DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis - Data Engineering and Analytics

Uncovering potential losses using knowledge graph embeddings and link prediction

Nishant Nigam

DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis - Data Engineering and Analytics

Uncovering potential losses using knowledge graph embeddings and link prediction

Aufdeckung potenzieller Verluste durch Verwendung von Wissensgraphen und Verknüpfungsvorhersagen

Author:	Nishant Nigam
Supervisor:	Prof. Dr. Christian Mendl
Advisors:	Dr. Felix Dietrich, Dr. Saahil Ognawala
Submission Date:	15.04.2022

I confirm that this master's thesis - data engineering and analytics is my own work and I have documented all sources and material used.

Munich, 15.04.2022 Nishant Nigam

Acknowledgments

I would like to express my sincere gratitude to my supervisor, Prof. Dr. Christian Mendl for giving me an opportunity to pursue my thesis work with the Chair of Scientific Computing in Computer Science at Technical University of Munich.

I am indebted to my advisor, Dr. Felix Dietrich, whose continuous support and guidance helped me extensively towards the accomplishment of this thesis. His insightful ideas about the project and constructive comments on the initial drafts of the document helped me organize this thesis in a much more systematic manner.

I am also grateful to my company advisor, Dr. Saahil Ognawala who continuously helped and assisted me throughout the development phase. His ideas and strategies helped me accomplish the complicated tasks by drilling them down to multiple easier tasks which lead to achieving the successful desired results.

Abstract

Knowledge graphs are a structured way of storing and providing logical representation of knowledge about the world. Most of the existing knowledge graphs contain knowledge that is incomplete, and only about a small subset of the entire knowledge about the world. In this thesis, we discuss the construction of company knowledge graph that contains data which deals with insured companies, their site locations and the losses encountered by them due to various causes like hurricanes and tornadoes. An existing knowledge graph from the company is the starting point. It contains information from various data sources, and has been configured by the company risk administrators. We, in this thesis, state that there exist relationships between the entities of the knowledge graph that are not represented by the graph, but can be predicted, either as unreported or potential links in the future. Here we, (a) implement link predictions between loss events and insured companies, using state-of-the-art graph embedding solutions, and (b) evaluate the effectiveness of link predictions using qualitative (with company underwriters) and quantitative (precision and recall) metrics. We use different types of embedding models to generate embeddings of the knowledge graph through extensive research of the existing models and perform various experiments using them on the existing knowledge graph. We give a comparative study between the models and then conclude by choosing the most optimal model that performs the best on our dataset. We first work with one link relationships and then extend our work for multiple links, i.e. with a more complex graph which has multiple link relationships. The performance of the current knowledge graph is encouraging for its use in the prediction of more links and nodes of the graph, which can help the company underwriters to take the knowledge graph towards completion.

Contents

1 Introduction

As a way to organize structured knowledge that represents the world and its entities, knowledge graphs are now emerging as a compelling way of abstraction of this knowledge using natural language processing and computer vision techniques. They are also playing an increasingly important role in representing the information extracted from multiple sources. The knowledge extracted from these multiple sources is then passed through different machine learning models to obtain good and knowledgeable predictions. With the help of knowledge graphs we can put the data in whatever context we need by linking and semantic metadata and the obtain a representation of data integration, unification, data analytics and sharing.

A knowledge graph is the representation of knowledge in the form of nodes and links. A basic knowledge graph is represented by a triple, a triple is a set of two nodes and a link between them. The nodes represent the knowledge contained in the database and can have information of different attributes like people, companies, computers etc. The link contains the information by which the two attributes are linked to each other. The links can contain the friendship relation between two people, network connections between two computers, customer relation between a company and people etc.

In the image below, we see that there are two nodes, A and C, and a relationship between them called B. The triple 'T' is the represented as $T = (A, C, B)$.

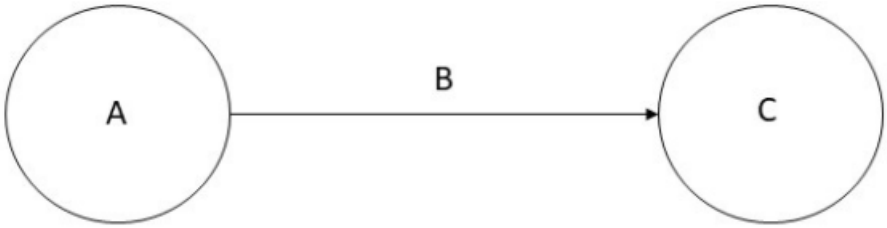

Figure 1.1: A simple graph

Several data management paradigms are incorporated into knowledge graphs like:

1. **Database:** Several structured queries can be used to access data from the knowledge graph.
2. **Graph:** As knowledge graph can be analysed similar to any other network structure.
3. **Knowledge base:** Contains different semantics that can be used to predict and conclude new facts from the existing knowledge base.

Let us now see some characteristics [Ont] of knowledge graphs that make them such a widely used data tool in recent times. They are:

1. **Performance**: The criticality of managing graphs of a huge amount of facts and properties have all been taken into consideration and have been proven efficient to use in various projects.
2. **Expressivity**: The standard semantic practices of graphs provides very nice representation of data and its knowledge using well structured data models, schema and vocabularies and represent all kinds of metadata.
3. **Interoperability**: SPARQL Graph Store is an open-source data management system that facilitates data integration and publishing. SPARQL Protocol for endpoints are the specifications used for data serialization and access.
4. **Standardization**: W3C community process ensures all of the above mentioned characteristics are standardized to adhere to the needs of different actors, from logicians to enterprise data management professionals to system operations teams.

In this thesis, we are using these characteristics of knowledge graphs and will predict links and nodes. We have an existing company knowledge graph from Munich Re which contains data of the insured companies, their sites and the losses that have occurred at these sites. These are the losses that have already been reported to the company. But there can be sites for the same company which was also effected by the same loss event and have undergone a loss, but the loss was not reported to the company (not added to the knowledge graph yet). We will predict the losses and the links to these sites and will see if the loss at the unreported site is likely to occur or not.

In section 2, we introduce main theoretical concepts and the state of the art of the knowledge graph links and nodes predictions. We will see different methodologies that can be used to solve the problem that we are working on.
In section 3, we discuss the problem in more detail and eventually discuss the approaches used to solve this problem. Here we will also see the results of the prediction based solution developed by us and introduce a learning to rank model to evaluate the performance of our developed model.
In section 4, we conclude the thesis by giving a brief summary and a short discussion about the results obtained and end the thesis by mentioning about the future works that can be built upon, or can be resolved using the state of the art of this thesis.

2 State of the art

This thesis deals with the link and nodes predictions of an enterprise knowledge graph. The existing knowledge graph is a directed representation of the knowledge that has to be used for the training of the models and then use the model to predict if the new links and nodes predicted are likely to exist or not. This problem is a prediction problem and there exist multiple ways to do this. Various research papers in the recent years suggest different ways that can be used to get the useful and insightful predictions from the knowledge graphs. Various methodologies have been incorporated in many famous knowledge graphs like DBPedia, Geonames, Wordnet, Factforge etc. and have been used in various domains like information retrieval and natural language processing.

Let us now see various state of the art techniques used in order to get the better and most insightful predictions of the knowledge graphs.

1. **Node2Vec Algorithm**:

   Node2vec[GL16] is an algorithm that is used to generate the embeddings of the entities of a graph or a network into a form that can be represented numerically. The embeddings are created using a neural network with three layers, namely, one input layer, one hidden layer and one output layer.

   The system cannot understand what the nodes and links of the graph actually mean, so it converts these nodes and links to a numerical representation and then try to identify the similarities or some sense out of these entities.

   In Fig. 2.1, we give a pictorial representation what the embeddings generation[Koe] actually mean. Here, we have a sample graph whose embeddings we want to generate. We then give the nodes and links of this graph as inputs to a neural network, whose architecture we have mentioned above, and then obtain a numerical representation of each entity (nodes and links).

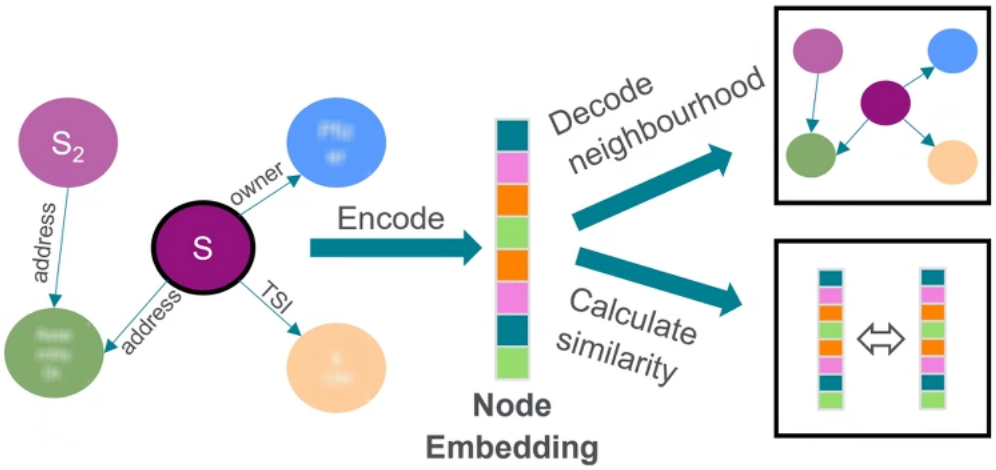

Figure 2.1: Embedding generation

This numerical representation is a one dimensional vector and is called the embedding of the entity. In the similar way, we develop node and link embeddings of all the nodes and links of the graph and perform operations with these embeddings. So, in this way, we can generate insights and the get some sense out of the initial existing nodes and links of the graph. These embedding can now be used to check the similarities and the entities or also to check the neighbours of the nodes and links by computing the simple vector distances etc.

Now let us see how node2vec algorithm works. There are two main steps involved in this algorithm. Let us see the steps in detail:

a) **Step 1: Random Walks Generation**

A Random walk[Xia+20] is a set of some nodes and links and represent a route from a start node(source) to another end node(destination) connected through some links in between. For eg., in Fig 2.1, a path, from node S to address can be considered as a Random Walk. So, to work with graphs, it generates various random walks from source node to the destination node in sets of three, i.e., every random walk will be a triple of two nodes and a link between them.

This is now similar to the word2vec[Mik+13] algorithm, where the nodes and links can be considered as words and the random walk generated can be considered as the sentences. This means, we can use the Skip-gram-model from word2vec algorithm to generate the embeddings of the random walk triples. This is the next step.

b) **Step 2: Skip-Gram Model**[McC]

In this step, we try to find the probability of an entity with respect to the other two entities in the random walk (triple). After first step, we have all the combinations or all the random walks generated for an entity, and we now find the probability of this target entity in context of the other two entities.

We pass the random walk in the form of a vector as input to a neural network, the output layer of the neural network is very dense and consists of softmax activation functions. This output layer will generate probabilities of the other entities in context with the target entity and will output a one dimensional vector of these probabilities. This is the final required vector that we needed and is called embedding.

In this way node2vec algorithm generates the embeddings of the nodes and links of a graph and this concludes the discussion about node2vec algorithm.

Next we see different existing Embedding models that are widely used in order to generate the embeddings and then use these embeddings to evaluate performance of the trained model on the dataset by calculating a score and then assigning the ranks to the triples on the basis of these scores.

Let us see them one by one.

a) **TransE Model:**

It is a Translational based embeddings model[Bor+13]. It works on the vector addition principle. For this, let us consider a triple denoted as (s,p,o), where s is the source node called as subject, p is the relationship between the two nodes and o is the destination node known as object.[Qia+18]

In Fig. 2.2, we see the embeddings of s, p and o represented graphically. The TransE model says that the sum of the vectors (embeddings) of subject and relationship should be as close as possible to the vector or embedding of the object, i.e., the sum and the object vector should be the nearest neighbours. The difference or the loss between the sum and the object embedding is calculated and is then called as TransE score. The scoring function used is given in Equation 2.1.[Lab]

$$f_{TransE} = - \left\| (e_s + r_p) - e_o \right\|_n \tag{2.1}$$

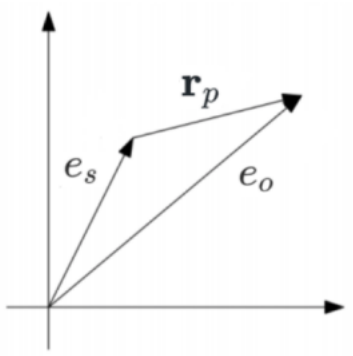

Figure 2.2: TransE representation[Lab]

b) **DistMult Model:**

It is a factorization based embeddings model[Yan+14]. It works on the principle of vector and scalar products. In this model also, the node and link embeddings are calculated using the node2vec algorithm, and then scoring is done. The difference in DistMult and TransE lies in the method of how the scores are calculated.

In this model, the scalar triple product of the three embeddings of a triple (s,p,o) is calculated. The value obtained after computation is scalar value and so its called as scalar triple product. The scoring function used in this model is given in equation 2.2[Lab]

$$f_{DistMult} = \langle r_p, e_s, e_o \rangle \tag{2.2}$$

A scalar triple product for three vectors s, p and o can be calculated as in equation 2.3.

$$(s \times p) \cdot o \tag{2.3}$$

and this yields a scalar number. This number is called the score for DistMult Model.

As we already know that dot product is symmetric in nature, so DistMult model is also a symmetric model by nature. This means that this model is capable to capturing the symmetric relations from the graph very easily and very accurately.

A symmetric relation is a relation which is true both ways. For eg. we have a relation like a person A is married to another person B, in this case, it is also correct to say that B is also married to A, and hence, symmetric.

Now let us see an extension of this model.

c) **ComplEx Model:**

This model[LWZ22] also works on the principle of vector multiplications and comes under factorization based models[Tro+16]. It is an extension of DistMult model. It also follows the same steps of node2vec algorithm to develop the embeddings for the graph entities but the difference lies in the method of scoring when compared to DistMult model.

In this model, instead of using the Real spaces for generating the embeddings, we use Complex spaces to generate the embeddings. So, the scalar triple product used earlier in DistMult model, is now known as Hermitian Product as we are working with complex spaces in this model.

The scoring fuction that is used in this model is given in equation 2.4[Lab]

$$f_{ComplEx} = Re(\langle r_p, e_s, \bar{e_o} \rangle) \tag{2.4}$$

In this scoring function, we use the conjugate transpose of one vector. This is done to make the hermitian product anti-symmetric. This is done so that the model is able to capture the anti-symmetric relations of the graph as well very accurately.

Anti-symmetric relations are those relations which are true only in one direction. For eg., we have a relation like, a person A is father of another person B. Here, the reverse is not true and hence these type of relations are called anti-symmetric relations.

d) **SimplE Model:**

The SimplE model[KP18] is also based translational embeddings approach. It states that embedding of each node of a triple, that is, head and tail of a simple relation is composed of two vectors each. For any entity e, the two vectors are represented as $h_e, t_e \in R^d$ and every single relation r has one single vector as $v_r \in R^d$. Here, h_e captures all the information of the head of the entity and t_e captures all the information about the tail of the entity. Here, the scoring is done using a similarity function. The function for an entity is given in the equation 2.5.

$$f_{SimplE\_Entity} = \langle h_{e_1}, v_r, t_{e_2} \rangle \tag{2.5}$$

So, to get a core for a triple, we need to calculate the similarity score of the whole relation, which consists of two nodes. This will be called as the score of the triple. To calculate this, we take the average of the similarity scores of both the nodes. The final scoring function for SimplE model is given in equation 2.6.

$$f_{SimplE} = \frac{1}{2}(\langle h_{e_i}, v_r, t_{e_j} \rangle + \langle h_{e_j}, v_{r^{-1}}, t_{e_i} \rangle) \tag{2.6}$$

There are many other models also which require high computing power and more deeper analysis and are basically the extensions of the available approaches. Few of them are given below:

a) **HolE Model**[HS17]

b) **QuatE Model**[Zha+19]

c) **MurP**[BAH19]

2. **Ampligraph 1.4.0**[Dub]

   It is an open source Python library that is used for links and nodes predictions in the knowledge graph. It is developed by Accenture Labs, Dublin. It is a branch of machine learning that deals with supervised learning on knowledge graphs.

   The library can be used for the following operations:

   a) To obtain new knowledge and insights from an existing knowledge graph.
   b) To complete the large enterprise knowledge graphs which are incomplete due to missing knowledge.
   c) To develop knowledge graph embeddings which can later be used for various tasks like similarity evaluation, neighbourhood analysis and entity predictions.
   d) To check the validity of a new unseen knowledge based on existing knowledge which has been obtained through embeddings generated for the knowledge graph.

   Some of the key features of Ampligraph are as follows:

   a) **Intuitive APIs:** The APIs and the methods offered by Ampligraph are easy to understand and the user can intuitively identify the correct API to be used for a specific task.
   b) **GPU-Ready:** Ampligraph works on the basis of Tensorflow, and so can easily be executed both on CPU and GPU environments.
   c) **Extensible:** Ampligraph offers its users the way to customize the embeddings models according to their needs. Users can set the their customized hyperparameters and can perform multiple experiments in order to find the best performing values.

   The library is very easy to install and can be installed using pip command. The command to install is given below:

   ```
   pip install ampligraph
   ```

3 Uncovering potential losses using knowledge graph embeddings and link prediction

In this thesis, we plan to use a knowledge graph that was developed in Neo4j. We also tried developing the knowledge graph in Azure Cosmos Database, but found that Neo4j is much better when compared to Azure cosmos DB.

Now let us see why we decided to choose Neo4j over Azure Cosmos DB:

1. **Intuitive:** Having a graph model helps the users to perform data ingestion an retrieval in a much easier and convenient way as the queries for Neo4j are very intuitive and easy to understand.
2. **Reliable:** As the Neo4j databbase follows all the ACID properties, i.e. Atomicity, Consistency, Isolation and Durability, the Neo4j database is very reliable and a very good way of storing the knowledge information.
3. **Ease of Data Manipulation:** In Azure Cosmos Database, the data that comes from the data lakes has to be converted into the schema of the documents that Cosmos DB can accept. For this extra computations are required to form the relevant data documents that can be later stored in the cosmos DB for the Gremlin API to understand. On contrary, in Neo4j, this computation step is not needed an the data from the data lake can directly be stored in Neo4j using the cypher queries that are used to perform Data Manipulations in Neo4j.
4. **Fast:** As compared to Azure Cosmos Database, while dealing with large amount of data, spark is used in order to perform bulk data manipulations which is a slow process. But in Neo4j, this operation is very fast and saves a lot of time.
5. **Economical** Azure cosmos database is expensive as compared to Neo4j. This helps in saving a lot of cost towards the company.

Due to the above mentioned reasons, we decided to build our knowledge graph in Neo4j. The graph is stored on the local server of the company, and can be accessed only through the permitted devices using the company VPN network by using the secured login credentials for the Neo4j server.

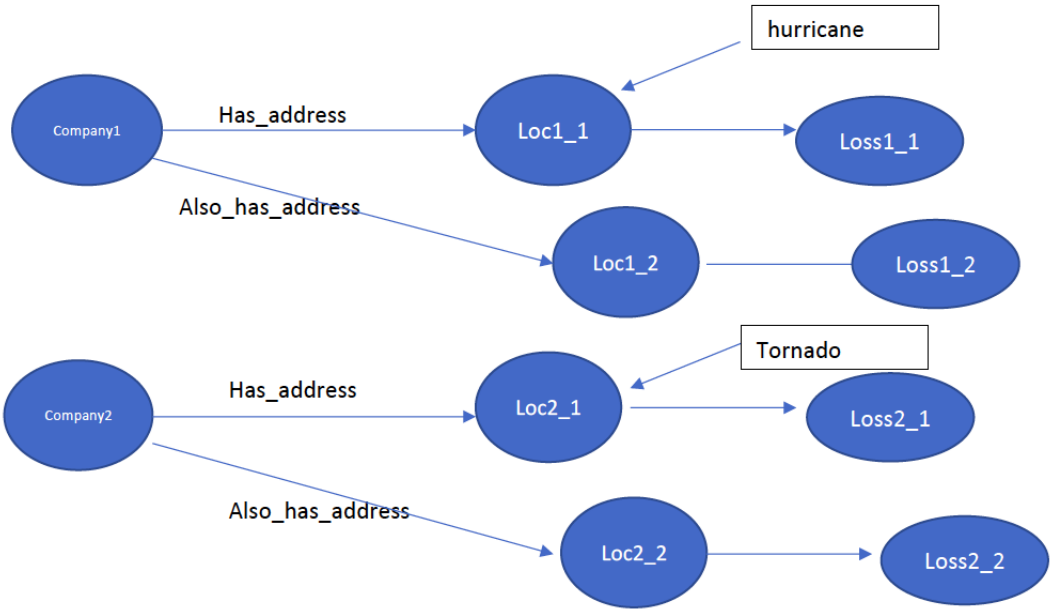

Figure 3.1: Toy use case. We have two companies and both companies have two locations. One location of both companies have suffered a loss and the loss is reported to the company. Second location of both companies have suffered losses but were not reported to the company. We will predict these losses in this thesis

3.1 Introduction to the Problem statement

Let us see the graph in Fig. 3.1 as a toy use case.

Here, we have two companies, Company1 and Company2. Both have two locations respectively, Company1 (Loc1_1 and Loc1_2) and Company2 (Loc2_1 and Loc2_2) and are connected with the link Has_address. Here, the use case is that, the locations Loc1_1 and Loc1_2 are close by and same for Loc2_1 and Loc2_2. So when a calamity like hurricane and tornado hits one of the locations of the both companies, the companies report the losses and so the nodes are added to the knowledge graph. As we see that the other location is also close to the first location of the company, there might be a possibility that the disaster might have affected the second location as well. This is the link that we need to predict. This will predict the loss at the other location or a potential risk loss for the future.

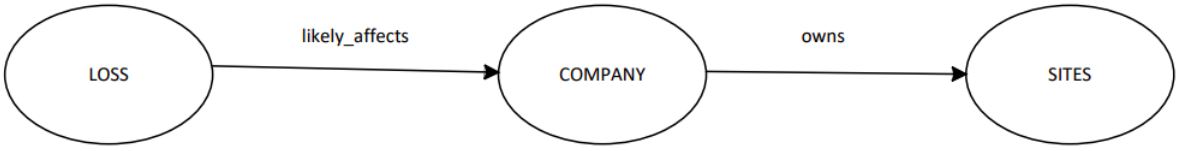

Figure 3.2: Data model of knowledge graph

3.2 Data Model

Now let us see the data model that we will work with. As the title of the thesis suggests that we aim to uncover or predict the losses using the Knowledge graph by using various embedding techniques for the nodes and the links of the graph that we have already seen in the previous State of the art section.

We will try to analyse if the given triple is correct, i.e., it is a potential loss and will examine the likelihood of that triple, else, if the triple we are testing is incorrect, which will mean that this particular triple is having very less likelihood and is not a potential loss. Incorrect or less likely triples may occur due to various reasons, like, the triple data itself is wrong, i.e., the team who has reported the loss to the company, reported the wrong data as it is, the data can be falsely generated or can be a fake data, which can be a very good use case in order to detect and capture the fake entries in the dataset and detect the anomalies, if any, in the dataset.

Now let us have a look at the Fig. 3.2. The figure depicts the data model of a triple that we will be using from our knowledge graph. This is a part of the graph that is important for the computations in this thesis.
Let us see each component of this part of the knowledge graph here in detail.

The data model in Fig. 3.2 consists of five components, three nodes and two links.
We will now have a look on these components in some detail.

Let us first look at the three nodes:

1. **LOSS Node**

   This node is called the 'LOSS' node. It consists of various attributes that are related to this loss event like date of the occurrence of loss, causes of the loss, status of the loss, amount of the sum insured for the company where the loss had occurred etc. These attributes contain all the details that are required to identify a loss event and are stored in the form of a JSON. These properties form a complete structure of the Loss node which will be used for future computations.

2. **COMPANY Node**

   This node is called the 'COMPANY' node. It consists of various attributes that are related to an insured company, like ID of the company, name of the company, headquarter location of the company, data source it has been received from etc. These attributes contain all the details that are required to identify an insured company and they are also stored in the form of a JSON. These properties form a complete structure of the Company node which will be used for future computations.

3. **SITE Node**

   This node is called the 'SITE' node. It consists of various attributes that are related to a specific site of a company, like ID of the site, name of the site, address of the site, latitude and longitude of the site etc. These attributes contain all the details that are required to identify a site and they are also stored in the form of a JSON. These properties form a complete structure of the Site node which will be used for future computations. There can be multiple sites that can be linked to a specific company, i.e., a company can own multiple sites at different locations.

Now let us have a look at the links used in the data model above in detail.

1. **LIKELY_AFFECTS Link**

   This is the link which is called as 'Likely_Affects' and joins 'LOSS' and 'COMPANY' nodes. The direction of this link is from Loss node to the company node. The triple generated by this link is **(Loss, Likely_Affects, Company)**. This triple depicts the relation between a loss and the company that it affected with the relationship of 'likely_affects' between them. The properties of this link just contain the name of the link and is also stored in the JSON format. This forms the complete structure of the link of 'likely_affects'.

2. **OWNS Link**

   This is the link which is called as 'OWNS' and joins 'COMPANY' and 'SITE' nodes. The direction of this link is from Company node to the site node. The triple generated by this link is **(Company, Likely_Affects, Site)**. This triple depicts the relation between a company and the site it owns with the relationship of 'owns' between them. The properties of this link just contain the name of the link and is also stored in the JSON format. This forms the complete structure of the link of 'likely_affects'.

Now as we have seen the data model in detail, let us now have a look at the dataset and the data preparation that was needed before we could start the experiments on the dataset.

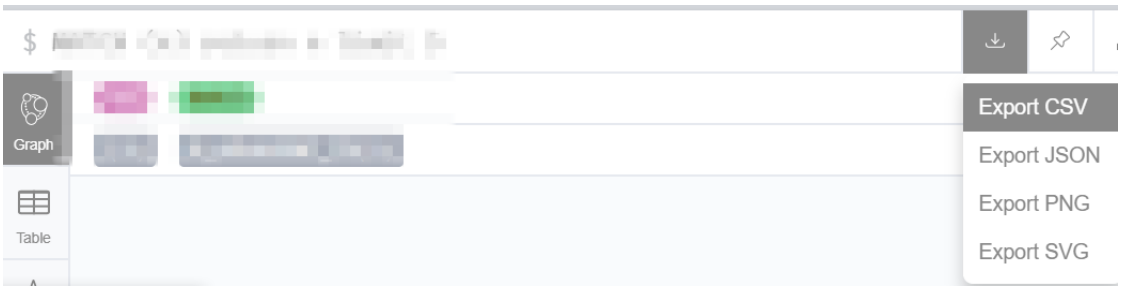

Figure 3.3: Export data to CSV file from Neo4j server.

3.3 Dataset preparation for experiments

As we have seen the data model, let us now see how we prepared the dataset and retrieved the data of nodes and links from the knowledge graph.

The data model that we need to work with is small subset of the big complete knowledge graph which has many more nodes and links and are linked to each other with many relationships. Here, we used Cypher queries to fetch the data that corresponds to our data model that we need to work with. We developed many different queries to fetch the data from different regions of the world.

The challenge that we faced while fetching the data using cypher was exporting this huge amount of data to csv file, that can later be used for experimentation.

The dataset contains approximately 4400 loss nodes which are linked to atleast one company by the link 'likely_affects'. The company node is connected to site node. Here, the amount of dataset depends on the number of site nodes. A company can have one site affected or can have multiple sites affected as well. So, in order to start our experiments, we limited the number of sites to two per company node and then tried preparing our dataset. So in this toy case, we had around 8800 records connecting all the three nodes and every company connected to any two of its sites.

In Fig. 3.3, we can see that the Neo4j server itself offers a way to download the data that has been retrieved from the knowledge graph using the using the cypher queries in the form of CSV, JSON, PNG and SVG. We will download the data in the form of CSV files as we will be using CSV files for the next part of our implementation.

As we mentioned above that downloading the data in CSV formats was possible when we have a relatively smaller dataset. So, if we want to incorporate all the sites that are owned by a specific company, the total number of records explode to 23 million, downloading that amount of data in CSV format was not feasible.

So, we spilt our data according to regions, like Europe, Americas, Asia etc. and download the data for each region separately by limiting and skipping the records in the cypher queries. Just for example, United States of America, alone consisted of approximately 3 million records, and the Neo4j could handle around 75000 records at a time, so it took around 35 iterations and 35 separate CSV files to get the data of USA. So, in order to simplify this, we discussed the issue internally and reached the conclusion that using ten iterations also instead of 35 could also work out for our experimentation. So, we took ten CSV files for USA, which consisted of around 800 thousand records. Similarly we did multiple iterations for different regions and fetched the data in separate CSV files. Later we combined data from all the CSV files to generate the final CSV file with all the records and then used this generated CSV file for our computations.

The whole implementation has been done in Python using various machine learning libraries that are supported by python, like pandas, matplotlib, numpy, ampligraph etc. Ampligraph is the main library that we are going to use to work on knowledge graphs.

We chose Ampligraph library to work because of the following advantages:

1. **Ease of implementation:** This library is simple and easy to use and understand. It contains intuitive methods and functions which are easy to keep in mind.
2. **Well structured:** The library is well structured and it is not difficult to follow the steps involved in the process of link predictions.
3. **Many embedding models:** The library also provides different kinds of embedding models that we can use and evaluate and choose the best one.

Until now we have seen the dataset which has all the sites of a company that was affected by the loss. Now let us look at the dataset in another way.

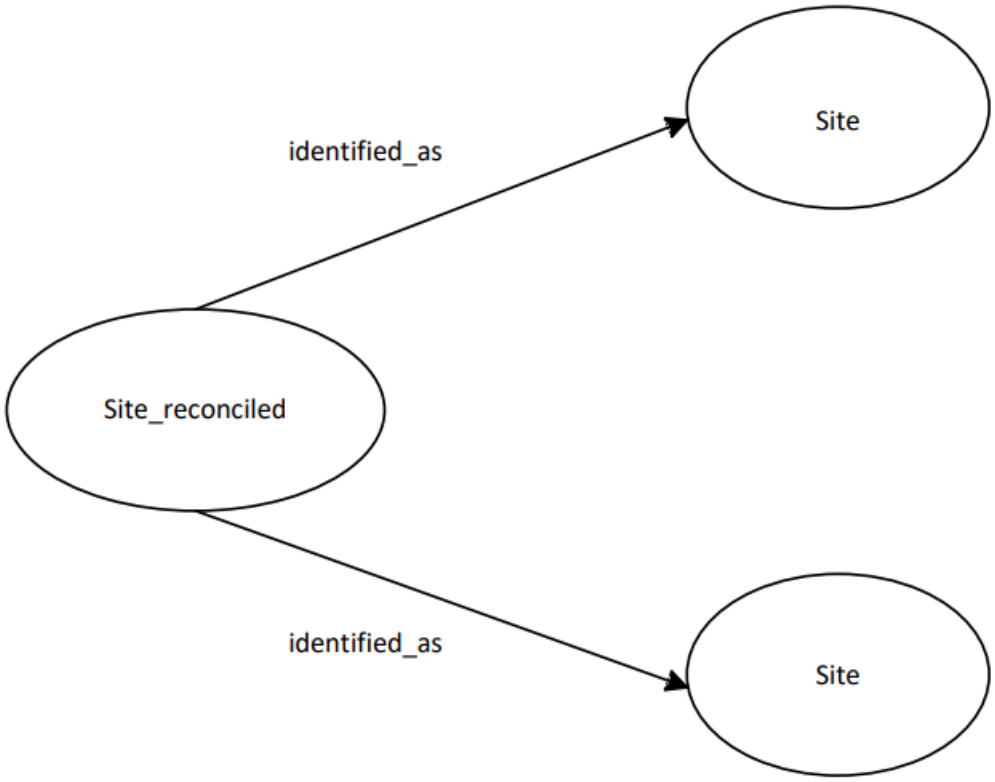

Figure 3.4: Reconciled Site triple

In order to solve the problem of multiple sites for a company, the developers of the knowledge graph came up with a solution of having one more additional node, called as 'Reconciled node' for the sites. This node can be called as the parent node for all the sites. It basically denotes the group of sites that are connected to a specific company.

The site reconciled node consist of the few details for the sites of the company, like the name of the company, data source, ID etc. It is connected to the company reconciled node on one end and to the sites on the other. The link between the site reconciled node and sites is shown in Fig. 3.4. The link used here to connect the reconciled site node to the site node is called 'identified_as'. Hence, we see that we use reconciled site node to identify any given site for the company. This reduces the large number of sites for the companies to just one single node and makes the computations easy.

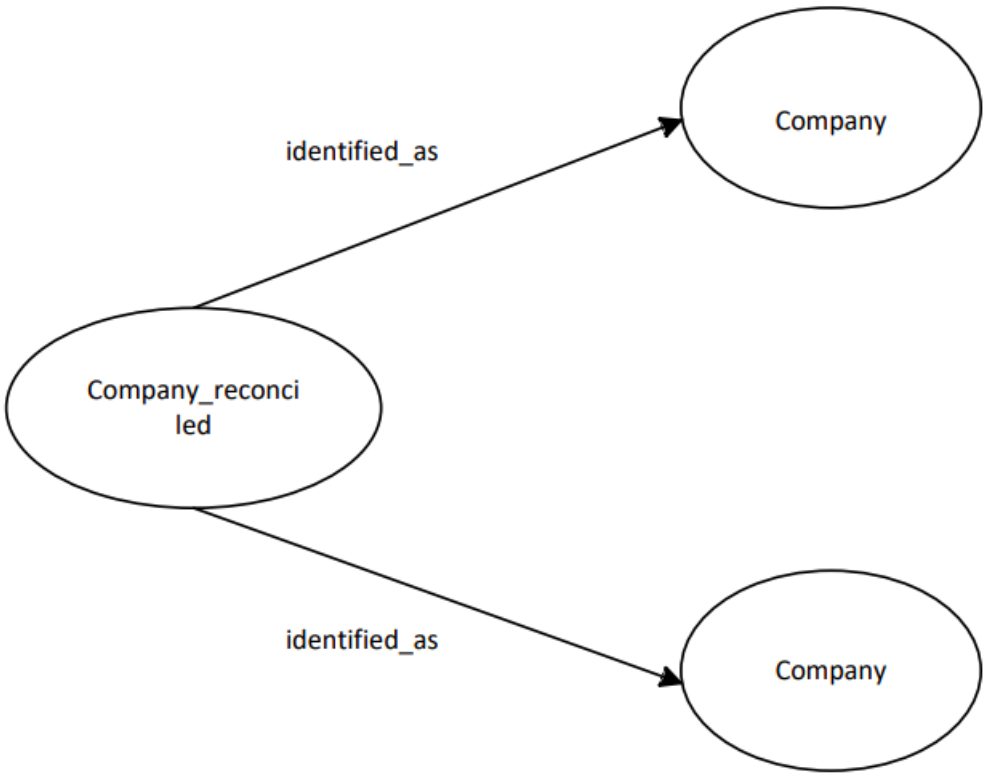

Figure 3.5: Reconciled company triple

Same thing has been done for the company nodes. There can be multiple company nodes that can represent the branches of the same companies with only minor details changed and majority of the details same. In this case as well, the branch companies have one single parent company and we call this parent company node as the reconciled company node. This has been shown in the Fig. 3.5.

So, if we consider the dataset with a loss linked to a reconciled company node which is then linked to a reconciled site node, we get around 4400 records, which is a very small number when compared with 23 million which we have seen in the previous dataset.

As we have mentioned that the reconciled site nodes do not have many attributes, it was a challenge to rebuild the properties of the reconciled node similar to an actual site node. A site node consists of numerous attributes which contain all the data and information i.e. all the knowledge that is related to that specific site. Directly copying the values from a site node to the reconciled site node would be a wrong way to generate the properties of reconciled node. So we perform some aggregations like taking the means of values of the properties that had numeric values. This way we were able to assign the values to the properties that we have added to the new generated reconciled node.

There were also many cases where the data itself was missing. The first dataset we considered is with all the sites, there were many records where some properties had missing values. To overcome this issue, we used the imputation techniques[Ngu]. Imputation is done whenever we deal with missing data problem. In our case, we also checked for the records where the data values were missing, what imputation techniques could be used. Most of the data attributes that had missing numeric values, we assigned the mean values to them, and for the few attributes which had string values, for example, data source attribute value is missing, in such cases, we checked the source of the company that site is connected to and then assign that source to the missing source attribute of the site.

In this way, we prepared the dataset with minimal discrepancy and errors, so that we can use it for our machine learning experiments in much more efficient manner and ultimately get the best possible predictions and insights from the dataset.

Now, in the next sections we will see how we implemented this project completely from the scratch and see each and every step in detail.

3.4 Evaluation Metrics

In this thesis, we will be performing learning to rank algorithm in order to evaluate the performance of our trained model. In our knowledge graph, we have data in the form of triples. We will take the correct test triple and then generate corruptions for each test triple. Then, we find a way to obtain the rank of the correct triple among all the corruptions by following few steps that have been discussed in detail in the next section.

We find the score for the correct test triple and also for the corruptions that have been generated using the formulae we saw earlier in state of the art section. The scoring function varies and depends on the type of embedding model we are using to train our model. Corruptions are generated for both subject and object and then we sort them in ascending order according to their scores. Then we check the position at which the correct test triple lies, both for subject and object corruptions and call this position as the rank of the test triple. In this way, we obtain two ranks, one for subject and one for object and then visualise them on various plots to get better insights. We will see these metrics and the plots in much detail in the coming sections.

3.5 Project implementation

3.5.1 Importing Data

Technologies Used:

1. **Neo4j**, for fetching the data from the knowledge graph using the cypher queries.
2. **Python 3.7**, including machine learning libraries like numpy, tensorflow, matplotlib etc. for most of the code.
3. **Ampligraph 1.4.0**[Cos+21], the main library that we will use to generate the embedding and ultimately perform the link predictions, which is the aim of this thesis.

Now let us see the implementation steps in detail.
Initial computation involves getting the data in the format that can be later processed by the library ampligraph for the training and evaluation. For this we utilize the power of ampligraph to work with CSV files. The CSV files should contain three columns, first column corresponds to the first node of the triple, second column corresponds to the relationship link of the triple and the third column corresponds to the second node of the triple.

Next we see how to import the data from the csv file.
For this we use the Pandas, which is a python library and is very helpful when working with CSV files. We use the method 'read_csv()' that is provided by pandas to read the CSV files. The syntax used for this method is given below:

```
csv_data = pd.read_csv('data_csv.csv')
```

After performing the reading operation using the code snippet mentioned above, the data that we retrieve from the csv is stored in the form of dataframes. Pandas dataframes are very convenient to use when the data is in tabular form and is imported from an external CSV file.

The CSV file that we have prepared in the previous section where we discussed the data preparation, contains two links and three nodes, resulting in five columns per record. However, the library can handle the CSV files with three columns only. So we modify the CSV file and try to bring the five columns to tree columns. For this

task, we take each record and keep the first triple as it is in the CSV file which corresponds to the first three columns of the CSV file. This first triple is the connection of Loss node to the company node via a relationship 'likely_affects' as we have seen before.

The next part of the record constitutes to another triple. This triple connects the company node to the site node via a relationship 'owns'. The link 'owns' and the site node corresponds to the fourth and fifth column of the CSV file. We now take this triple and bring it below the first triple.

For this, we use pandas. We write a python script that captures the third, fourth and fifth column of the CSV file and appends it below the first three columns, i.e., third column is appeneded below the first column, fourth column is appended below the second column and the fifth column is appended below the third column. Doing such way, will bring our CSV dataset from five columns to three columns and thus, making our CSV file iterable by the ampligraph library for further model training and evaluations.

However, ampligraph takes an array of data or a python list which is then passed through different embedding models that we are going to work with later in this project. So, we then convert the dataframes obtained from the CSV file to a numpy array and in this way our data is ready to be processed further. Below is the syntax that can be used to convert a dataframe to a numpy array.

```
data_array = Pandas_dataframe.to_numpy()
```

Hence, now we use this array that we have generated from the pandas dataframes and proceed with further steps.

3.5.2 Training, Validation and Test dataset split

As we know, for any Machine Learning or a Data Science project, we need to split the complete dataset into three parts, namely:

1. **Training Set**
2. **Validation Set**
3. **Test Set**

We now see each of these datasets in detail:

1. **Training set**:

   This data set is used to train the model that we are going to use. We use this part of the dataset to fit our model according to our needs. This is the part of the dataset that the model actually sees and learns from. The training phase is responsible to learn the hyperparameters, if any, of the model and if the hyperparameters are optimal, the model is used for the final predictions.

2. **Validation Set**:

   This is the part of the dataset that we use for hyperparameters tuning. This is also called as 'Dev set' or the 'Development set' which makes sense as we are using this during the development stage of our final model. We use this validation set more than once to evaluate the performance of the model as test set should be used only once to avoid any bias. Our aim is to keep fine tuning the parameters and so the model only sees the data and do not learns from it.

3. **Test Set**:

   This is the part of the dataset that is used for the final testing of the trained model. The model that we have trained using the training dataset and have fine tuned the parameters using the validation dataset, we use this final model on the test dataset to finally evaluate the performance of our model. This means the results that we obtain or the predictions that we obtain after passing the test

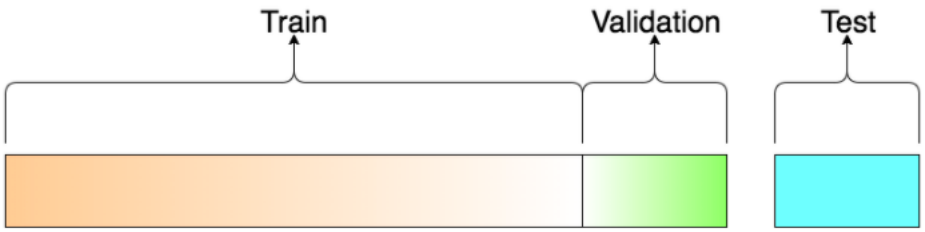

Figure 3.6: Dataset split, Training(80%), Validation(10%) and Test(10%)

dataset to the model, are our final results and the insights of the dataset.

Now we see how the dataset is split into Training, Validation and Test datasets.

Fig. 3.6 shows a typical split of the whole dataset. Usually a common practice is to divide the complete dataset in the 3 parts with training dataset being 80% of the whole dataset and validation and test datasets being 10% each.

In the Fig. 3.6 we can see that the Validation set is shown attached to the training Dataset. This is because validation set is also a part of the training as we use it to hypertune the parameters of our models. Test set is shown seperately because test data should never be touched during training. This prevents having extra biasing and the model can then predict and give the best results and insights at the first time when the test set is put through the model.

As we have already mentioned before that in this project, we are using the Ampligraph library from python to develop the machine learning solution for the problem statement and then use the same library to perform the link predictions. So to split the complete dataset into Training, Validation and Test dataset, we use an inbuilt method provided by this library.

The method is called **train_test_split_no_unseen()** and takes two paramenters. This method divides the entire dataset into training dataset and test dataset. The parameters of this method are:

1. **Dataset array**: This is the dataset that we need to split in to training data and test data.
2. **Test set size**: This is the percentage or the amount of records that we put that are

sliced from the complete dataset and are put in another array.

The method returns the two dataet after splitting, Training and test dataset.

For our computations, we have put the test set size to be 20%. We will use 10% of it for the test dataset and the remaining half we will see below.

Since we have split it into only two dataset as of now - train and test dataset, we also need a third split of the dataset as validation dataset as we have seen before. For this, we use the same method to split the dataset and this time, instead of complete dataset, we only pass our test dataset and assign the new test data size as 50%. This way we split the previously obtained test dataset in to half, 10% goes for validation dataset and the remaining 10% of the whole data goes for the actual test set. This newly obtained test dataset will be used at the end and we will not touch this dataset even once during our training phase.

Once the test data set is ready, we have to now check for the triples that might have had connections to the other triples that are in the training dataset. For this, we find connections between the nodes that are in test dataset and training dataset. If the relation between the any node from test dataset exist with any node from the training dataset, we remove that triple from our test dataset. This will help to create a more robust model with minimal bias introduction due to the test dataset. Hence, the dataset is split and is ready for computations.

3.5.3 Training Phase

In this sub-section of our main experiment part of the thesis, we will see how the training is done, different models that we have used to create the embeddings of the links and the nodes, how well we fit the model and how well our trained model performs on the validation set.

For this we have experimented with three embedding models that we have already seen and read about in the state-of-the-art section of our thesis. The research papers associated to these models have also been cited and can be referred if more in-depth knowledge is required.

Let us now see all the three models in detail and the parameters that they take for training the model.

1. **TransE Model**

   As we have seen in the state of the art section that this is a translation based embedding model, it works on the principal of nearest neighbourhood of the vectors. The concept behind this model is that the embedding of the object of the triple should be the nearest neighbour of the embedding vector obtained by the sum of the embeddings of the subject and the link of the triple.

   This way, the difference between these two embedding vectors is the loss of the TransE model which also we have seen in the state of the art section. Through evaluating it on the validation dataset, we try to minimize this loss and obtain the best possible embeddings of the entities of each triple.

   Now let us have a look at the syntax of the TransE model.

   It is imported from the module called 'latent_features' of the ampligraph library and is imported in the following way:

```
from ampligraph.latent_features import TransE
```

The syntax to create the TransE model is given below:

```
model = TransE(batches_count, seed, epochs, k, optimizer,
    optimizer_params, loss, regularizer, regularizer_params)
```

Let us see each of these parameters in detail.

Table 3.1: The table shows all the parameters used in the model with their descriptions.

Parameter	Description
batches_count	This is the number of the batches in which the training set must be split while running the training loop.
seed	The seed is used by the random numbers generator functions used internally.
epochs	It is the number of the iterations that are done during the training phase.
k	It is the required dimension of the embedding space where the embeddings for the nodes and the links are generated. So, it is the dimension of the embedding vector that is generated.
optimizer	Here we get various optimizers to choose from. The optimizers that are offered are 'Stochastic Gradient Descent', 'Adaptive Gradient', 'Adam' and 'Momentum'. We can choose any one of these optimizers and check which one performs the best on the validation set.

optimizer_params	These are the hyper parameters of the optimizer that are to be learned during the training of the model. We then check how these parameters work on the validation dataset and then fine tune them to obtain the optimal parameters. It accepts the parameters in the form of a dictionary and the most important parameter to keep track of is the learning rate of the neural network used internally.
loss	The library, similar to optimizer, also provides various options of the loss functions to choose from. Various loss functions provided are 'pairwise margin-based', 'negative log loss likelihood', 'absolute margin likelihood', 'adversarial sampling', etc. We can then try out some of these loss functions and the decide which one performs the best for our dataset.
regularizer	It is used to regularize the loss function that we choose to use in our project. Based on the parameters provided for the regularizer, we can decide if we want to use L1, L2 or L3 regularizer.
regularizer_params	These are the parameters that we pass to decide which regularizer do we want to use. It is also passed in the form of a dictionary.

Depending on the values we provide to the above mentioned parameters of the TransE, we tried a few sets of parameters and found the following set of the parameters to perform the best on the validation dataset.

The values that performed best are given in Table 3.2

Table 3.2: The table shows all the parameters used in the model with their values for TransE model.

Parameter	**Value**
batches_count	20
seed	0
epochs	75
k	150
optimizer	'adam'
optimizer_params	{'lr' : 1e-3}
loss	'pairwise'
loss_params	{'margin' : 1}
regularizer	'LP'
regularizer_params	{'p' : 3, 'lambda' : 1e-3}

This way we create the TransE Model.
Next, we try to fit this model on our Training set that we had formed before.
For this as well, the ampligraph library provides with a very good method called as 'fit()'. This also takes a parameter which we see below after the syntax.
The syntax used to train the model on the Training set is given below:

```
model.fit(training_data)
```

Here we see that the fit() method takes one parameter. This is the training data on which we want to train and fit the model that we have created before. So the training data is in the form of an array of 'n' dimensions, where n is the number of records of the training dataset.

This ends the model creation using TransE model. Let us see the other models as well.

2. **DistMult Model**

   This is another type of model that is based on the factorization principles, unlike translation that was used for TransE model which we discussed above.

   In this model, as we saw in the state of the art section, scalar triple product is calculated for all the three embeddings of the subject, link and the object of a triple. The value obtained from this scalar triple product is the value that is assigned to the scoring function. Here also, the loss is calculated in the same way as in the TransE model.

   The advantage of this model is that, it can easily capture the symmetries of the knowledge graph link as dot product is symmetric. We have seen this also in detail in state-of-the-art section.

   Now we see how we can import and implement the DistMult model

   It is also imported from the module called 'latent_features' of the ampligraph library similar to the TransE model and is imported in the following way:

```
from ampligraph.latent_features import DistMult
```

   The syntax to create the TransE model is given below:

```
model = TransE(batches_count, seed, epochs, k, optimizer,
    optimizer_params, loss, regularizer, regularizer_params)
```

   We see that most of the parameters used in DistMult model are same as the TransE model. This is because both the models follow the similar algorithm of Node2vec to form the embeddings of the subject, link and the object of a triple. Only difference lies in the scoring functions that we have seen in the state of the art section as well where we have discussed the scoring functions of all the models that we have used in this thesis in detail.

   Now, similar to what we did for TransE model, we tried a few sets of the parameter values and found a set that performed the best for this model. The parameter set is given in table 3.3. The values might change depending on the

dataset and the amount of data we work with.

Table 3.3: The table shows all the parameters used in the model with their values for DistMult model.

Parameter	**Value**
batches_count	10
seed	0
epochs	75
k	150
optimizer	'adam'
optimizer_params	{'lr' : 1e-3}
loss	'pairwise'
loss_params	{'margin' : 5}
regularizer	'LP'
regularizer_params	{'p' : 3, 'lambda' : 1e-3}

This way we create the DistMult Model.

Next, we try to fit this model on our Training set that we had formed before.

For this as well, we use the 'fit()' method like we did for TransE model before. This also takes a parameter which we see below after the syntax.

The syntax used to train the model on the Training set is given below:

```
model.fit(training_data)
```

This ends the creation of the DistMult model.

3. **ComplEx Model**

This is the last embedding model that we have tried in this thesis. It is also based on the principle of factorization. This model is similar to DistMult model which we discussed before, but extends few things that makes it more robust and widely used model.

In this model also, as we discussed in the state of the art, we calculate the scalar triple product. The different between DistMult model and ComplEx model is that, the embeddings in ComplEx model are made in complex spaces unlike In DistMult where the embedding are generated in the Real spaces. Here, to calculate the score we take the embeddings of the subject and link same as they are generated by the node2vec algorithm, but we take the conjugate transpose of the third vector, i.e., the embedding of the object of the triple.

This way, the dot product does not remain symmetric and becomes anti-symmetric, which helps in capturing all the anti-symmetries of the knowledge graph triples. For our dataset, as we have seen earlier, that the links that we have are in the form of (Loss, likely_affects, Company), and most of the links of our dataset seem to be anti-symmetric. This gives us an insight that ComplEx model should perform the best for our dataset. We will see the results and evaluation in the next subsections of the thesis.

Now we see how we can import and implement the ComplEx model.

It is also imported from the module called 'latent_features' of the ampligraph library similar to the TransE and DistMult models and is imported in the following way:

```
from ampligraph.latent_features import ComplEx
```

The syntax to create the ComplEx model is given below:

```
model = ComplEx(batches_count, seed, epochs, k, optimizer,
    optimizer_params, loss, regularizer, regularizer_params)
```

We see that most of the parameters used in ComplEx model also are same as the

TransE and DistMult models. This is also because all the three models follow the same algorithm of Node2vec to form the embeddings of the subject, link and the object of a triple. Only difference lies in the scoring functions that we have seen in the state of the art section as, where we have discussed the scoring functions of all the models in detail.

Now, similar to what we did for TransE and DistMult models, we tried a few sets of the parameter values and found a set that performed the best for this model. The parameter set is given in table 3.4. The values might change depending on the dataset and the amount of data we work with.

Table 3.4: The table shows all the parameters used in the model with their values for ComplEx model.

Parameter	**Value**
batches_count	10
seed	0
epochs	100
k	150
optimizer	'adam'
optimizer_params	{'lr' : 1e-3}
loss	'nll'
regularizer	'LP'
regularizer_params	{'p' : 2, 'lambda' : 1e-3}

This way we create the ComplEx Model.

Now we will see how the validation is done.

Before starting with the validation, we will discus the metrics introduced in previous section in more detail that we are going to calculate and use to evaluate the performance of our models.

Let us see the per triple metrics given below:

1. **Score**

   This is the score value that is calculated by the model using the scoring functions that we have seen before. Each model has different scoring functions and so assigns the score to the triples differently.

   To obtain the score of a triple, we can use the following syntax. This also makes use of the method 'predict()' which is also provided by the ampligraph library and predicts the score for that triple. To simplify, we can extract one test triple and just pass it through the predict() method. The method takes only the test triple as a parameter in the form of an array and returns a number, which we call as score. The score can be negative or positive, float or integer, depending on the test triple and the scoring function we use.

   Let us now see how do we use this predict() method.

```
score = model.predict(test_triple)
```

   Now the question arises, what does this score tell us? The straight answer to this question is 'Nothing'. It is just a number and to interpret it significantly we can follow two options.

   a) We can perform hypothesis tests, for this we can create a list of many hypothesis that we aim to test, assign them this score and then take first n hypothesis as True triples.
   b) Or we can perform the learning to rank task. For this we generate negative samples or corruptions, score all the triples, both true and corrupted, and then see the rank of the correct triple. Rank can tell us how good is our model performing on the validation set.

2. **Rank**

   We use learning to rank task in order to evaluate the performance of our model in this thesis. To obtain the rank we have to follow a set of steps. The steps are given below:

   a) **Step 1:** Compute the score of each triple.

   b) **Step 2:** Get subject corruptions. These are generated when we corrupt the subject, keeping the other entities, link and the object constant for each triple.

   c) **Step 3:** Compute the score of the generated subject corruptions as well.

   d) **Step 4:** Sort the generated subject corruptions in ascending order according to the score, find the position where the correct or the true triple lies. This is the rank of the subject

   e) **Step 5:** Repeat the same steps two to four for the objects of the triples to create the object corruptions. The rank we obtain here are the object ranks.

   f) **Step 6:** Return a set of tuple containing two values, subject rank and object rank.

   Why two ranks?

   In the steps above, we saw that we are generating the subject and object corruptions and then ranking them separately. Let us consider subject corruptions first. Here, if the rank obtained by the correct triple or the test triple that we are evaluating, is a good rank, then this means that the model is more likely to predict the correct subject for that specific link and object that we kept constant. Good rank means it ranks among the top most triples and the model predicted the correct subject very fast without any issue. But if the rank obtained is a low rank or a bad rank, this means that the model is not able to predict the correct subject and predicts the corrupted triples first before predicting the true triple.

   The same reasoning applies to the object ranks. Good ranks mean that the model is able to predict the object correctly and easily for that specific link and

subject that we kept constant. But if the rank is bad, it also means that the model is not able to predict the correct triple easily, and instead keeps predicting the wrong or the corrupted triples before predicting the correct one.

Now let us see some of the results that we have obtained from all of our work discussed above.

We will first visualise the dataset that we are working with. The complete dataset was unable to visualize in one visual, so we tried visualising a part of the data that captures most of the insights of the knowledge graph that we have used.

In Fig. 3.7 we can see many nodes and links. The centre node of each structure denotes a loss node and the links are the likely_affects and the connected nodes are the company nodes.

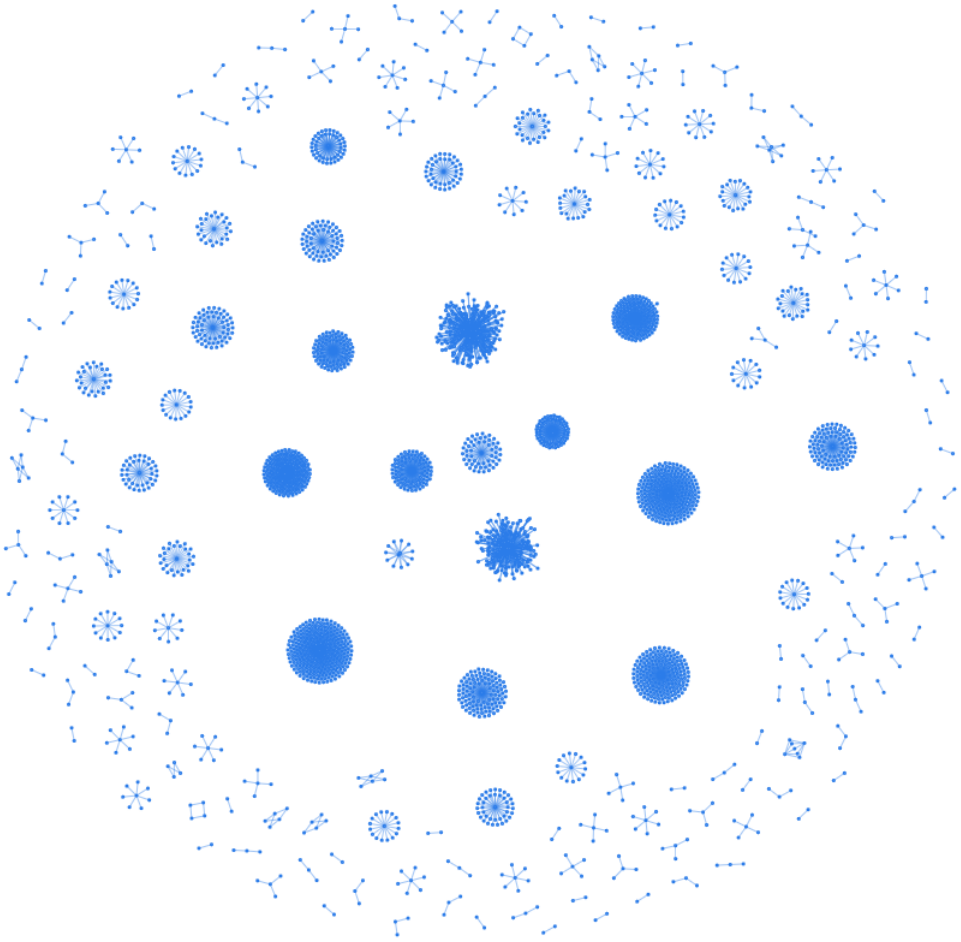

Figure 3.7: Visualisation of a subset of the company Knowledge graph: The small clusters on the outside show the losses that have effected only a few companies. The dense clusters inside show the most common losses that have effected many companies at once. The central node is the Loss node and the surrounding connected nodes are the effected Company nodes.

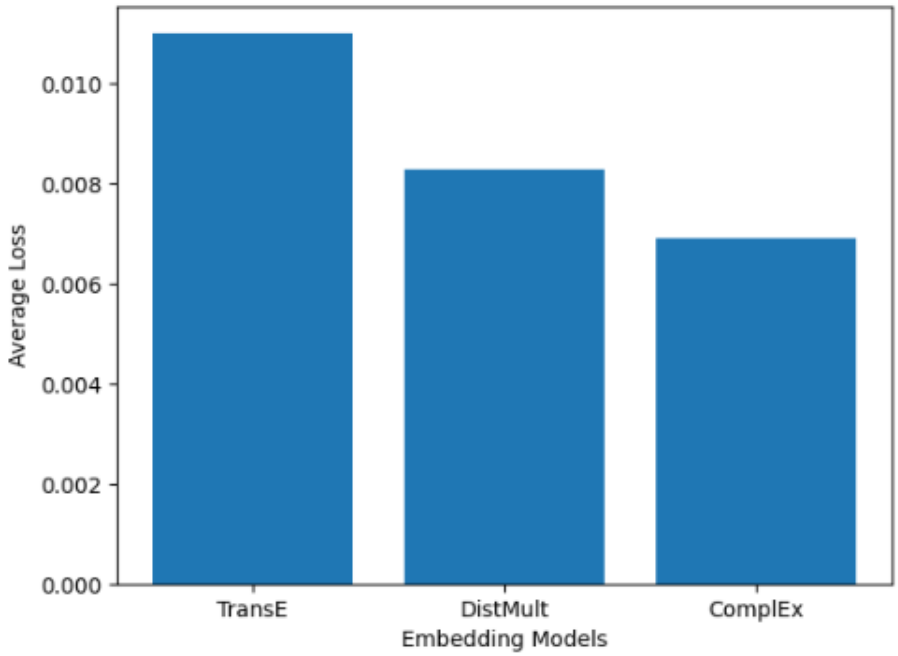

Figure 3.8: Comparison of all three models and their average losses

Now let us see the performance of the three models and compare them to each other.

Fig. 3.8 shows us the average losses of all the three models. We see that the losses obtained by TransE model is approximately 0.012, DistMult model is approximately 0.0085 and ComplEx model is approximately 0.0069.

We see that the ComplEx model performs the best on our validation data. This is what we expected. As we have seen before that ComplEx models captures the anti-symmetries of the graph easily. As in our knowledge graph, all of the links are anti-symmetric, which means that the links that we are working with are of the triple (Loss, Effects, Company) form and the reverse of this triple, i.e., (Company, Effects, Loss) is not true. Hence, ComplEx model should have performed the best and we obtain the same result as well with our experiments.

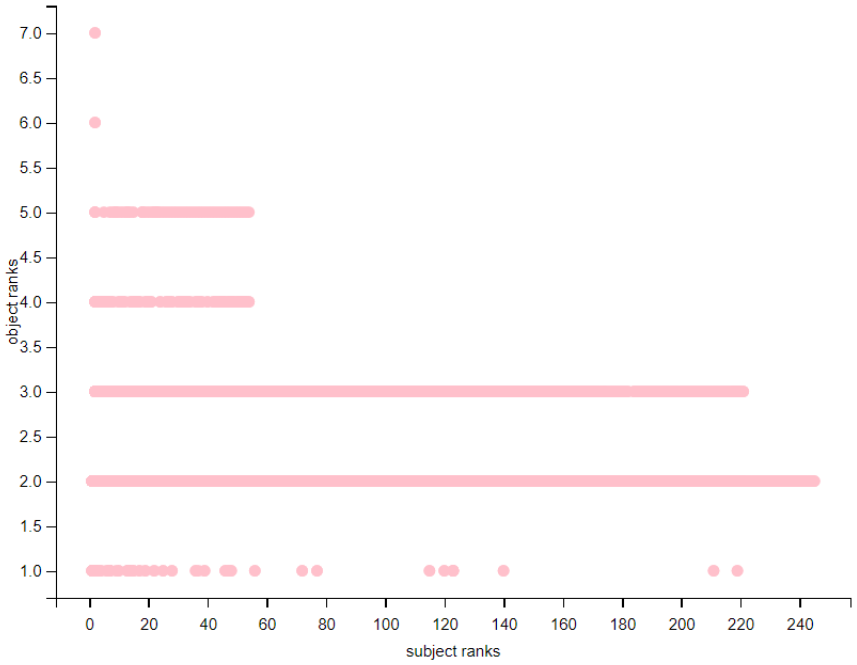

Figure 3.9: Actual Training Ranks

3.6 Results

As we have seen before, we have used the Learning to Rank approach to evaluate the performance of the model that we chose to proceed in our thesis. In this section, we will see the ranks of the training set as well as test set and try to find out how good is our model working.

Since, we have a large dataset of around 12 million records, it is very important to see what range of ranks can be considered good and what can be considered bad ranks. For this, we first try to visualise the training set ranks, as we already know that the training data set is correct and should give the correct range of the ranks.

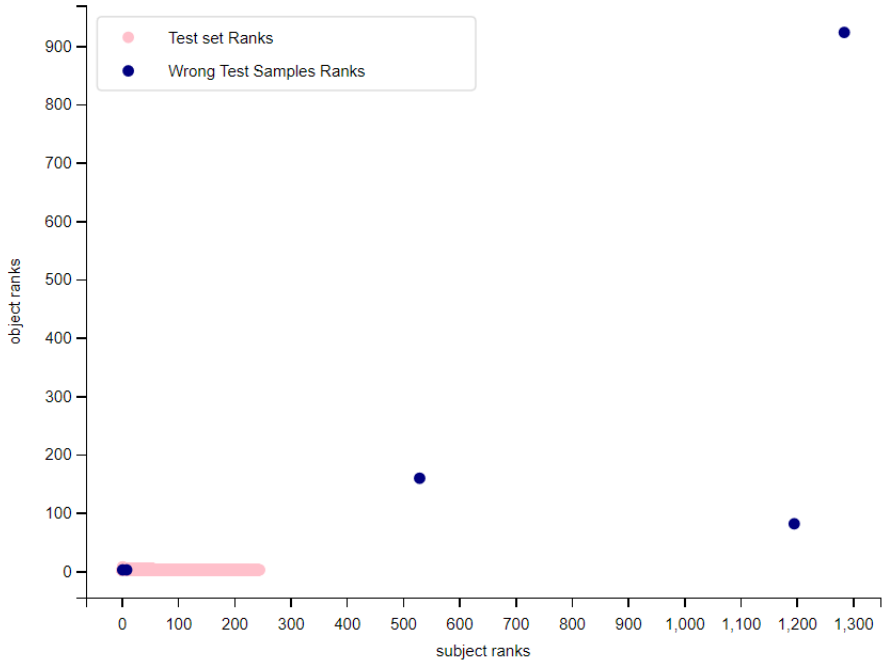

Figure 3.10: Actual Test Ranks

In Fig. 3.9 we plot the subject ranks on x-axis and object ranks on y-axis. We then get a range of ranks for both subjects and objects and as these ranks have been generated from the training set, we can consider these ranks as good ranks. Here we see that for subjects, the good rank ranges from 0 to 240 and for objects, it ranges from 0 to 7.

Here, the main thing to observe is that, if the ranks are close to the origin, it implies that the ranks are good, and as the rank moves away from the origin, it implies that the ranks are not very good.

Now we try to visualise the test data set ranks on a graph and see what ranks are we able to obtain from our model.

In Fig. 3.10, we have plotted Test ranks in the same manner as we saw the plot for the training ranks, where subject ranks are plotted on x-axis and object ranks are plotted on y-axis. The ranks are plotted in two colors, pink and blue. We will now see the meaning of these colored ranks in further parts.

The pink ranks indicate that the ranks that have been predicted by our model are correct and lie in the ranges that we have identified earlier from the Training ranks. The blue ranks indicate some specific test samples that we have taken and considered separately in order to give better insights about the performance of the model.

To do this, we have considered five test triples. Two Triples are the correct test triples and the other three triples have been deliberately made wrong by us. Out of these three wrong test triples, one wrong test triple is from the same region, but the loss would not have effected the company included in that triple, in our case we have taken both loss and company from US region, but the loss has not effected the company. For the other two wrong test triples, we have taken losses and companies as well from different regions. For one wrong triple, we took loss from US region and the company from UK region and for the other wrong test triple, we took loss from US region and the company from China.

The ranks that we obtained for the correct triples are [9 2] and [2 2], which are correct and can be easily seen in Fig. 3.10 in the region of correct rank ranges.

For the wrong test triple where the loss and company both are from same region, we obtain the rank [530 159], which is bad as the triple is less likely to occur.

For the other two wrong test triples, where the losses and companies are both from different regions, we obtain the ranks as [1285 923] and [1196 81], which are more bad as these triples are least likely to occur.

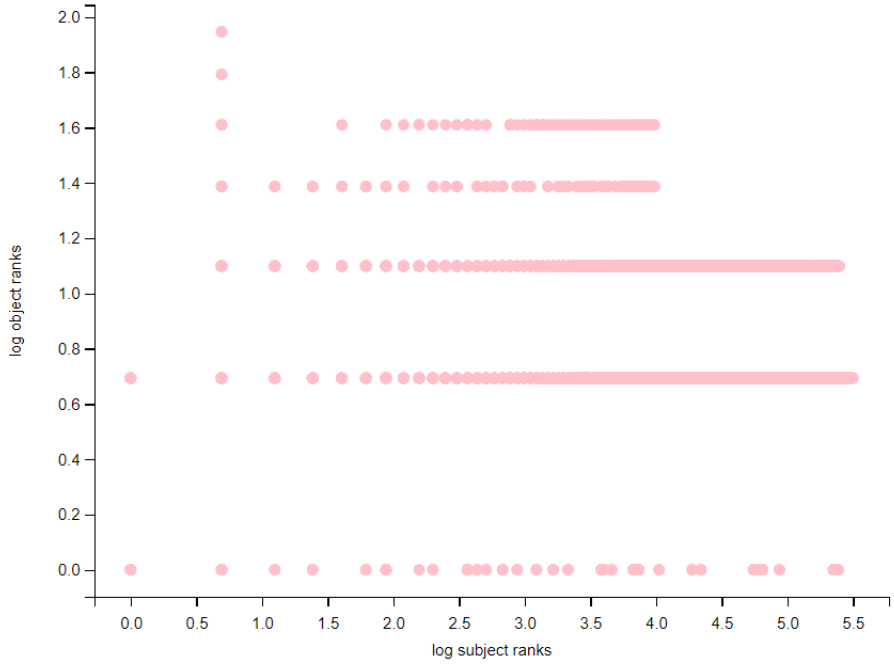

Figure 3.11: Log Train Ranks

In order to have a better visualisation of the ranks, we now present the graphs of log of the ranks instead of the actual ranks as it is tough to have clear visuals from Fig. 3.9 and Fig. 3.10.

In Fig. 3.11 and Fig. 3.12 as well, log ranks that are closer to the origin are considered good ranks and further ranks can be considered not good. In Fig. 3.12 we can clearly see the two correct test triples that we have tested and are giving good ranks, but the other three test triples that were deliberately made wrong are giving bad ranks, which was also the expected result.

This concludes our main chapter of the thesis which was focused on the demonstration of the work and the process followed in order to achieve the desired results.

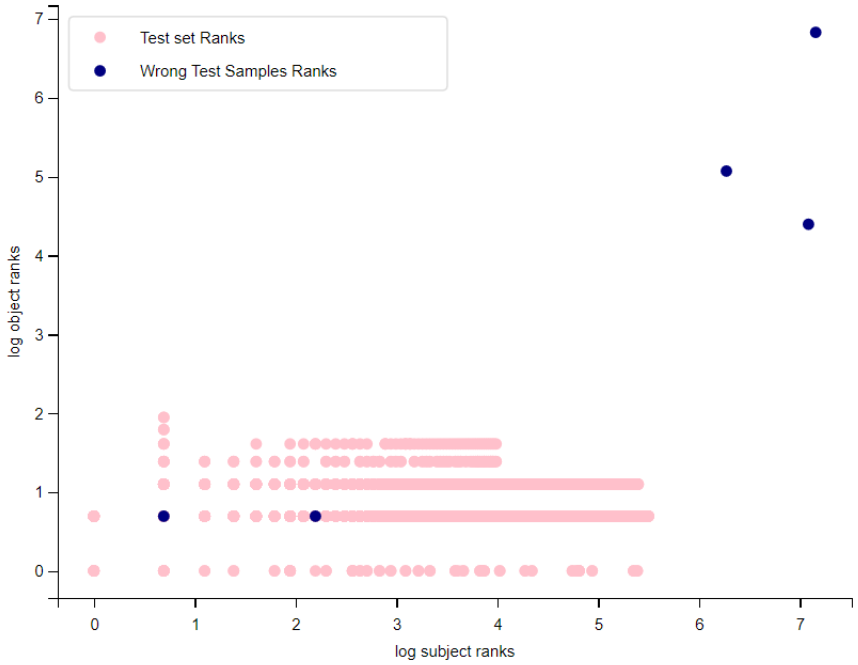

Figure 3.12: Log Test Ranks. Here, blue dots represent the ranks of the chosen test samples. Two test samples which lie in the pink range are the correct samples ranks. The other three blue dots represent the ranks of the wrong test samples. They are samples where the loss is highly unlikely to effect the company. These ranks lie far from the correct range and are considered as bad ranks

4 Conclusion

The study and the work in this thesis aimed at finding the optimal method for performing the link and node prediction of a knowledge graph. We used the existing incomplete knowledge graph from the company and presented solutions to complete the graph by predicting nodes and the links. We discussed in detail about the state of the art for the knowledge graph embedding models. The intensive research performed during the initial phase of the thesis helped in choosing the right tools and models for performing the experiments and then evaluating them on the test dataset. We chose to experiment with three types of embedding models, namely, TransE, DistMult and ComplEx models, the choice of these also depends on the nature of the graph and the dataset we are working with. We found that all the three models are efficient and perform decently well on the knowledge graph, but ComplEx model performed the best with minimum average loss among all the three models. For evaluation, we consider learning to rank approach where we follow a set of pre-defined steps to rank the test samples. We then plot the ranks on a 2d graph and visualise them. We compare the ranks of the test samples with the correct range of ranks obtained from training dataset and see that the ranks of the wrong test samples are very bad, which means that model performs well and predicts the links and nodes efficiently.

To demonstrate performance in much better way, we performed the analysis with five specific test samples. Out of these five test samples, two test samples were perfectly correct but the other three test samples were made deliberately wrong by us. We corrupted the data to check if the model can identify the wrong data. After evaluation, we saw that the two test samples which were correct gave good ranks, whereas, the other three test samples which were wrong gave very bad ranks. This tells us that the test samples that gave bad ranks are very highly unlikely to exist, which means that the link between the entities is least or not probable. This is also the expected result, as we know that the test samples are wrong and do not exist. So, we can say that our model performs very well and gives accurate desired results.

The results discussed above can be very useful in various use cases. The ranks that lie outside the good rank region, which we call bad ranks, can be considered as outliers. We can also sometimes get outlier ranks for the test samples without corrupting the

actual data and this can help us to capture fraud and unreliable data. The bad ranks will be the indication to look into those data samples again and try to find out what can be the issue for it having a bad rank. This will involve some discussion with the company underwriters who take care of the data on day to day basis. We can dig deep in to these data samples and find out the root cause. Also, using the embeddings generated, we can perform similarity and neighbourhood analysis for the entities. We can find similarities between the reports that are generated for each loss event and find the similar probable reports, so the underwriters can predict the future loss reports for the insured companies. We can also enhance the work on this thesis towards the completion of the knowledge graph by adding the unseen data samples based on their scores and ranks, and if they lie in the good rank ranges, they can be added to the existing knowledge graph. This concludes the final chapter of this thesis.

Bibliography

[BAH19] I. Balazevic, C. Allen, and T. M. Hospedales. "Multi-relational Poincaré Graph Embeddings." In: *CoRR* abs/1905.09791 (2019). arXiv: 1905.09791.

[Bor+13] A. Bordes, N. Usunier, A. Garcia-Duran, J. Weston, and O. Yakhnenko. "Translating Embeddings for Modeling Multi-relational Data." In: *Advances in Neural Information Processing Systems*. Ed. by C. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K. Weinberger. Vol. 26. Curran Associates, Inc., 2013.

[Cos+21] L. Costabello, sumitpai, N. McCarthy, P. Tabacof, R. McGrath, C. L. Van, A. Janik, C. Clauss, A. Alto, D. Tekin, P.-Y. Vandenbussche, and Aayam. *Accenture/AmpliGraph: AmpliGraph 1.4.0*. May 2021. URL: https://doi.org/10.5281/zenodo.4792436.

[Dub] A. L. Dublin. *Ampligraph documentation*. URL: https://docs.ampligraph.org/en/1.4.0/.

[GL16] A. Grover and J. Leskovec. "node2vec: Scalable Feature Learning for Networks." In: *CoRR* abs/1607.00653 (2016). arXiv: 1607.00653.

[HS17] K. Hayashi and M. Shimbo. "On the Equivalence of Holographic and Complex Embeddings for Link Prediction." In: *CoRR* abs/1702.05563 (2017). arXiv: 1702.05563.

[Koe] W. Koehrsen. *Embeddings generation*. URL: https://towardsdatascience.com/neural-network-embeddings-explained-4d028e6f0526.

[KP18] S. M. Kazemi and D. Poole. *SimplE Embedding for Link Prediction in Knowledge Graphs*. 2018. DOI: 10.48550/ARXIV.1802.04868.

[Lab] A. Labs. *Tutorial of Knowledge Graph*. URL: https://kge-tutorial-ecai2020.github.io/ (visited on 09/04/2020).

[LWZ22] X. Li, Z. Wang, and Z. Zhang. "Complex Embedding with Type Constraints for Link Prediction." In: *Entropy* 24.3 (2022). ISSN: 1099-4300. DOI: 10.3390/e24030330.

[McC] C. McCormick. *Skipgram model*. URL: http://mccormickml.com/2016/04/19/word2vec-tutorial-the-skip-gram-model/.

[Mik+13] T. Mikolov, K. Chen, G. Corrado, and J. Dean. *Efficient Estimation of Word Representations in Vector Space*. 2013. DOI: 10.48550/ARXIV.1301.3781.

[Ngu] M. Nguyen. *Data Imputation techniques*. URL: https://bookdown.org/mike/data_analysis/imputation-missing-data.html.

[Ont] OntoText. *Fundamentals of Knowledge Graph*. URL: https://www.ontotext.com/knowledgehub/fundamentals/what-is-a-knowledge-graph/.

[Qia+18] W. Qian, C. Fu, Y. Zhu, D. Cai, and X. He. "Translating Embeddings for Knowledge Graph Completion with Relation Attention Mechanism." In: *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI-18*. International Joint Conferences on Artificial Intelligence Organization, July 2018, pp. 4286–4292. DOI: 10.24963/ijcai.2018/596.

[Tro+16] T. Trouillon, J. Welbl, S. Riedel, É. Gaussier, and G. Bouchard. "Complex Embeddings for Simple Link Prediction." In: *CoRR* abs/1606.06357 (2016). arXiv: 1606.06357.

[Xia+20] F. Xia, J. Liu, H. Nie, Y. Fu, L. Wan, and X. Kong. "Random Walks: A Review of Algorithms and Applications." In: *IEEE Transactions on Emerging Topics in Computational Intelligence* 4.2 (Apr. 2020), pp. 95–107. DOI: 10.1109/tetci.2019.2952908.

[Yan+14] B. Yang, W.-t. Yih, X. He, J. Gao, and L. Deng. *Embedding Entities and Relations for Learning and Inference in Knowledge Bases*. 2014. DOI: 10.48550/ARXIV.1412.6575.

[Zha+19] S. Zhang, Y. Tay, L. Yao, and Q. Liu. "Quaternion Knowledge Graph Embedding." In: *CoRR* abs/1904.10281 (2019). arXiv: 1904.10281.