3': 1.0 if ranking <= 3 else 0.0,

                                'HITS@10': 1.0 if ranking <= 10 else 0.0,

                            })

                        if step % args.test\_log\_steps == 0:

                            logging.info('Evaluating the model... (%d/%d)' % (step, total\_steps))

                        step += 1

            metrics = {}

            for metric in logs[0].keys():

                metrics[metric] = sum([log[metric] for log in logs])/len(logs)

        return metrics

**KGEModel function**

The KGEModel function represents a knowledge graph embedding model. It encapsulates the architecture and functionality of the embedding model used for learning entity and relation representations.

* **\_\_init\_\_ method :** The \_\_init\_\_ method initializes the KGEModel object with the number of entities (num\_entities), number of relations (num\_relations), and the dimensionality of the embeddings (embedding\_dim).
* **forward method :** The forward method performs the forward pass through the model, given positive and negative samples (positive\_sample and negative\_sample) and the training or evaluation mode (mode). It computes the loss based on the model's output and the desired training objective.

**train\_step function**

The train\_step function in knowledge graph embedding is a function that performs a single training step for the model using a batch of training samples. It typically includes the forward pass, loss computation, backward pass (gradient calculation), and optimization.

**4.3 Model Evaluation**

Assess the performance of the trained model on various tasks, such as link prediction, triple classification, or knowledge graph completion. Use evaluation such as KG model for training and evaluating the model.

To do this we write this KGE function in run.py file

kge\_model = KGEModel(

        model\_name=args.model,

        nentity=nentity,

        nrelation=nrelation,

        hidden\_dim=args.hidden\_dim,

        gamma=args.gamma,

        double\_entity\_embedding=args.double\_entity\_embedding,

        double\_relation\_embedding=args.double\_relation\_embedding

    )

    logging.info('Model Parameter Configuration:')

    for name, param in kge\_model.named\_parameters():

        logging.info('Parameter %s: %s, require\_grad = %s' % (name, str(param.size()), str(param.requires\_grad)))

    if args.cuda:

        kge\_model = kge\_model.cuda()

**Inference :** Once the model is trained, you can use the learned embeddings for downstream tasks. Apply the embeddings to perform tasks like entity classification, relation prediction, similarity search, or knowledge graph reasoning.

For interference we use GUI and to use that we need to install CUDA in the Systems and display the results.

**4.4 Model Executing**

Once the model is trained, you can use the learned embedding for downstream tasks. Apply the embeddings to perform tasks like entity classification, relation prediction, similarity search, or knowledge graph reasoning.

Here we need design the run.py to execute the model

**Running command :** We need to give arguments in the command line prompt to run the file

def parse\_args(args=None):

    parser = argparse.ArgumentParser(

        description='Training and Testing Knowledge Graph Embedding Models',

        usage='train.py [<args>] [-h | --help]'

    )

    parser.add\_argument('--cuda', action='store\_true', help='use GPU')

    parser.add\_argument('--do\_train', action='store\_true')

    parser.add\_argument('--do\_valid', action='store\_true')

    parser.add\_argument('--do\_test', action='store\_true')

    parser.add\_argument('--evaluate\_train', action='store\_true', help='Evaluate on training data')

    parser.add\_argument('--countries', action='store\_true', help='Use Countries S1/S2/S3 datasets')

    parser.add\_argument('--regions', type=int, nargs='+', default=None,

                        help='Region Id for Countries S1/S2/S3 datasets, DO NOT MANUALLY SET')

    parser.add\_argument('--data\_path', type=str, default=None)

    parser.add\_argument('--model', default='TransE', type=str)

    parser.add\_argument('-de', '--double\_entity\_embedding', action='store\_true')

    parser.add\_argument('-dr', '--double\_relation\_embedding', action='store\_true')

    parser.add\_argument('-n', '--negative\_sample\_size', default=128, type=int)

    parser.add\_argument('-d', '--hidden\_dim', default=500, type=int)

    parser.add\_argument('-g', '--gamma', default=12.0, type=float)

    parser.add\_argument('-adv', '--negative\_adversarial\_sampling', action='store\_true')

    parser.add\_argument('-a', '--adversarial\_temperature', default=1.0, type=float)

    parser.add\_argument('-b', '--batch\_size', default=1024, type=int)

    parser.add\_argument('-r', '--regularization', default=0.0, type=float)

    parser.add\_argument('--test\_batch\_size', default=4, type=int, help='valid/test batch size')

    parser.add\_argument('--uni\_weight', action='store\_true',

                        help='Otherwise use subsampling weighting like in word2vec')

    parser.add\_argument('-lr', '--learning\_rate', default=0.0001, type=float)

    parser.add\_argument('-cpu', '--cpu\_num', default=10, type=int)

    parser.add\_argument('-init', '--init\_checkpoint', default=None, type=str)

    parser.add\_argument('-save', '--save\_path', default=None, type=str)

    parser.add\_argument('--max\_steps', default=100000, type=int)

    parser.add\_argument('--warm\_up\_steps', default=None, type=int)

    parser.add\_argument('--save\_checkpoint\_steps', default=10000, type=int)

    parser.add\_argument('--valid\_steps', default=10000, type=int)

    parser.add\_argument('--log\_steps', default=100, type=int, help='train log every xx steps')

    parser.add\_argument('--test\_log\_steps', default=1000, type=int, help='valid/test log every xx steps')

    parser.add\_argument('--nentity', type=int, default=0, help='DO NOT MANUALLY SET')

    parser.add\_argument('--nrelation', type=int, default=0, help='DO NOT MANUALLY SET')

    return parser.parse\_args(args)

It will take input from the command we give in CLI.

We take the different input command lines for different data sets on the below arguments collection.

# Best Configuration for TransE

#

bash run.sh train TransE FB15k 0 0 1024 256 1000 24.0 1.0 0.0001 150000 16

bash run.sh train TransE FB15k-237 0 0 1024 256 1000 9.0 1.0 0.00005 100000 16

bash run.sh train TransE wn18 0 0 512 1024 500 12.0 0.5 0.0001 80000 8

bash run.sh train TransE wn18rr 0 0 512 1024 500 6.0 0.5 0.00005 80000 8

bash run.sh train TransE countries\_S1 0 0 512 64 1000 0.1 1.0 0.000002 40000 8 --countries

bash run.sh train TransE countries\_S2 0 0 512 64 1000 0.1 1.0 0.000002 40000 8 --countries

bash run.sh train TransE countries\_S3 0 0 512 64 1000 0.1 1.0 0.000002 40000 8 --countries

#

Here we need to give commands to the command prompt to run the run.py and we have different types datasets so we need to use different commands for different datasets as below.

**Run.py**

def parse\_args(args=None):

    parser = argparse.ArgumentParser(

        description='Training and Testing Knowledge Graph Embedding Models',

        usage='train.py [<args>] [-h | --help]'

    )

    parser.add\_argument('--cuda', action='store\_true', help='use GPU')

    parser.add\_argument('--do\_train', action='store\_true')

    parser.add\_argument('--do\_valid', action='store\_true')

    parser.add\_argument('--do\_test', action='store\_true')

    parser.add\_argument('--evaluate\_train', action='store\_true', help='Evaluate on training data')

    parser.add\_argument('--countries', action='store\_true', help='Use Countries S1/S2/S3 datasets')

    parser.add\_argument('--regions', type=int, nargs='+', default=None,

                        help='Region Id for Countries S1/S2/S3 datasets, DO NOT MANUALLY SET')

    parser.add\_argument('--data\_path', type=str, default=None)

    parser.add\_argument('--model', default='TransE', type=str)

    parser.add\_argument('-de', '--double\_entity\_embedding', action='store\_true')

    parser.add\_argument('-dr', '--double\_relation\_embedding', action='store\_true')

    parser.add\_argument('-n', '--negative\_sample\_size', default=128, type=int)

    parser.add\_argument('-d', '--hidden\_dim', default=500, type=int)

    parser.add\_argument('-g', '--gamma', default=12.0, type=float)

    parser.add\_argument('-adv', '--negative\_adversarial\_sampling', action='store\_true')

    parser.add\_argument('-a', '--adversarial\_temperature', default=1.0, type=float)

    parser.add\_argument('-b', '--batch\_size', default=1024, type=int)

    parser.add\_argument('-r', '--regularization', default=0.0, type=float)

    parser.add\_argument('--test\_batch\_size', default=4, type=int, help='valid/test batch size')

    parser.add\_argument('--uni\_weight', action='store\_true',

                        help='Otherwise use subsampling weighting like in word2vec')

    parser.add\_argument('-lr', '--learning\_rate', default=0.0001, type=float)

    parser.add\_argument('-cpu', '--cpu\_num', default=10, type=int)

    parser.add\_argument('-init', '--init\_checkpoint', default=None, type=str)

    parser.add\_argument('-save', '--save\_path', default=None, type=str)

    parser.add\_argument('--max\_steps', default=100000, type=int)

    parser.add\_argument('--warm\_up\_steps', default=None, type=int)

    parser.add\_argument('--save\_checkpoint\_steps', default=10000, type=int)

    parser.add\_argument('--valid\_steps', default=10000, type=int)

    parser.add\_argument('--log\_steps', default=100, type=int, help='train log every xx steps')

    parser.add\_argument('--test\_log\_steps', default=1000, type=int, help='valid/test log every xx steps')

    parser.add\_argument('--nentity', type=int, default=0, help='DO NOT MANUALLY SET')

    parser.add\_argument('--nrelation', type=int, default=0, help='DO NOT MANUALLY SET')

    return parser.parse\_args(args)

def override\_config(args):

    with open(os.path.join(args.init\_checkpoint, 'config.json'), 'r') as fjson:

        argparse\_dict = json.load(fjson)

    args.countries = argparse\_dict['countries']

    if args.data\_path is None:

        args.data\_path = argparse\_dict['data\_path']

    args.model = argparse\_dict['model']

    args.double\_entity\_embedding = argparse\_dict['double\_entity\_embedding']

    args.double\_relation\_embedding = argparse\_dict['double\_relation\_embedding']

    args.hidden\_dim = argparse\_dict['hidden\_dim']

    args.test\_batch\_size = argparse\_dict['test\_batch\_size']

def save\_model(model, optimizer, save\_variable\_list, args):

    argparse\_dict = vars(args)

    with open(os.path.join(args.save\_path, 'config.json'), 'w') as fjson:

        json.dump(argparse\_dict, fjson)

    torch.save({

        \*\*save\_variable\_list,

        'model\_state\_dict': model.state\_dict(),

        'optimizer\_state\_dict': optimizer.state\_dict()},

        os.path.join(args.save\_path, 'checkpoint')

    )

    entity\_embedding = model.entity\_embedding.detach().cpu().numpy()

    np.save(

        os.path.join(args.save\_path, 'entity\_embedding'),

        entity\_embedding

    )

    relation\_embedding = model.relation\_embedding.detach().cpu().numpy()

    np.save(

        os.path.join(args.save\_path, 'relation\_embedding'),

        relation\_embedding

)

def read\_triple(file\_path, entity2id, relation2id):

    '''

    Read triples and map them into ids.

    '''

    triples = []

    with open(file\_path) as fin:

        for line in fin:

            h, r, t = line.strip().split('\t')

            triples.append((entity2id[h], relation2id[r], entity2id[t]))

    return triples

def set\_logger(args):

    '''

    Write logs to checkpoint and console

    '''

    if args.do\_train:

        log\_file = os.path.join(args.save\_path or args.init\_checkpoint, 'train.log')

    else:

        log\_file = os.path.join(args.save\_path or args.init\_checkpoint, 'test.log')

    logging.basicConfig(

        format='%(asctime)s %(levelname)-8s %(message)s',

        level=logging.INFO,

        datefmt='%Y-%m-%d %H:%M:%S',

        filename=log\_file,

        filemode='w'

    )

    console = logging.StreamHandler()

    console.setLevel(logging.INFO)

    formatter = logging.Formatter('%(asctime)s %(levelname)-8s %(message)s')

    console.setFormatter(formatter)

    logging.getLogger('').addHandler(console)

def main(args):

if (not args.do\_train) and (not args.do\_valid) and (not args.do\_test):

raise ValueError('one of train/val/test mode must be choosed.')

if args.init\_checkpoint:

override\_config(args)

elif args.data\_path is None:

raise ValueError('one of init\_checkpoint/data\_path must be choosed.')

if args.do\_train and args.save\_path is None:

raise ValueError('Where do you want to save your trained model?')

if args.save\_path and not os.path.exists(args.save\_path):

os.makedirs(args.save\_path)

    # Write logs to checkpoint and console

    set\_logger(args)

    with open(os.path.join(args.data\_path, 'entities.dict')) as fin:

        entity2id = dict()

        for line in fin:

            eid, entity = line.strip().split('\t')

            entity2id[entity] = int(eid)

    with open(os.path.join(args.data\_path, 'relations.dict')) as fin:

        relation2id = dict()

        for line in fin:

            rid, relation = line.strip().split('\t')

            relation2id[relation] = int(rid)

    # Read regions for Countries S\* datasets

    if args.countries:

        regions = list()

        with open(os.path.join(args.data\_path, 'regions.list')) as fin:

            for line in fin:

                region = line.strip()

                regions.append(entity2id[region])

        args.regions = regions

    nentity = len(entity2id)

    nrelation = len(relation2id)

    args.nentity = nentity

    args.nrelation = nrelation

    logging.info('Model: %s' % args.model)

    logging.info('Data Path: %s' % args.data\_path)

    logging.info('#entity: %d' % nentity)

    logging.info('#relation: %d' % nrelation)

    train\_triples = read\_triple(os.path.join(args.data\_path, 'train.txt'), entity2id, relation2id)

    logging.info('#train: %d' % len(train\_triples))

    valid\_triples = read\_triple(os.path.join(args.data\_path, 'valid.txt'), entity2id, relation2id)

    logging.info('#valid: %d' % len(valid\_triples))

    test\_triples = read\_triple(os.path.join(args.data\_path, 'test.txt'), entity2id, relation2id)

    logging.info('#test: %d' % len(test\_triples))

    #All true triples

    all\_true\_triples = train\_triples + valid\_triples + test\_triples

    kge\_model = KGEModel(

        model\_name=args.model,

        nentity=nentity,

        nrelation=nrelation,

        hidden\_dim=args.hidden\_dim,

        gamma=args.gamma,

        double\_entity\_embedding=args.double\_entity\_embedding,

        double\_relation\_embedding=args.double\_relation\_embedding

    )

    logging.info('Model Parameter Configuration:')

    for name, param in kge\_model.named\_parameters():

        logging.info('Parameter %s: %s, require\_grad = %s' % (name, str(param.size()), str(param.requires\_grad)))

    if args.cuda:

        kge\_model = kge\_model.cuda()

    if args.do\_train:

        # Set training dataloader iterator

        train\_dataloader\_head = DataLoader(

            TrainDataset(train\_triples, nentity, nrelation, args.negative\_sample\_size, 'head-batch'),

            batch\_size=args.batch\_size,

            shuffle=True,

            num\_workers=max(1, args.cpu\_num//2),

            collate\_fn=TrainDataset.collate\_fn

        )

        train\_dataloader\_tail = DataLoader(

            TrainDataset(train\_triples, nentity, nrelation, args.negative\_sample\_size, 'tail-batch'),

            batch\_size=args.batch\_size,

            shuffle=True,

            num\_workers=max(1, args.cpu\_num//2),

            collate\_fn=TrainDataset.collate\_fn

        )

        train\_iterator = BidirectionalOneShotIterator(train\_dataloader\_head, train\_dataloader\_tail)

        # Set training configuration

        current\_learning\_rate = args.learning\_rate

        optimizer = torch.optim.Adam(

            filter(lambda p: p.requires\_grad, kge\_model.parameters()),

            lr=current\_learning\_rate

        )

        if args.warm\_up\_steps:

            warm\_up\_steps = args.warm\_up\_steps

        else:

            warm\_up\_steps = args.max\_steps // 2

    if args.init\_checkpoint:

        # Restore model from checkpoint directory

        logging.info('Loading checkpoint %s...' % args.init\_checkpoint)

        checkpoint = torch.load(os.path.join(args.init\_checkpoint, 'checkpoint'))

        init\_step = checkpoint['step']

        kge\_model.load\_state\_dict(checkpoint['model\_state\_dict'])

        if args.do\_train:

            current\_learning\_rate = checkpoint['current\_learning\_rate']

            warm\_up\_steps = checkpoint['warm\_up\_steps']

            optimizer.load\_state\_dict(checkpoint['optimizer\_state\_dict'])

    else:

        logging.info('Ramdomly Initializing %s Model...' % args.model)

        init\_step = 0

    step = init\_step

    logging.info('Start Training...')

    logging.info('init\_step = %d' % init\_step)

    logging.info('batch\_size = %d' % args.batch\_size)

    logging.info('negative\_adversarial\_sampling = %d' % args.negative\_adversarial\_sampling)

    logging.info('hidden\_dim = %d' % args.hidden\_dim)

    logging.info('gamma = %f' % args.gamma)

    logging.info('negative\_adversarial\_sampling = %s' % str(args.negative\_adversarial\_sampling))

    if args.negative\_adversarial\_sampling:

        logging.info('adversarial\_temperature = %f' % args.adversarial\_temperature)

    # Set valid dataloader as it would be evaluated during training

  if args.do\_train:

        logging.info('learning\_rate = %d' % current\_learning\_rate)

        training\_logs = []

        #Training Loop

        for step in range(init\_step, args.max\_steps):

            log = kge\_model.train\_step(kge\_model, optimizer, train\_iterator, args)

logs.append(log)

            if step >= warm\_up\_steps:

                current\_learning\_rate = current\_learning\_rate / 10

                logging.info('Change learning\_rate to %f at step %d' % (current\_learning\_rate, step))

                optimizer = torch.optim.Adam(

                    filter(lambda p: p.requires\_grad, kge\_model.parameters()),

                    lr=current\_learning\_rate

                )

                warm\_up\_steps = warm\_up\_steps \* 3

            if step % args.save\_checkpoint\_steps == 0:

                save\_variable\_list = {

                    'step': step,

                    'current\_learning\_rate': current\_learning\_rate,

                    'warm\_up\_steps': warm\_up\_steps

                }

                save\_model(kge\_model, optimizer, save\_variable\_list, args)

            if step % args.log\_steps == 0:

                metrics = {}

                for metric in training\_logs[0].keys():

                    metrics[metric] = sum([log[metric] for log in training\_logs])/len(training\_logs)

                log\_metrics('Training average', step, metrics)

                training\_logs = []

            if args.do\_valid and step % args.valid\_steps == 0:

                logging.info('Evaluating on Valid Dataset...')

                metrics = kge\_model.test\_step(kge\_model, valid\_triples, all\_true\_triples, args)

                log\_metrics('Valid', step, metrics)

        save\_variable\_list = {

            'step': step,

            'current\_learning\_rate': current\_learning\_rate,

            'warm\_up\_steps': warm\_up\_steps

        }

        save\_model(kge\_model, optimizer, save\_variable\_list, args)

    if args.do\_valid:

        logging.info('Evaluating on Valid Dataset...')

        metrics = kge\_model.test\_step(kge\_model, valid\_triples, all\_true\_triples, args)

        log\_metrics('Valid', step, metrics)

    if args.do\_test:

        logging.info('Evaluating on Test Dataset...')

        metrics = kge\_model.test\_step(kge\_model, test\_triples, all\_true\_triples, args)

        log\_metrics('Test', step, metrics)

    if args.evaluate\_train:

        logging.info('Evaluating on Training Dataset...')

        metrics = kge\_model.test\_step(kge\_model, train\_triples, all\_true\_triples, args)

        log\_metrics('Test', step, metrics)

if \_\_name\_\_ == '\_\_main\_\_':

    main(parse\_args())

**parser\_args funtion**

The parse\_args function is typically used to parse command-line arguments passed to a script using the argparse module.

* **argparse.ArgumentParser :** The argparse.ArgumentParser class is used to create an argument parser object, which handles the definition and parsing of command-line arguments.
* **parser.add\_argument :** The parser.add\_argument method is used to define command-line arguments. Each argument is specified by providing a name, type, default value, and help message.
* **parser.parse\_args :** The parser.parse\_args method is called to parse the command-line arguments and return a namespace object (args) containing the parsed argument values.

**override\_config function**

The override\_config function is a utility function that allows modifying the values of configuration parameters based on user-defined overrides. It takes a base configuration object and a dictionary of overrides as input and applies the overrides to the configuration.

**save\_model function**

The save\_model function in knowledge graph embedding is used to save the trained model parameters to disk for future use or deployment.

**read\_triple function**

The read\_triple function in a KGE (Knowledge Graph Embedding) project is typically used to read and load the triplets (i.e., triples) from a knowledge graph dataset.

**set\_logger function**

The set\_logger function in a KGE (Knowledge Graph Embedding) project is typically used to configure the logging settings and create a logger object.By calling the set\_logger function and providing a log file path, create a logger object with the desired logging settings. This logger can then be used throughout your KGE project to log messages, track progress, and record important events.

**main(args) function**

The main(args) function in a KGE (Knowledge Graph Embedding) project typically serves as the entry point of the script or application. It coordinates the different stages of the project, such as data loading, model initialization, training, evaluation, and saving the trained model. The main function takes an args object or dictionary as an argument, which contains the parsed command-line arguments or configuration settings.

* The \_\_name\_\_ = '\_\_main\_\_' construct in a KGE (Knowledge Graph Embedding) project is typically used to ensure that a specific block of code is executed only when the script is run directly, rather than being imported as a module by another script.
* The \_\_name\_\_ variable is a built-in variable in Python that represents the name of the current module. When a script is run directly as the main module, the value of \_\_name\_\_ is set to '\_\_main\_\_' .
* The \_\_name\_\_ == '\_\_main\_\_' condition checks if the script is being run directly. If the condition is true, the code inside the if block will be executed.

**4.5 Datasets:**

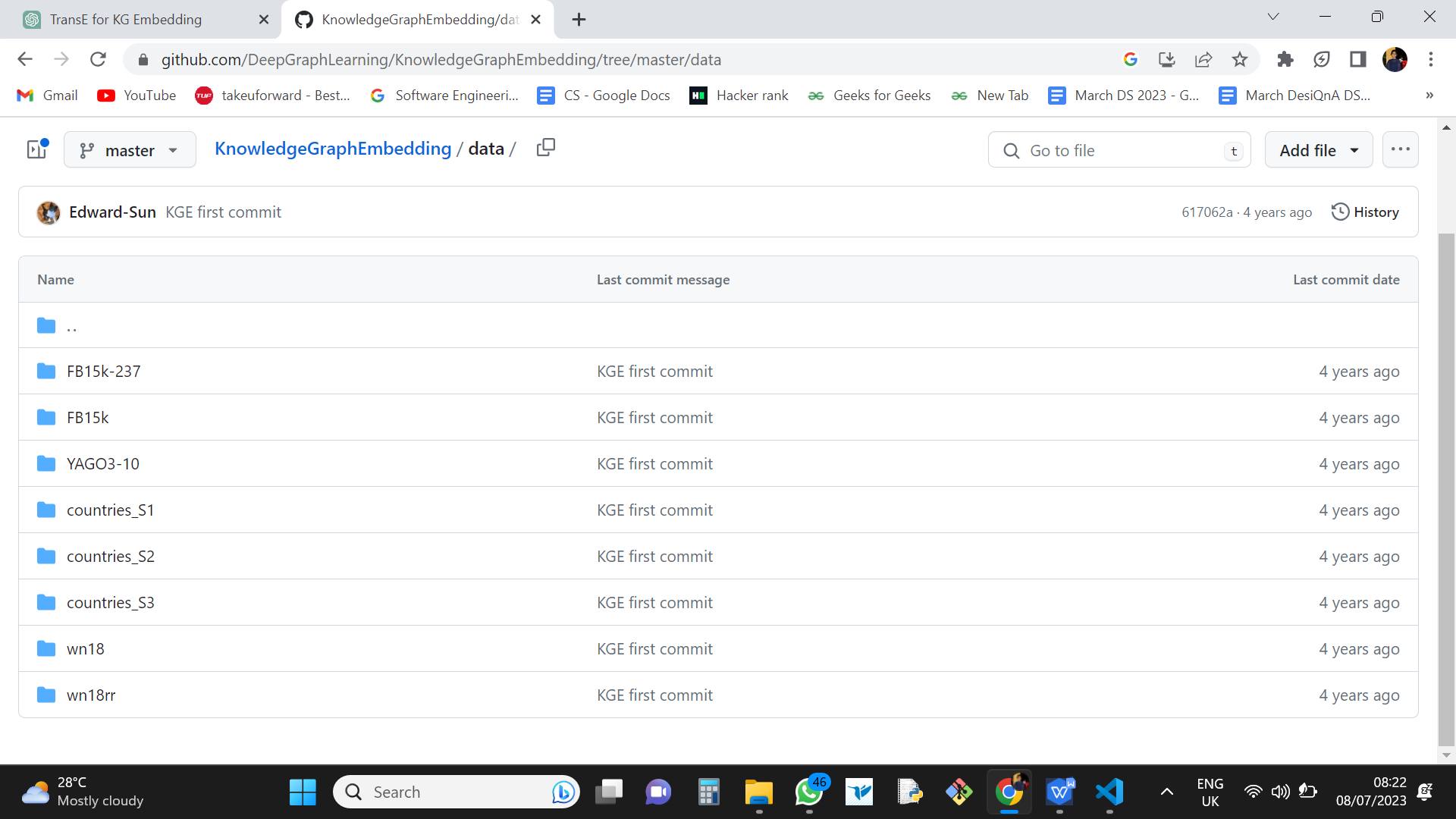


Figure 5 DATASETS

**4.6 Commands to execute in TransE**

**For wn18\_run.py:**

py codes/run.py --do\_train --cuda --do\_valid --do\_test --data\_path data/WN18 --model TransE -n 256 -b 1024 -d 1000 -g 24.0 -a 1.0 -adv -lr 0.0001 --max\_steps 150000 -save trails/fb15kTrail --test\_batch\_size 16 -de

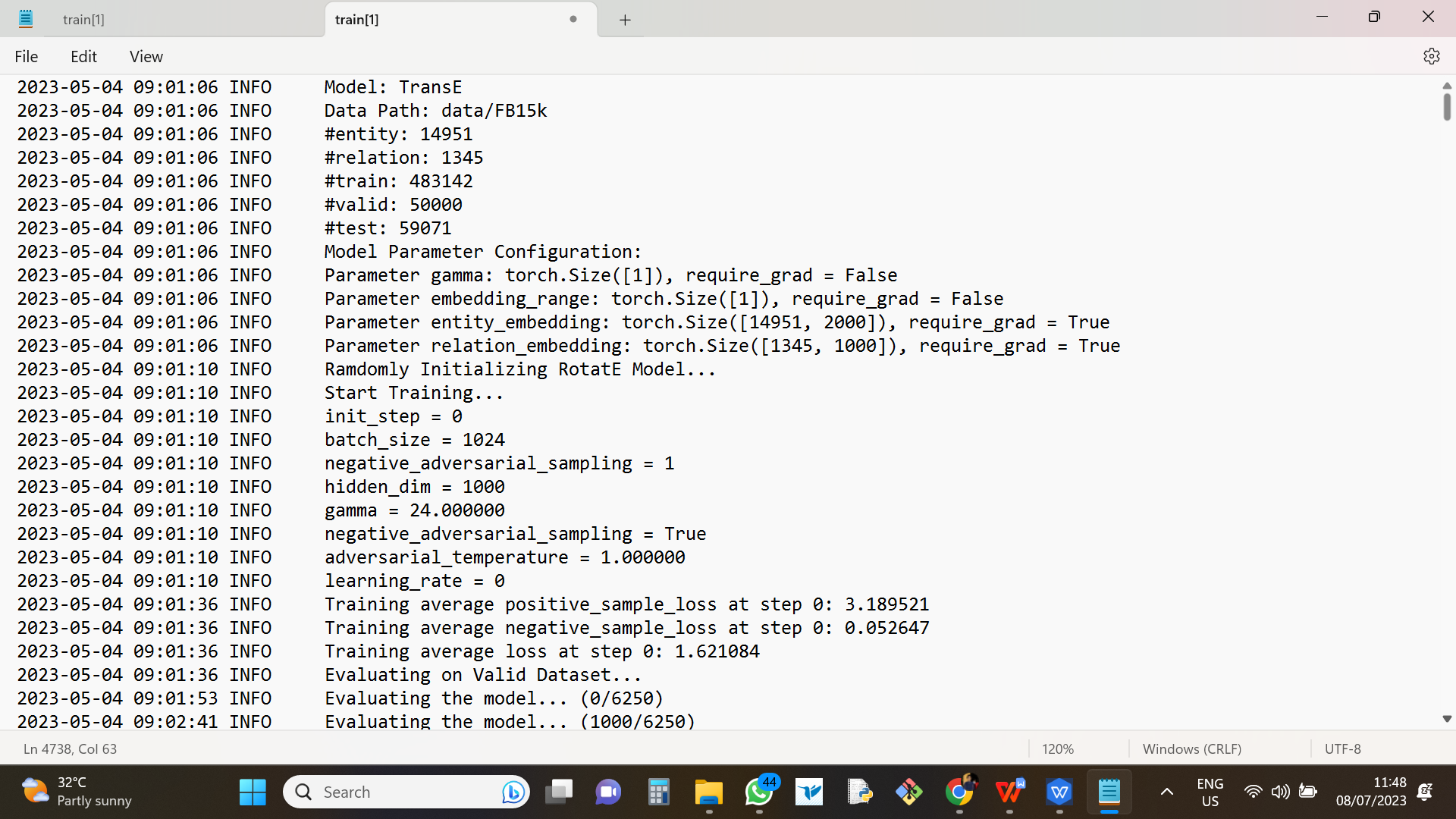
The given command is a Python script to train a TransE model on the WN18 dataset. It uses CUDA for GPU acceleration and performs training, validation, and testing. The model parameters include embedding size (256), batch size (1024), dimensionality (1000), margin (24.0), alpha (1.0), and learning rate (0.0001). The maximum number of training steps is set to 150,000, and the trained model is saved in the "trails/fb15kTrail" directory. The testing is performed with a batch size of 16, and the script is in German (-de).

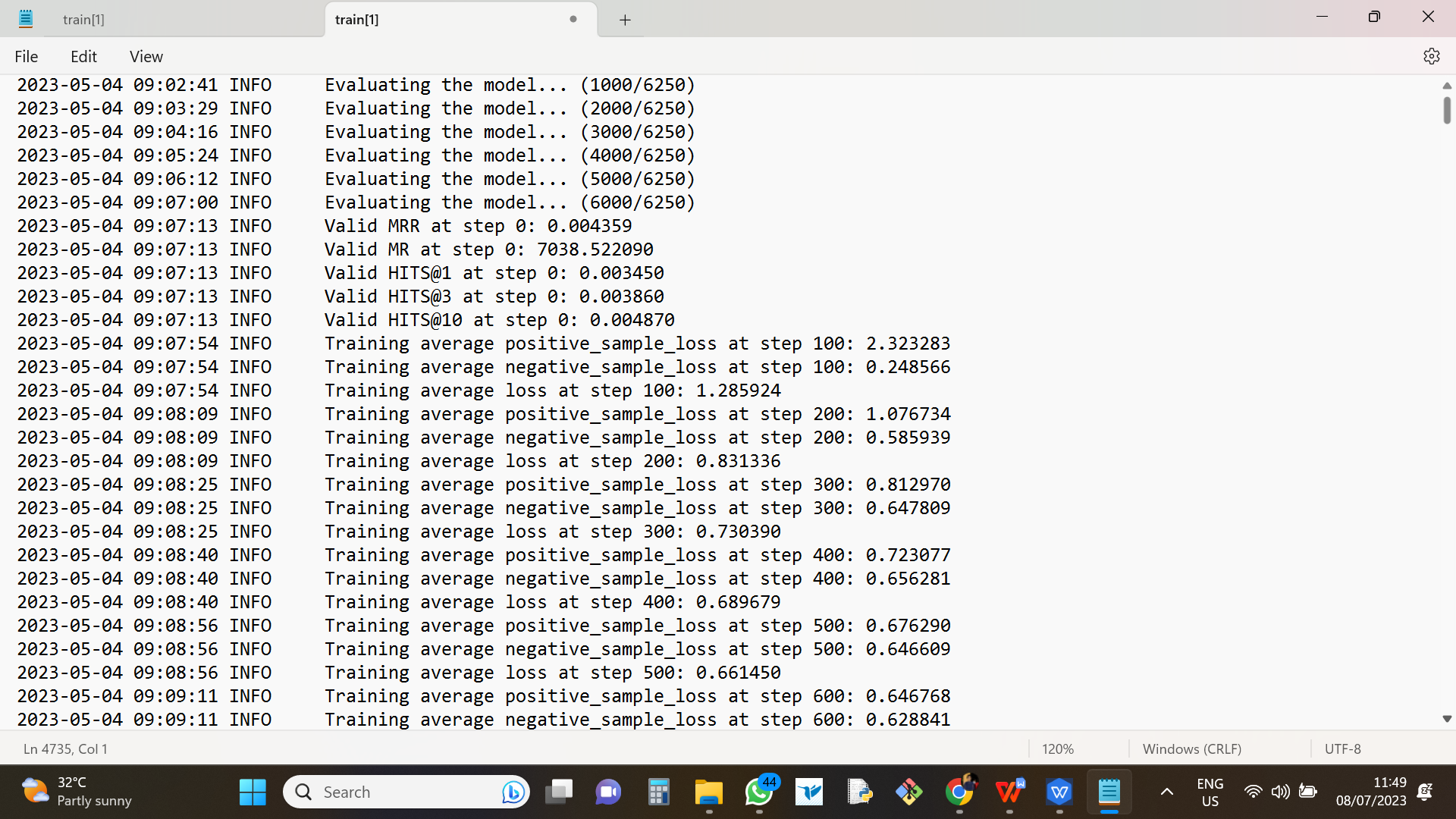
**For fb15k\_run.py:**

py codes/run.py --do\_train --cuda --do\_valid --do\_test --data\_path data/FB15k --model RotatE -n 256 -b 1024 -d 1000 -g 24.0 -a 1.0 -adv -lr 0.0001 --max\_steps 150000 -save trails/fb15kTrail --test\_batch\_size 16 -de

This command is a Python script for training a RotatE model on the FB15k dataset. It utilizes GPU acceleration through CUDA and performs training, validation, and testing. The model parameters include embedding size (256), batch size (1024), dimensionality (1000), margin (24.0), alpha (1.0), and learning rate (0.0001). The maximum number of training steps is set to 150,000, and the trained model is saved in the "trails/fb15kTrail" directory. Testing is conducted with a batch size of 16, and the script is in German (-de).

**4.7 Executing output**





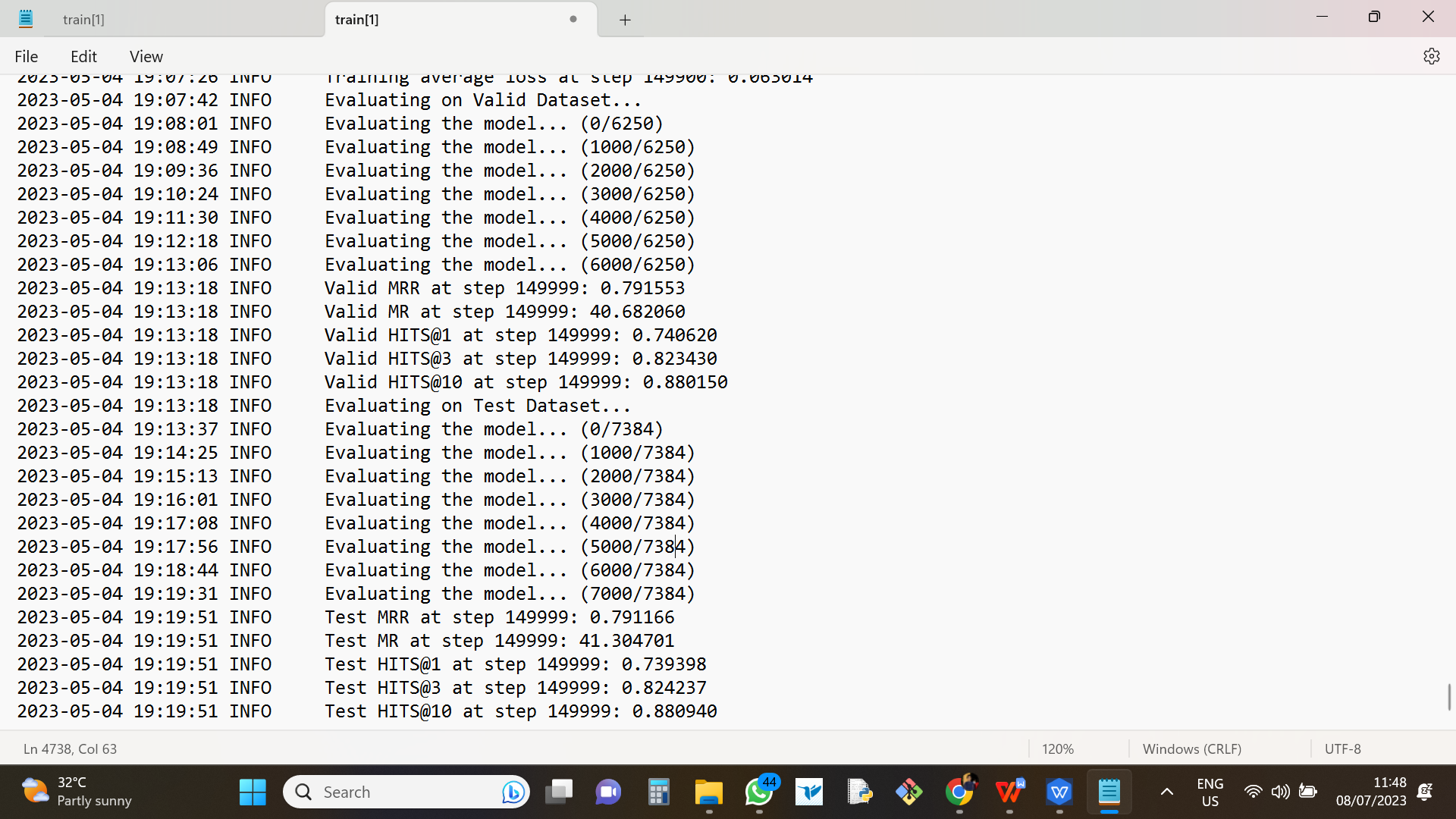


Figure 6 run.py output

The knowledge graph embedding model, RotatE, was trained on the FB15k dataset for knowledge graph completion. The dataset consisted of 14,951 entities and 1,345 relations, with 483,142 samples used for training and 50,000 samples for validation. The model's parameters included entity and relation embeddings. During training, the model was initialized randomly, and various hyperparameters such as batch size, negative adversarial sampling, hidden dimension, gamma, and learning rate were configured. The training progress was logged, displaying average positive and negative sample losses at each step. The model's performance was evaluated on the validation dataset, with periodic updates on the evaluation progress captured in the log snippet.

# **CHAPTER 5**

# **Results**

## **5.1 Comparitive Results on implementing TransE Model**

**FB15K DATASET**

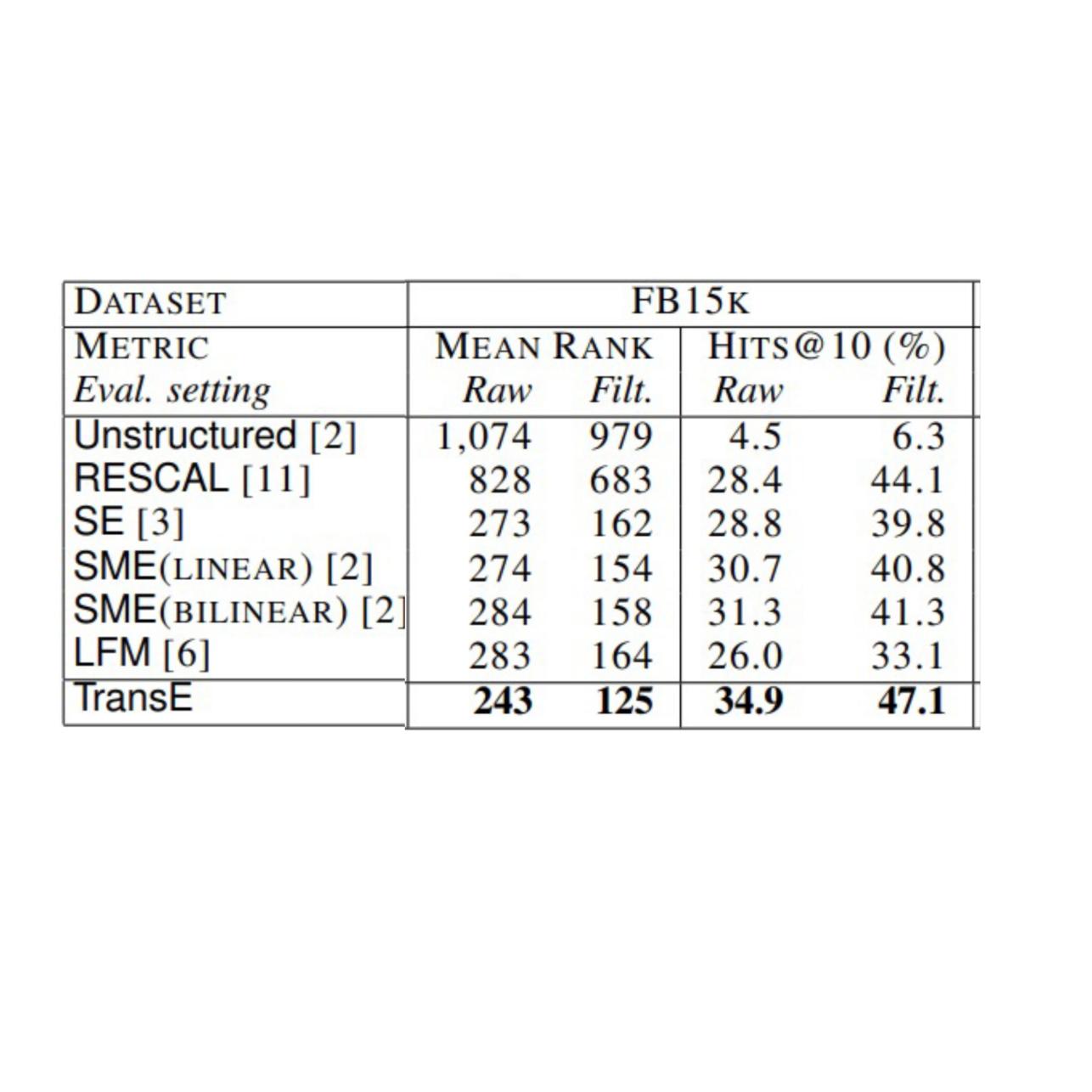


Figure 7 FB15K DATASET

**FB15K DATASET Example relations**

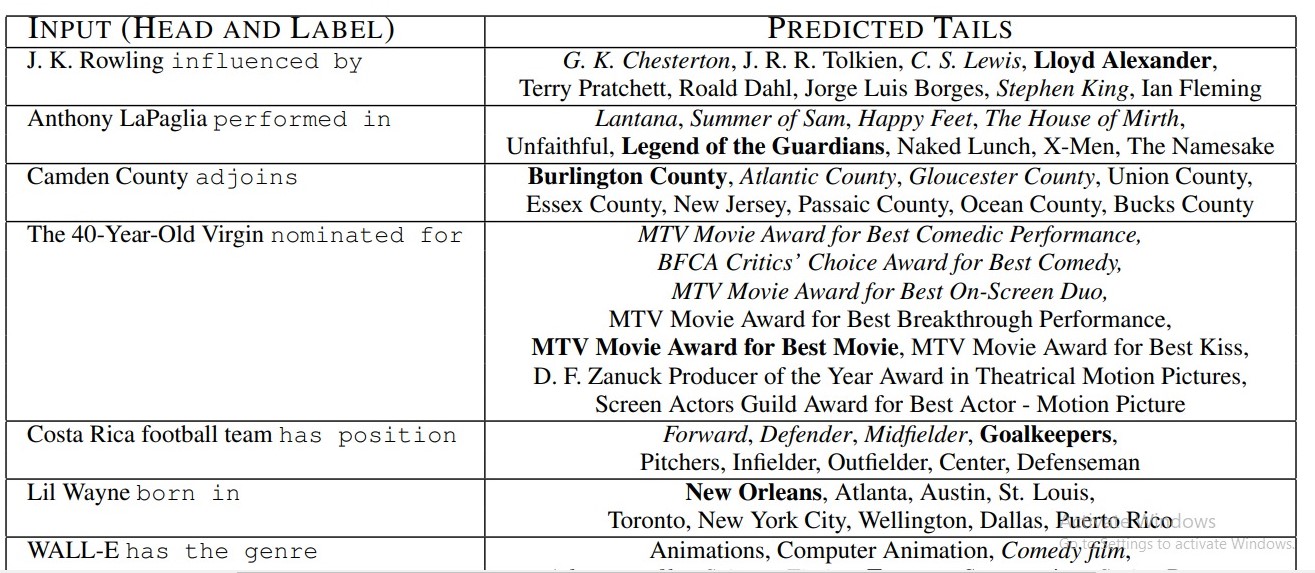


Figure 8 Prediction example

**FB15K DATASET Results**

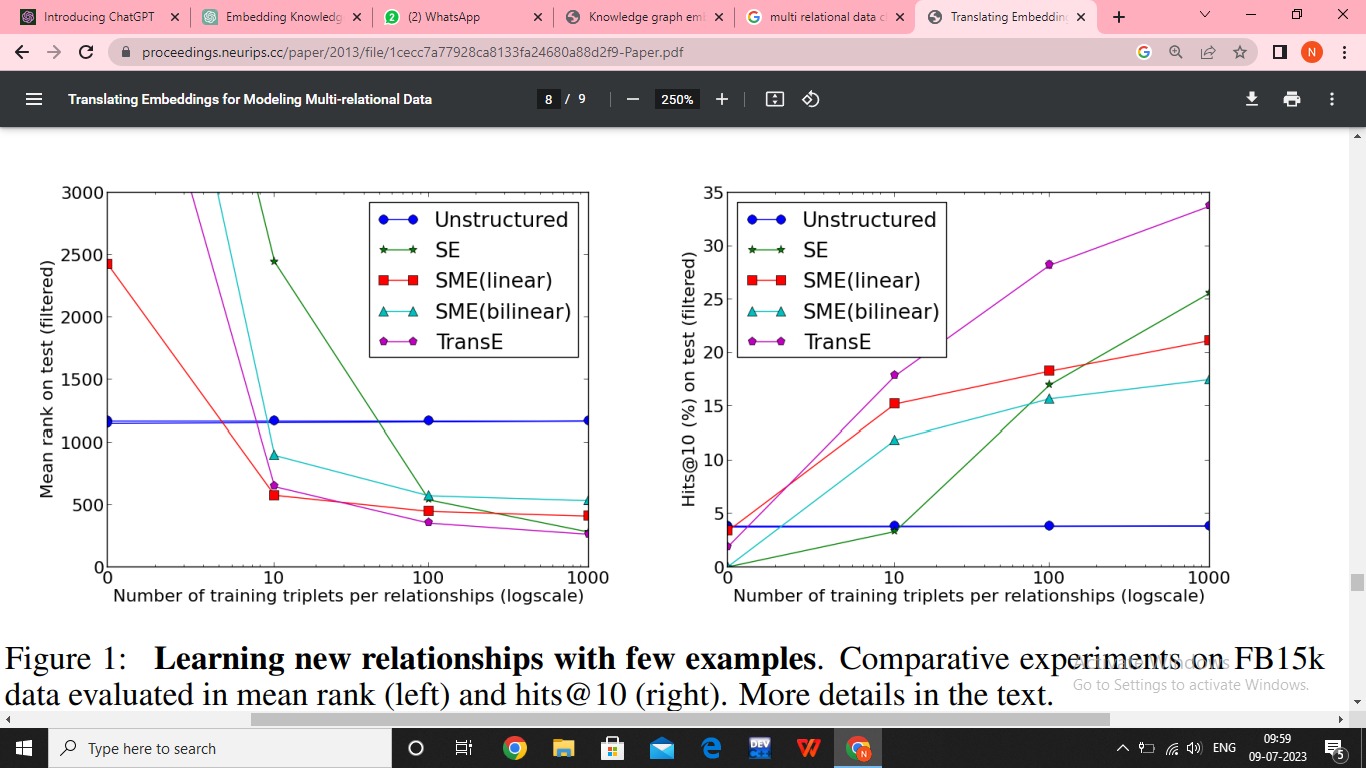


Figure 9 FB15K Result

**WN18 DATASET**

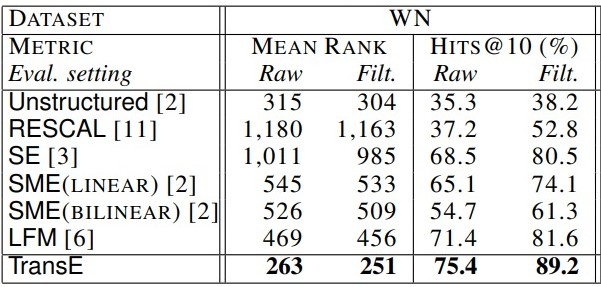
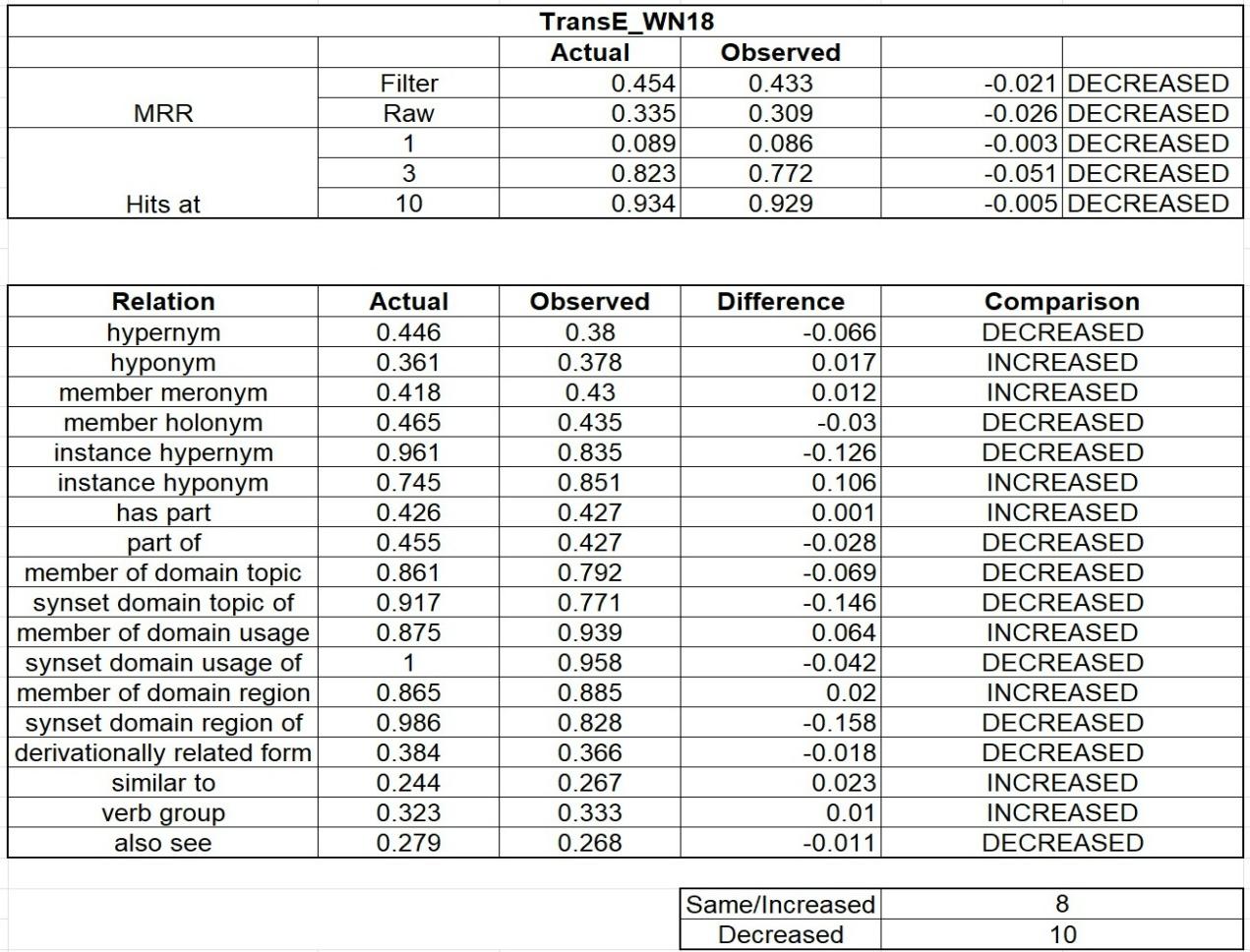


Figure 10 WN18 DATASET RESULTS

## **5.2 Asymmetric Relations in Different Models**

### 1.TransE\_WN18



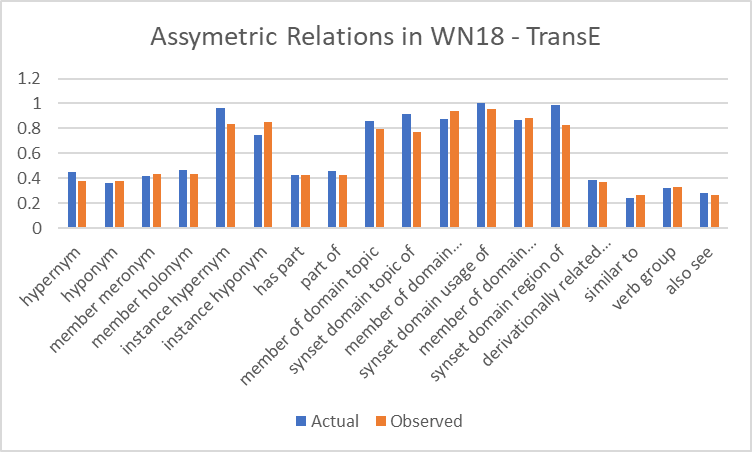
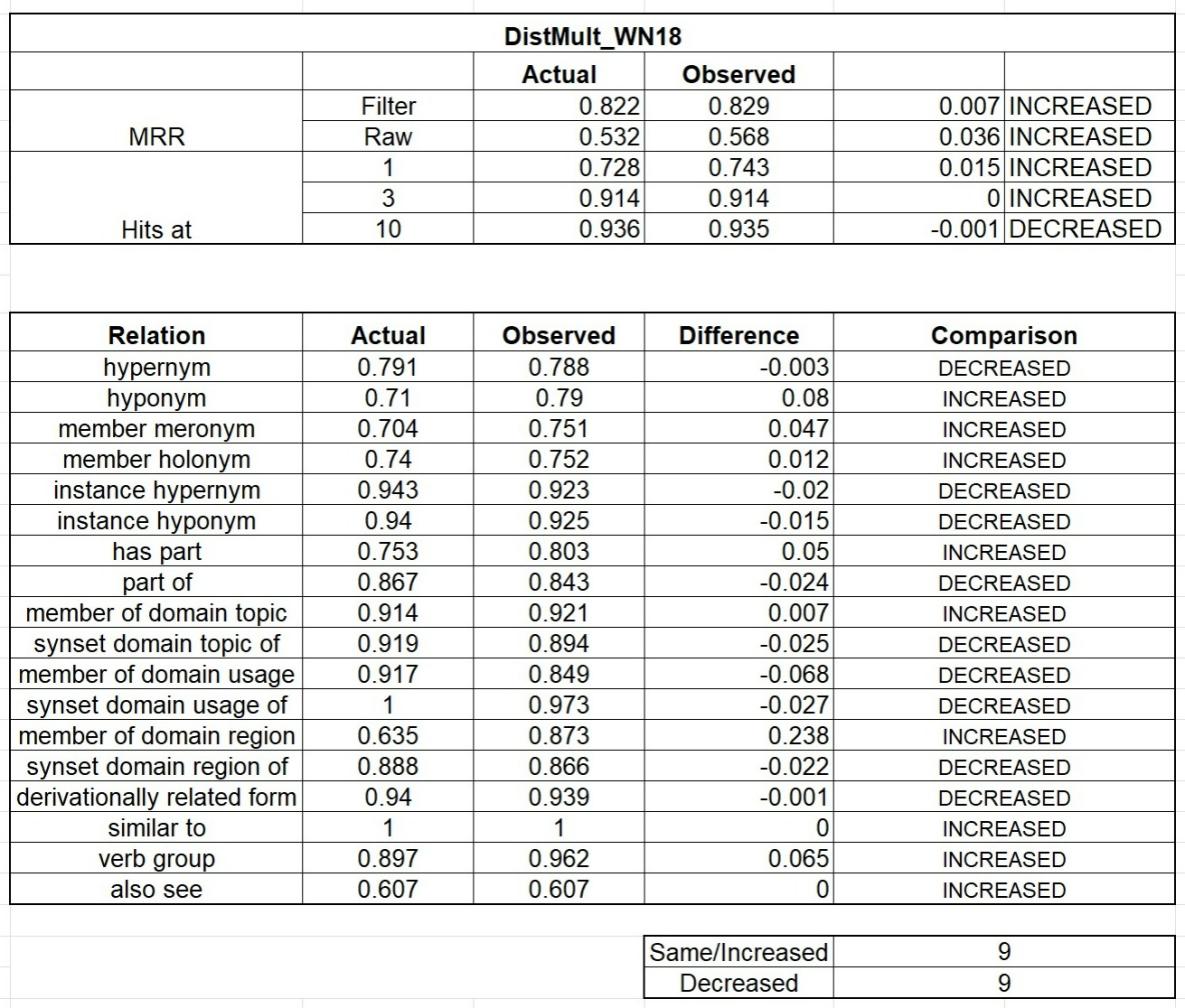


Figure 11 Asymmetric Relations in WN18 Dataset - TransE

### 2. DistMult\_WN18



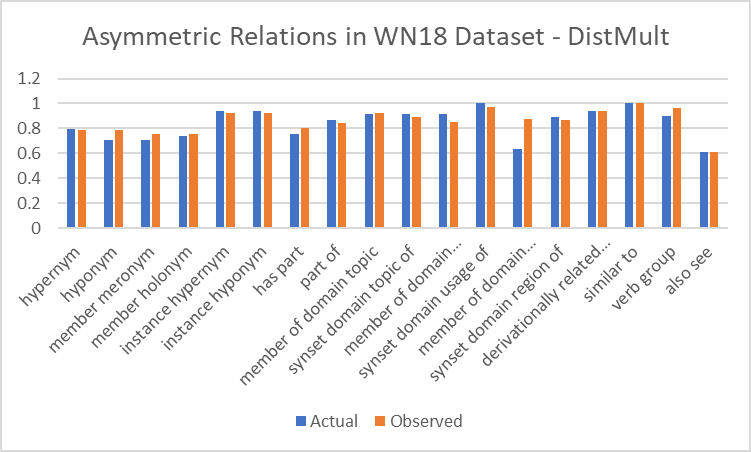


Figure 12 Asymmetric Relations in WN18 Dataset - DistMult

**Comparison of different models :**

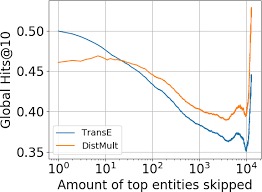


Figure 13Comparison of models

# **CHAPTER 6**

# **Conclusion & Future scope**

**Conclusion**

We proposed a new approach to learn embeddings of KBs, focusing on the minimal parametrization of the model to primarily represent hierarchical relationships. We showed that it works very well compared to competing methods on two different knowledge bases, and is also a highly scalable model, whereby we applied it to a very large-scale chunk of Freebase data. Although it remains unclear to us if all relationship types can be modeled adequately by our approach, by breaking down the evaluation into categories (1-to-1, 1-to-Many, . . . ) it appears to be performing well compared to other approaches across all settings.

# **Future Scope**

* **Enhanced Performance :** Improving TransE's effectiveness by addressing its limitations and exploring advanced embedding techniques.
* **Multimodal Knowledge Graphs:** Adapting TransE for knowledge graphs with diverse modalities, such as images, videos, or audio.
* **Real-world Applications:** Expanding TransE's usage in domains like recommender systems, question answering, and information retrieval.

# **CHAPTER 7**

# **References**

* https://proceedings.neurips.cc/paper/2013/file/1cecc7a77928ca8133fa24680a88d2f9-

Paper.pdf

* https://github.com/DeepGraphLearning/KnowledgeGraphEmbeddinghttps://data-

flair.training/blogs/python-project-driver-drowsiness-detection-system/

* https://towardsdatascience.com/embedding-models-for-knowledge-graph-completion-

a66d4c01d588https://www.geeksforgeeks.org/introduction-to-tensorflow/

* https://www.youtube.com/watch?v=YQCG408wgLw
* https://aws-dglke.readthedocs.io/en/latest/kg.html