Universidade de Aveiro
2021

Rodrigo Amaral
Ribeiro Pereira

Descoberta de conhecimento biomédico através de representações contínuas de grafos multi-relacionais

Biomedical knowledge discovery through multi-relational graph embeddings

Universidade de Aveiro
2021

Rodrigo Amaral Ribeiro Pereira

Descoberta de conhecimento biomédico através de representações contínuas de grafos multi-relacionais

Biomedical knowledge discovery through multi-relational graph embeddings

Dissertação apresentada à Universidade de Aveiro para cumprimento dos requisitos necessários à obtenção do grau de Mestre em Engenharia de Computadores e Telemática, realizada sob a orientação científica do Professor Doutor Sérgio Guilherme Aleixo de Matos, Professor Auxiliar do Departamento de Engenharia de Telecomunicações e Informática da Universidade de Aveiro

o júri / the jury

presidente / president	**Professor Doutor Joaquim João Estrela Ribeiro Silvestre Madeira** Professor Auxiliar na Universidade de Aveiro
vogais / examiners committee	**Professor Doutor Joel Perdiz Arrais** Professor Auxiliar no Departamento de Engenharia Informática da Fac. de Ciências e Tecnologia da Universidade de Coimbra (arguente)
	Professor Doutor Sérgio Guilherme Aleixo de Matos Professor Auxiliar na Universidade de Aveiro (orientador)

agradecimentos / acknowledgements

Gostaria de começar por agradecer ao Professor Sérgio Matos pela sua orientação, mostrando-se sempre disponível para se reunir comigo e esclarecer as minhas dúvidas ou discutir ideias.
Gostaria também de fazer um agradecimento geral ao corpo docente do DETI, que durante estes últimos cinco anos para além de todo o conhecimento teórico e prático que me ofereceram, me ajudaram a desenvolver o meu senso crítico, a minha ética de trabalho e muitas outras competências que serão certamente fundamentais na minha vida profissional.

Agradeço obviamente à minha família, especialmente aos meus pais que moldaram a pessoa que sou hoje e me apoiaram em todas as etapas da minha vida e ao meu irmão Guilherme que para além de ser família é um dos meus melhores amigos.

Finalmente agradecer ao meu grupo de amigos. Tanto ao Frederico, ao Luís e ao Lucas que conheço desde de que me lembro. Como àqueles que tive a oportunidade de conhecer durante estes últimos cinco anos, o Pinho, o Francisco, o Miguel, o Diego, o Dinis, a Margarida, o Chaves, o Hugo, o Pousa, e muitos outros que por uma questão de manter esta secção curta não menciono diretamente.

Resumo

Grafos de conhecimento são grafos multi-relacionais que permitem organizar informação de maneira a que esta seja não apenas passível de ser inquirida, mas que também permita a inferência lógica de nova informação por parte de humanos e especialmente sistemas computacionais. Recentemente vários métodos têm vindo a ser criados de maneira a maximizar a informação que pode ser retirada destas estruturas, sendo a área de "Machine Learning" um dos grandes propulsores para tal. "Knowledge graph embeddings" (KGE) permitem que os componentes destes grafos sejam mapeados num espaço latente, de maneira a facilitar a aplicação de tarefas como a predição de novas ligações no grafo ou classificação de nós.

Neste trabalho foram exploradas as capacidades e limitações da aplicação de modelos baseados em "Knowledge graph embeddings" a redes biomédicas existentes, dado que a biomedicina é uma área na qual têm sido feitos esforços no sentido de organizar a sua vasta base de conhecimento em grafos de conhecimento, e onde esta capacidade de predição pode ser usada para potenciar avanços nos seus diversos domínios. Para tal, no presente trabalho, vários modelos foram estudados e uma pipeline foi criada para treinar os mesmos sobre algumas redes biomédicas. Os resultados mostram que estes modelos conseguem de facto ser precisos no que diz respeito à tarefa de predição de ligações em alguns conjuntos de dados, contudo esta precisão aparenta ser afetada por características inerentes à estrutura do grafo.

Adicionalmente, com o conhecimento adquirido durante a realização deste trabalho foi criado um "notebook" que tem como objetivo servir como uma introdução à área de "Knowledge graph embeddings" para investigadores interessados em explorar a mesma.

Abstract

Knowledge graphs are multi-relational graph structures that allow to organize data in a way that is not only queryable but that also allows the inference of implicit knowledge by both humans and, particularly, machines. In recent years new methods have been developed in order to maximize the knowledge that can be extracted from these structures, especially in the machine learning field. Knowledge graph embedding (KGE) strategies allow to map the data of these graphs to a lower dimensional space to facilitate the application of downstream tasks such as link prediction or node classification. In this work the capabilities and limitations of using these techniques to derive new knowledge from pre-existing biomedical networks was explored, since this is a field that not only has seen efforts towards converting its large knowledge bases into knowledge graphs, but that also can make use of the predictive capabilities of these models in order to accelerate research in the field. In order to do so, several KGE models were studied and a pipeline was created in order to obtain and train such models on different biomedical datasets. The results show that these models can make accurate predictions on some datasets, but that their performance can be hampered by some inherent characteristics of the networks.

Additionally, with the knowledge acquired during this research a notebook was created that aims to be an entry point to other researchers interested in exploring this field.

Contents

Contents **i**

List of Figures **v**

List of Tables **vii**

1 Introduction **1**

2 Background **3**
2.1 Word Embeddings . . . 3
2.2 Graphs Embeddings . . . 4
2.2.1 Single-relational network embedding methods . . . 4
Factorization Methods . . . 5
Random Walk Based Methods . . . 6
Neural Network Based Methods . . . 8
2.2.2 Multi-relational network embedding methods . . . 8
Matrix Factorization/Tensor Decomposition Models . . . 8
Geometrical Models . . . 9
Deep Learning Models . . . 9
2.3 Graph embedding based link prediction and applications in the biomedical field . . . 11
2.3.1 Pharmaceutical Field . . . 12
2.3.2 Multi-omics Field . . . 12
2.3.3 Clinical field . . . 13

3 Methods **16**
3.1 Technologies . . . 16
3.1.1 Neo4J . . . 16
Comparison with alternatives . . . 17
Discussion . . . 22
3.1.2 Pykg2vec library . . . 22
Comparison with alternatives . . . 22
Discussion . . . 25

3.2 Embedding and link prediction . . . 25
3.2.1 Models used . . . 25
TransE . . . 25
DistMult . . . 26
ConvE . . . 26
3.2.2 Data preparation . . . 27
3.2.3 Training pipeline . . . 27
3.2.4 Training Methods . . . 28
Relevant Global Configuration Parameters . . . 29
Relevant Model Hyperparamters . . . 30
Running the trainer . . . 33
3.2.5 Hyperparameter optimization . . . 34
Additional arguments used in BaysOptimizer . . . 35
Running the optimizer . . . 36
3.2.6 Testing and Metrics . . . 38
Raw vs Filtered Ranking . . . 38
Mean Rank (MR) . . . 39
Mean Reciprocal Rank (MRR) . . . 39
Hits@X . . . 39
3.3 Limitations of the methodology used . . . 39

4 Experiments and Results 41
4.1 Experimental playground . . . 41
4.2 Experiment One: UMLS' Semantic Network . . . 41
4.2.1 Dataset . . . 41
4.2.2 Hyperparameter search and training of the final model . . . 42
Timings . . . 43
Testing Metrics . . . 43
4.2.3 Discussion . . . 45
4.3 Experiment Two: Comprehensive Antibiotic Resistance Database . . . 46
4.3.1 Dataset . . . 46
4.3.2 Hyperparameter search and training of the final model . . . 49
Timings . . . 50
Testing Metrics . . . 50
4.3.3 Discussion . . . 50
4.4 Experiment Three: Altered Comprehensive Antibiotic Resistance Database 53
4.4.1 Dataset . . . 54
4.4.2 Hyperparameter search and training of the final model . . . 59
Timings . . . 59
Testing Metrics . . . 61
4.4.3 Discussion . . . 61

5 Conclusion 63

Bibliography **65**

Appendix I **71**

List of Figures

2.1 Frequency of different terms on literature across the years 4
2.2 DeepWalk pipeline 6
2.3 Results of different graph embedding methods on a mirrored network [1] . 7

3.1 Example of a simple city and country relation on GraphDB's data model . 17
3.2 Example of a simple city and country relation on Neo4J's data model . . . 18
3.3 Example of a simple city and country relation on Grakn's data model . . . 18
3.4 GraphDB's visual graph tool 20
3.5 Grakn Workbase 20
3.6 Neo4J visualization tools 21
3.7 ConvE's architecture 27
3.8 Pipeline used in order to obtain, train and test the models used 28
3.9 L1 vs L2 norm 32
3.10 Tensorboard tool for visualizing effects of different sets of hyperparameters 37

4.1 UMLS's training Loss and validation fMRR during the training process for the different models 44
4.2 Distribution of relation types on the UMLS dataset 46
4.3 Overview of the Original CARD Network 48
4.4 Leave-one-out 5 fold cross validation split strategy 49
4.5 Training loss and validation fMRR of the different splits along the epochs on the original ARO dataset 51
4.6 Efflux pumps from the MATE superfamily 57
4.7 Visualization of the modified dataset 58
4.8 Training loss and validation fMRR of the different splits along the epochs on the altered ARO dataset 60

List of Tables

2.1 Taxonomy and summary of the studied KGE methods 11
2.2 Researches that uses single-relational graph embeddings on biomedical networks 14
2.3 Researches that uses multi-relational graph embeddings on biomedical networks 15

3.1 Comparison of training times of the different studied KGE libraries 23

4.1 Hyperparameters used in the last version of each model for the UMLS dataset 43
4.2 Total training time and training time per epoch of the models on the UMLS dataset in seconds 44
4.3 Testing results of the different models on the UMLS dataset 45
4.4 Best hyperparameters found for the "ideal" split of the original CARD's ARO dataset 50
4.5 Total training time and train time per epoch metrics of the different models for the original ARO dataset. All the values presented are in seconds . . . 52
4.6 Testing results of the different models on the original CARD's ARO dataset 52
4.7 Best hyperparameters found for the "ideal" split of the altered CARD's ARO dataset 59
4.8 Total training time and train time per epoch metrics of the different models for the altered ARO dataset. All the values presented are in seconds 61
4.9 Testing results of the different models on the altered CARD's ARO dataset 61

Chapter 1

Introduction

With the rise in the amount of data produced and archived in the last decades in the research and business fields, the necessity to make sense and to derive new knowledge from it also grew.

Following the release of Google's Knowledge Graph, interest around this type of structure rose. The main reason being that it allows not only to store the data, but to maintain structural meaning between said data, allowing for seamless integration between knowledge from various domains, as such many fields saw efforts being made in order to generate such structures, one such field being the biomedical field with projects such as Bio2RDF[2] and UniProtKB[3].

Alongside this movement, the Machine Learning (**ML**) community also became interested in the idea of developing ML methods that could derive new knowledge based on the one already existent in these structures, being it by predicting new links, labeling exiting nodes, or discovering groups of nodes that share similar characteristics (clustering). Performing such tasks on graph like structures was however not a new concept. Pioneer works such as Locally linear embedding (LLE) [4] proposed in 2000, explored the idea of representing the entities of a graph as vectors that preserved the context about their structural context. Using these vectors it was then possible to apply some more traditional ML methods, such as neural networks, in order to derive new knowledge. However, knowledge graphs differ from the traditional single-relational graphs that these algorithms worked with, the main difference being the multi-relational nature of knowledge graphs comparatively to the others. In multi-relational graphs the connections between the entities contained additional information about the type of relation the connected entities establish between them. As such, a new set of solutions was developed in order to not only embed the entities of the graph but also the different types of relations in it.

Despite growing in popularity in the last years, knowledge graph embeddings are a new concept, and research around it is still being done. This work places focus on applying such concepts to the growing number of available biomedical knowledge graphs and understanding the viability of using such methods to improve and/or accelerate the discovery of new knowledge.

As such, this work proposes to fulfil the following objectives:

- Exploring the state of the art knowledge graph embedding methods, going in depth on the theoretical concepts behind them;
- Comparing different open source libraries available for Knowledge graph embedding;
- Defining a pipeline to find, train and test knowledge graph embedding models for link prediction on biomedical datasets;
- Creating an educational notebook to serve as an entry point for researchers interested in getting started with using Knowledge graph embeddings.

Chapter 2

Background

The concept of embedding is not new. Its rise in popularity is connected to the work of Bengio et al. described in the paper “A Neural Probabilistic Language Model” [5], where the authors propose a new method that allows obtaining lower dimension representations of words while preserving information about the context they appear on a given corpus, based on having neural networks learning a distributed representation of the words. This representation allows computer systems to have a deeper understanding of the words, closing the gap between human and machine understanding of concepts (it allows, for example, to answer questions such as “How similar are apples to oranges?”). Given this property this paper created a new wave of research in the ML field, especially in the Natural Language Processing (NPL) community.

In this chapter, the concept of embedding will be further explored, starting with word embeddings, all the way up to single and multi-relational graph embeddings, discussing their potential use on biomedical data along the way.

2.1 Word Embeddings

In 2003 Bengio et al.[5] proposed the first neural probabilistic model for the task of predicting the probability of a sequence of words. Compared to the N-gram model, which was the state-of-the-art model at the time, the model proposed by Bengio, learnt distributed representations of words (embeddings), which allowed the model to understand similarities between words, and as such make better predictions. However the popularity of word embedding in the NLP research field is attributed to Mikolov et al., who in 2013 created “word2vec” [6], a set of models that allowed training embeddings of words that could then be applied on several NLP tasks. The paper proposed two single hidden layer feed forward network architectures, the “Continuous Bag of Words” (CBOW) model and the “Skip-Gram” model, both having become standards in the word embedding world [1]. In the CBOW model, the network learns to predict the target word based on the words in

[1] https://code.google.com/archive/p/word2vec

its k-sized neighborhood (its context). On the other hand the Skip-Gram model does the inverse, meaning that it learns to predict the context based on the target word.

After training the network on a training set, the weight matrices of the single hidden layer of the model are low dimensional representations of the embeddings.

2.2 Graphs Embeddings

Networks (or graphs) are a prominent concept in the academic world. In the book "Network Science" [7], the author shows that the use of the term is increasing at a faster rate than terms such as "evolution" or "quantum" since the 1980's (figure 2.1).

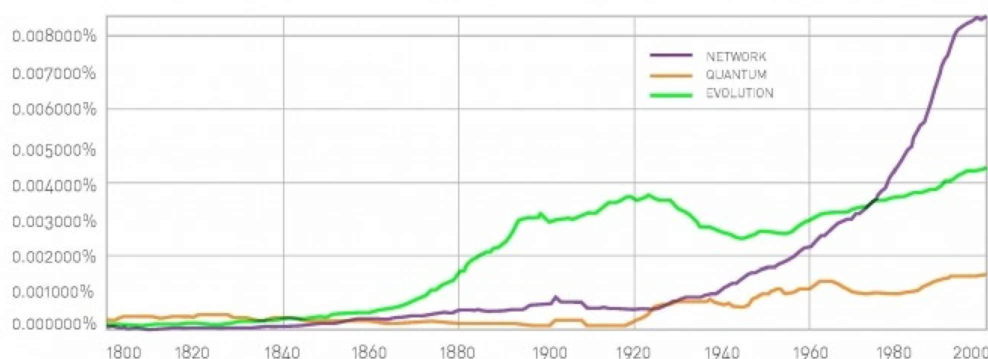


Figure 2.1: Frequency of different terms on literature across the years

Source: http://networksciencebook.com/chapter/1#summary

Given the popularity of these structures, the ML community developed an interest in being able to extract information from them. The concept of graph embeddings appeared as a way to represent the nodes of a graph in a manner that is easy to feed to pre-existing ML models.

The different existing methods can be categorized by the type of graph they work with and by the strategy they use to obtain the embeddings. In terms of the type of network they operate upon, they can be categorized into "Single-relational network embedding methods" and "Multi-relational network method".

2.2.1 Single-relational network embedding methods

Single relational graphs/networks are graphs whose connections are unlabeled, meaning that it is not possible to differentiate two connections between nodes.

Inside the category of single-relational network embedding methods, different strategies are used to obtain the embeddings. According to a survey made by Palash Goyal et al. [8] they fall under one of the following three categories:

- Factorization methods;
- Random-walk methods;
- Neural network based methods.

Factorization Methods

The key idea behind factorization methods is the decomposition of a matrix that represents connections between nodes of the graph into a product of lower dimension matrices, which will be used to obtain the embeddings. Different algorithms can use different types of matrices, such as adjacency matrices (eg. LLE [4]), Laplacian matrices (eg. Laplacian Eigenvectors [9]), node transition probability matrices (eg. GraRep [10]), and others.

In 2000, Roweis et al. proposed “Locally Linear Embedding” (LLE) as a solution to the non-linear dimensionality reduction problem. This algorithm starts by constructing a similarity graph by connecting each datapoint with its k-nearest neighbors (based on some measure of distance, such as Euclidean distance). Then, on the basis that the dataset is big enough and that every node in this graph has very close neighbours, it assumes that each node can be represented as a linear combination of its corresponding neighbours. Based on this concept, it proceeds to obtain an adjacency matrix W, where each element W_{ij} represents the weight of node j in the representation (linear combination) of node i. To obtain such a matrix, a cost function based on the error of reconstruction of the original datapoint(X_i) given its neighbours(X_j) is minimized (Equation 2.1).

$$E(W) = \sum_i |X_i - \sum_i (W_{ij}X_j)|^2 \tag{2.1}$$

Then, given the proximity between the nodes and the datapoint in question, it can be assumed that the reconstruction matrix `W` is approximately the same in a lower dimension ($d < D$). As such, it is possible to obtain a lower dimension representation of X_i, represented by Y_i, by minimizing the cost function (Equation 2.2), that like the previous one, is based on the locally linear reconstruction errors, but in which the weights W_{ij} are fixed and the optimization is done in relation to Y_i.

$$\phi(Y) = \sum_i |Y_i - \sum_i (W_{ij}Y_j)|^2 \tag{2.2}$$

The aim of traditional matrix factorization such as Locally Linear Embeddings and Laplacian EigenMaps is the factorization of first-order proximity matrices (for example the adjacency matrix). More recent matrix factorization methods work over matrices that allow the preservation of higher order proximity, in order to embed the graph structure. GraRep [10] and HOPE [11] are good examples of the latter case. GraRep makes use of the node transition probability matrix, whilst HOPE uses a node similarity matrix, based on Katz Index and Common Neighbors similarity metrics.

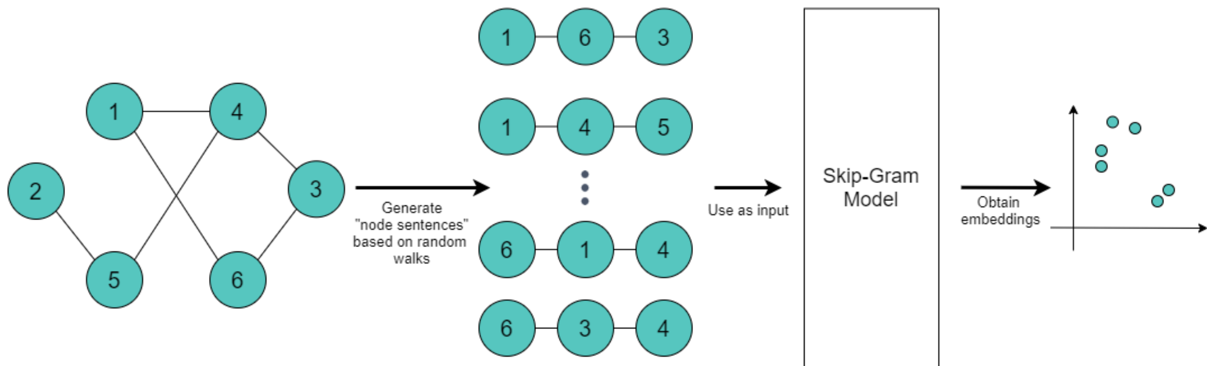


Figure 2.2: DeepWalk pipeline

Random Walk Based Methods

Random walk based methods are heavily inspired by the core ideas behind the Skip-Gram variant of `word2vec` algorithm discussed before. The core mechanism behind random walk approaches is to derive sentence-like structures from graphs, that instead of being sequences of words, are now sequences of nodes. This is done by randomly or "semi-randomly" traversing the graph to generate said sequences.

DeepWalk [12] was one of the pioneer works in the field. It is the most basic form of a random-walk algorithm. It performs M completely random N-jump walks, for every node of the graph, generating a corpus like dataset that can be used to train a Skip-Gram `word2vec` model.

Node2vec [13], appears as a flexible variant of DeepWalk in terms of the sampling strategy used. The authors state that the problem of sampling neighbours of a node can be viewed as a problem of local search. As such, they argue that at least the two extremes of local search (breadth-first search and depth-first search) should be supported. Thus their sampling algorithm allows for Breadth-first Sampling (**BFS**) whose node sequences will highlight the role of the nodes on their immediate neighbourhood, Depth-first Sampling (**DFS**) which emphasizes the position of the node on the whole graph, as well as in between sampling techniques. This feature is achieved by the introduction of two new hyperparameters p (return parameter) and q (inout parameter). The return parameter p controls the probability of revisiting the previous node of the sequence, while the inout parameter q controls the probability of transitioning to unexplored nodes.

More recent algorithms still try to improve the random walk strategy, by discovering and solving some inherent flaws of its predecessors. `Struc2vec` [1] for example solves the structural similarity conservation problem that other models have. For bigger graphs, the conservation of structural similarity between two distant nodes (distance larger than the skip-gram window) is impossible, since they will never appear in the same context (in the same node sequence). To solve this problem `struc2vec` proposes the generation and use of a metagraph, instead of working directly on the original one. The first step to construct this metagraph is to obtain a measure of structural similarity between all pairs of nodes for each possible neighbourhood size, which ranges from zero to the diameter of the graph (largest

path on the graph). After obtaining the similarity measures it is possible to generate an N-layered graph, where each layer n is a complete weighted graph and the connections between a pair of nodes are representative of their structural similarity considering an n ($n \leqslant N$) neighbourhood size. It is also a property of this metagraph that the different layers are connected by their corresponding nodes (node x on layer 0 connects to node x on layer 1, so on and so forth). With this structure it is possible to perform biased random walks, and similarly to `node2vec`, have two hyperparameters that allow us to stay in a close neighbourhood (by "restricting" the walk to lower levels of the multilayered graph, where small radius neighbourhoods are considered), or explore structurally similar nodes further away, by branching into higher level layers.

Figure 2.3 shows the results of embedding a "symmetric" graph with `deepwalk`, `node2vec` and `struc2vec`. It is made clear that `struc2vec` does a better job at keeping isometric nodes closer in the embedding space. This of course comes at the cost of a higher level of complexity compared with the aforementioned algorithms, given the necessity to calculate similarity for each pair of nodes, as well as the cost of representing and storing the multilayered graph.

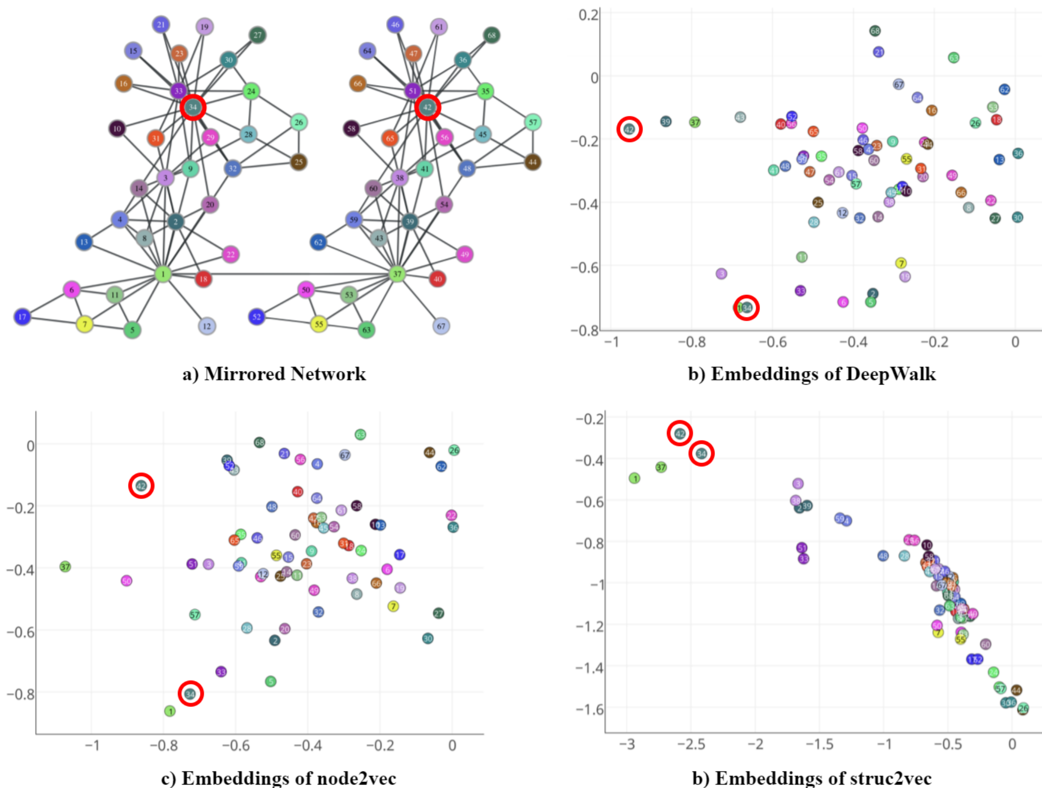


Figure 2.3: Results of different graph embedding methods on a mirrored network [1]

Neural Network Based Methods

Previous works such as `Deepwalk` or `node2vec` use neural networks to obtain the embeddings, requiring an initial processing of the data in order to obtain input for that network. Other methods uniquely based on neural networks exist that do not require complex strategies to obtain an input for the network. Many of these neural network solutions are based around a particular architecture called deep autoencoders. An example of an algorithm that adopts such an approach is the structural deep network embedding algorithm (`SDNE`) [14]. It uses a semi-supervised deep autoencoder network that preserves the second-order proximity of each node by reconstructing its original neighborhood structure passed as input. At the same time, the pairwise similarities of some pairs of nodes are used in order to adjust the embedding process in a supervised way.

2.2.2 Multi-relational network embedding methods

The embedding methods referenced until this point are designed considering networks with indistinguishable connections types between their nodes. Independently of this impediment, it has been shown previously that they do a great job at extracting information from real world networks, such as social networks and biomedical networks (for example, protein-protein interaction networks). However, there are many networks, biomedical ones included, that do not follow this simplistic architecture. These networks are multi-relational, meaning the relations established between these nodes are labeled in some way. Many of these multi-relational graphs are derived from Knowledge Bases (**KB**) and are more commonly referred to as Knowledge Graphs (**KG**) [15]. For these types of graph both the entities and relations have to be embedded. Similarly to what was mentioned until now, many strategies have been thought out to achieve this goal, each with their own advantages and disadvantages. The current existing knowledge graph embedding methods can be broadly classified into three groups [16]:

- Matrix Factorization or Tensor Decomposition Models;
- Geometric Models;
- Deep Learning Models.

Matrix Factorization/Tensor Decomposition Models

Matrix factorization methods are based on a 3-dimensional matrix $\mathcal{M}^{e \times e \times r}$, where e is the number of entities and r the number of relation types in the original graph and where each element $\mathcal{M}_{i,j,r}$ indicates the existence of a relation r between entities i and j (1 if it exists or 0 if not). The matrix can then be decomposed into a combination of lower dimension vectors that are to be used as the embeddings for both entities and relations. The different algorithms can then minimize the value of an operation involving the embeddings of a given known fact. Different algorithms use different operations to compute the scoring.

Geometrical Models

Considering the premises that an entity of the knowledge graph is a point on a d dimensional latent space, and that relations are transformations on that latent space, geometrical models adjust their representations of both entities and relations in such a way that for any known fact/triple in a training set, the transformation of the head entity of the triple given the relation of the same triple should be close to the tail entity of the triple. As such, the general scoring function of these models can be described by Equation 2.3:

$$\phi(h, r, t) = \delta(\tau(h, r), t), \text{given } \tau \text{ is a transformation type} \tag{2.3}$$

Different algorithms in this category, can use different types of transformations and/or different latent spaces in order to define their scoring function.

Deep Learning Models

The model categories presented before make use of simple operators such as multiplication or sum of embeddings so that they can be scalable and support datasets with larger sizes. However, this can come at the cost of learning less expressive representations of the entities in the graph. In these shallow models, the size of the embeddings is the only hyperparameter that can be modified in order to obtain more features (more expressive), which can in turn lead once again to scalability problems since the total number of parameters to train is given by $(num_ent + num_rel) * embedding_size$. Deep models use deep neural networks that learn a set of parameters (weights and biases), that are "global" and do not increase exponentially with the number of relations and entities of the network. This feature allows to increase the complexity of the network and learn more expressive features, without having to sacrifice scalability by largely increasing the embedding size. Algorithms that fall into this category normally differentiate themselves by the architecture of the network used.

Table 2.1 presents a division of the studied methods based on their taxonomy, as well as the general idea on how they work.

Type of Model	Model Name	Year of publication	Summary
Matrix Factorization	DistMult [17]	2015	Entities are represented as a d dimensional vector. Relation embeddings are diagonal matrices. The scoring function is a trilinear product of entities and relations. Treats all relations as symmetric.
	Complex [18]	2016	Similar to DistMult, but extends to the complex vector space. Eliminates problem of treating all relations as symmetric
	HolE [19]	2016	Does not use the trilinear product as the scoring function. Uses circular correlation. The relations embeddings have the same shape as the entities'.
	ANALOGY [20]	2017	The relation vector (r) must be a normal matrix. For each pair of relations their composition must be cumulative.
	SimplE [21]	2018	Similar to DistMult but each entity has two associated embeddings depending on their role as head or tail.
Geometry based	TransE [22]	2013	Represents relations between the embeddings of the entities as translations. Ideally: $h + r = t$ Performs poorly with one-to-many and many-to-one relations.
	TransH [23]	2014	Similar to TransE but sees relations as translations on an hyperplane. Each relation is characterized by two vectors. Allows for one-to-many and many-to-one relations.
	TransR [24]	2015	Separates the entity space from the relation space (each relation has its space). Different relations focus on different aspect of the entities involved.
	RotatE [25]	2019	Extrapolates transformations to the complex space. Relations represent rotation on this space. Allows for more complex patterns, such as symmetry, anti-symmetry, inversion, etc.

Type of Model	Model Name	Year of publication	Summary
Deep Learning	ConvE [26]	2017	Head and relation embedding are reshaped and concatenated forming a matrix. That matrix is passed by various convolutional layers and a dense layer. The result is combined (dot product) with all possible tails, obtaining a "prediction" for each tail.
	ConvKB [27]	2017	Tail embedding is also concatenated to the input matrix. Final classification is now binary (fact or not a fact).
	ConvR [28]	2019	Similar to ConvE, but does not concatenate the head and relation embedding nor it uses global filters. It constructs relation specific filters and "applies" them to a matrix obtained from the head vector.

Table 2.1: Taxonomy and summary of the studied KGE methods

2.3 Graph embedding based link prediction and applications in the biomedical field

The prediction of new interactions between entities can be said to be a desirable feature when it comes to the biomedical field, as it allows researchers to more easily generate new hypotheses of possible associations, and quickly transition to testing phases of the research. As such, there is a rising interest in using machine learning models on the existing data of the field. However, in order to take advantage of these methods, it is necessary to feed them the already existing knowledge, so that they can generate new one. A traditional way of doing this is to manually engineer a set of features that best describe the entities of the dataset in question. However, this approach comes with some problems. Firstly, the time investment and expertise needed to define such a set of features. Secondly, the fact that the entity representations obtained tends to not be very reusable, needing to be re-engineered when in a new problem context. And finally, the fact that data can require heavy data pruning, when there is a considerable amount of data entries with incomplete sets of features. The use of graph embedding methods solves the aforementioned problems since it extracts a set of latent features from the node interactions, and as such makes it easier to use biomedical datasets without having as much expert knowledge in the area.

Under the biomedical scope, several fields can benefit from the use of such graph embedding methods.

2.3.1 Pharmaceutical Field

The pharmaceutical field has a large and ever-growing industry associated with it, whose European market was valued at around 207 thousand million euros in 2017 [29]. However, the industry has large costs associated with research and development (around 35 thousand million euros in 2017 [29]). As such, new methodologies that can aid the drug discovery process, like link prediction based on knowledge graph embeddings, can be highly valuable.

Drug repurposing (or drug repositioning) is one of the areas where link prediction can be applied. It represents the concept of taking pre-existing and tested drugs and finding new uses for them. This strategy reduces not only the time to market, given that the repurposed drugs have already undergone numerous stages of testing that ensure their safety, but it also reduces the costs associated with the testing phases as well as development related costs (production structures, etc). In sum, this strategy offers a better trade of risk vs reward than other alternatives [30]. This type of analysis is usually based on drug-target and drug-disease interaction networks [31]. Drug-target interaction (**DTI**), as the name suggests, refers to interactions between different chemical compounds and biological targets on an organism (most frequently proteins), usually resulting in the change of behaviour of the latter. Networks that model such interactions can vary in complexity. The most simple kind are bipartite networks of drugs and their targets, in which there is only one type of relation and thus easily embedded by the single-relational network embedding methods. However, drug-target networks can become more complex, for example by specifying in what way the entities relate with each other (adding relation types). In this case, the network turns into a multi-relational one and has to be handled by the appropriate embedding methods.

Drug-disease interaction (**DDI**) refers to the interaction between different chemical compounds and different diseases. Similarly to **DTI** networks, they can range from simple bipartite networks, to more complex multi-relational networks, where the interactions are more descriptive, providing more information on how a certain drug affects a particular disease (for example “prevents”a “treats”, “alleviates”)[32].

Another use of link prediction under the scope of the pharmaceutical field is the discovery of new adverse drug reactions [33]. Adverse drug reactions (**ADR**), are side effects, normally undesirable, caused by one or multiple drugs. These effects are unpredictable and sometimes manifest under very specific circumstances, thus issuing the need for rigorous and numerous test phases before drugs reach the market. Graph embedding methods can be used to feed link prediction models and help with the generation of hypotheses, thus accelerating the testing phases and reducing costs.

2.3.2 Multi-omics Field

Another field that benefits from the use of link prediction is the multi-omics field. Omics refers to the studying and understanding of the structure, function, and dynamics of particular families of molecules, such as genes (genomics), proteins (proteomics), lipids (lipidomics), transcripts (transcriptomics), among others [31]. Protein-protein interaction

(**PPI**) networks are prime examples of networks that can be combined with link prediction in order to gain further understanding on the underlying molecular mechanisms different proteins are involved in. Protein-protein interactions are crucial in proteomics, given that the corroboration of certain interactions can often serve as evidence to determine the function of unlabeled proteins [34]. Two popular approaches when studying these interactions are mass spectrometry (**MS**) and yeast two-hybrid screening (**Y2H**) [34], but these methods are costly. Looking specifically at the yeast two-hybrid screening, it allows to check for the interaction between two proteins, and can cost thousands of dollars per pair of proteins tested. It is in this context that graph embedding based link prediction can help reduce the costs by generating hypotheses of probable interactions, meaning that researchers can send fewer pairs for screening.

2.3.3 Clinical field

Lastly, it is worth mentioning the medical and clinical fields as potential beneficiaries of link prediction methods. Electronic health records (**EHR**) are the digital versions of the traditional paper medical records. These allow for the reduction of medical error and delays in treatments by providing better accessibility and clarity compared to the traditional paper versions [35]. However, since this information is in most cases inserted by human hand, it is prone to contain inconsistencies and incompleteness. Given the modeling of these records into a knowledge graph structure, it is possible to use link prediction methods in order to fill in the missing information [36].

Tables 2.2 and 2.3 presents some literature related to the use of graph and knowledge graph embedding respectively on tasks of link prediction over biomedical networks.

Paper	Year of Publication	Task	Embedding Method	AUC
node2vec: Scalable Feature Learning for Networks [13]	2016	PPI prediction	node2vec	75.43% (using 10-fold Cross Validation)
Deep mining heterogeneous of biomedical linked data to predict novel drug-target associations [37]	2017	DTI prediction	DeepWalk	98.96% (using 10-fold Cross Validation)
Large-scale extraction of drug–disease pairs from the medical literature [38]	2017	DDI prediction	Expanded LINE	——
Predicting MicroRNA-Disease Associations Using Network Topological Similarity Based on DeepWalk [39]	2017	MiRNA-disease association prediction	DeepWalk	93.70% (using 5-fold Cross Validation)
Integrating node embeddings and biological annotations for genes to predict disease-gene associations [40]	2018	Disease-gene association prediction	node2vec (refined with biological annotations)	88.00% (using 5-fold Cross Validation)
Predicting miRNA-disease association from heterogeneous information network with GraRep embedding model [41]	2020	MiRNA-disease association prediction	GraRep	91.25% (using 5-fold Cross Validation)

Table 2.2: Researches that uses single-relational graph embeddings on biomedical networks

Work	Year of Publication	Task	Embedding Method	MRR	Hit@10
Inference of Biomedical Relations Among Chemicals, Genes, Diseases, and Symptoms Using Knowledge Representation Learning [42]	2019	New biomedical relation inference	TransE	17.28%	——-
Clinical Knowledge Graph Embedding Representation Bridging the Gap between Electronic Health Records and Prediction Model [36]	2019	EHR embedding	HEXTRATO (TransE variant for ontology data)	Best aprox. 57.6%	——-
Distantly Supervised Biomedical Knowledge Acquisition via Knowledge Graph Based Attention [43]	2019	UMLS graph completion	SimplE NER (SimplE based method)	33.90%	65.10%
Drug Target Discovery Using Knowledge Graph Embeddings [44]	2019	DTI prediction	ComplEx-SE (ComplEx based method)	78.00	88.00
Incorporating Domain Knowledge into Medical NLI using Knowledge Graphs [45]	2020	NLI enrichment via KG embeddings	DistMult	———	———

Table 2.3: Researches that uses multi-relational graph embeddings on biomedical networks

Chapter 3

Methods

The main objective of this work is to explore the use of link prediction methods based on KGE over biomedical networks. Such a goal is achieved by studying, applying and comparing different state-of-the-art models to a set of biomedical networks. Another objective is the creation of a simple pipeline that can serve as an entry point, for both biomedicine and computer science researchers, into this field. As such, this pipeline must provide some level of abstraction, while at the same time allowing the user to dive into the intricacies of the models used.

3.1 Technologies

Given the objectives mentioned above, and considering the time frame of development available, some open-source tools were used to help fulfill the objectives proposed:

- **Neo4J:** A pure graph database, used to store and easily modify the dataset of one of the experiments made in this work.
- **Pykg2vec:** A python library that implements a vast array of state-of-the-art knowledge graph embedding models, as well as tools that facilitate optimization and link prediction.

The following sections describe these technologies and a comparative analysis with other options.

3.1.1 Neo4J

Neo4J is a graph database that allows the storing, visualization and modification of graph-like structures. In simple terms "a graph database is a database that uses a graph structure"[46]. They are composed of two types of elements: nodes and edges. Nodes represent the entities of a dataset, for example, proteins in a **PPI** network. Edges on the other hand are the relations that connect those nodes. The main advantage of using such

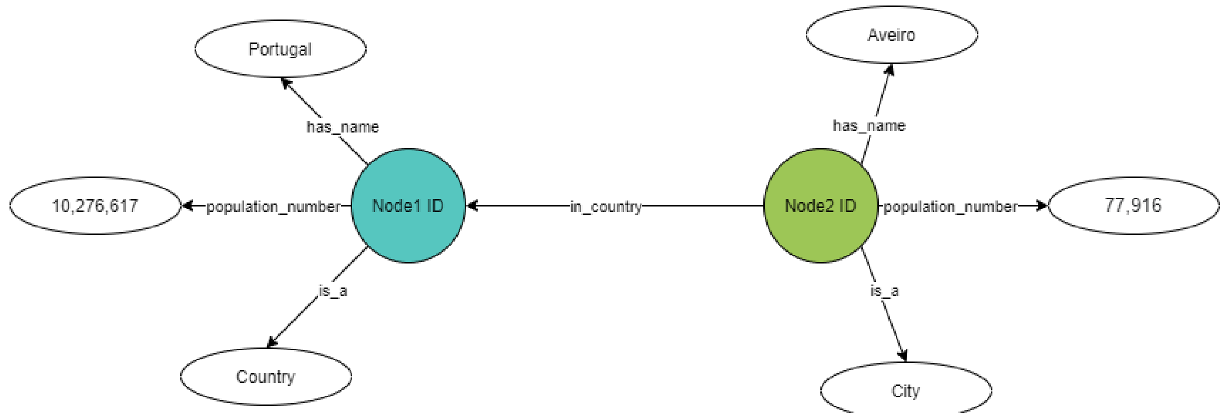


Figure 3.1: Example of a simple city and country relation on GraphDB's data model

databases when working with graph data is the abstraction they provide, which allows the user to query the data in a "graph intuitive" manner, despite the way the data is stored underneath.

Comparison with alternatives

"Graph database" is an umbrella term. There exists numerous graph databases that vary significantly in terms of their storage mechanism as well as their data model. For this work, three graph databases were explored: Neo4J, Grakn and Ontotext's GraphDB. The choice for Neo4J over the other alternatives took into consideration diverse aspects, such as query language, visualization tools, integration with other technologies, among others.

Data model Data model, or more specifically, graph data model, refers to how the databases represent the core components of the graph: nodes, relations and their properties. The three databases chosen cover different data model variants: property graphs, hypergraphs and triple stores [47].

Ontotext's GraphDB falls into the RDF Triple Store category. There are some divergences in the community on whether triple stores should belong to the graph database category. This work follows the classifications proposed by the book "Graph Databases" [48], that argues that this type of data model falls in fact under the graph database category, given they are logically linked. However, it is stated that they are to be distinguished from "native" graph databases such as Neo4J since they do not support index-free adjacency. Triplestores derive from the Semantic web movement, and as such they are built having in mind the RDF language. Their graph model is a true representation of this format. All entities are nodes and are connected with relation links. Since it does not natively support properties, it requires the explicit representation of the properties as nodes connected to the entity they are associated with (Figure 3.1).

Neo4J has a property graph data model. This means that both entities and relations can have properties associated with them, without having to have workarounds for these

Figure 3.2: Example of a simple city and country relation on Neo4J's data model

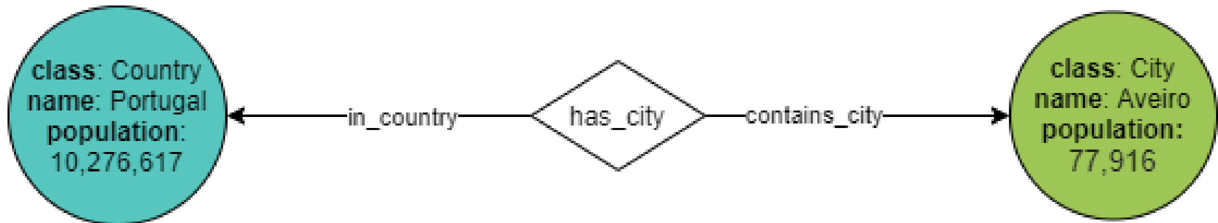


Figure 3.3: Example of a simple city and country relation on Grakn's data model

---

kinds of situations, as occurs with triplestores (Figure 3.2).

Grakn follows a hypergraph data model. A hypergraph is a special type of graph that has hyper-edges instead of the typical edges/relations seen on other graph types. Whilst the typical edge assumes there are only two end point entities (a head and a tail), hyper-edges allow relations with two or more entities by introducing an intermediary "relation node". However this forces all relations to be "bidirectional" (Figure 3.3).

In terms of data model, GraphDB and Neo4j seem to be the better choices, since their structure is much more similar to the input accepted by the studied knowledge graph algorithms, whilst Grakn's model diverges from that ideal and as such requires some data manipulation when exporting the dataset which is not ideal.

Query language Regarding their query languages, the main comparison points looked at were the simplicity, ease of learning, and the capabilities they provide (however this last point is heavily influenced by the data model).

GraphDB uses the standard RDF query language SPARQL. Given the underlying data model which does not support properties on nodes, this query language can become quite verbose.

Grakn uses their own query language `Graql`. Similarly to RDF it is very triple oriented and seems adapted for something like an OWL structure.

Neo4j also uses their own query language `Cypher`. This language is more modeled for graph traversing, it is visually intuitive, making it the most easy to read and understand of all the three query languages. On top of its simplicity, Cypher also provides a set of easily accessible built-in procedures that further simplify more complex queries.

The following snippets present a side-to-side comparison of the syntax of the different languages for querying the names of cities that belong to the country "Portugal" and have a population count higher than 10000.

```
#SPARQL:

PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX places: <http://www.example.org/places#>

SELECT ?city_name
WHERE {
?country places:name "Portugal".
?city rdf:isa places:City;
        places:name ?city_name ;
    places:population ?population .
?city places:in_country ?country.
FILTER (?population > 10000) .
}
```

```
--GRAKQL:

match
$c isa city, has population>10000, has name $c_n;
$p isa country;
(in_country: $c, contains_city: $p) isa has_city;
get $c_n;
```

```
//CYPHER:

Match (n:City)-[:in_country]->(m:Country)
where n.population>10000
return n.name
```

Ease of data integration Integrating multiple datasets is a common task when it comes to working with biomedical data. As such, it was taken into consideration, the ease of performing such a task on the given database alternatives.

On this subject Neo4J and GraphDB have clear advantages given their schema-free capabilities compared to Grakn that, like relational databases, need a schema to be set beforehand, making it harder to easily add new data from different sources without needing to change the original schema.

Visualization tools Visualizing the graph, or certain parts of it, is a valuable tool when creating or modifying a dataset. All the studied alternatives provide a client with visualization capabilities.

Ontotext's GraphDB visual graph tool (Figure 3.4) allows to see specific nodes and their relations or even subgraphs resulting from a SPARQL query.

Grakn's "Workbase" is a special tool that allows the user to connect to a running Grakn server, execute and see the results of "match" queries. This tool also allows to visualize the inherent schema of the database (Figure 3.5).

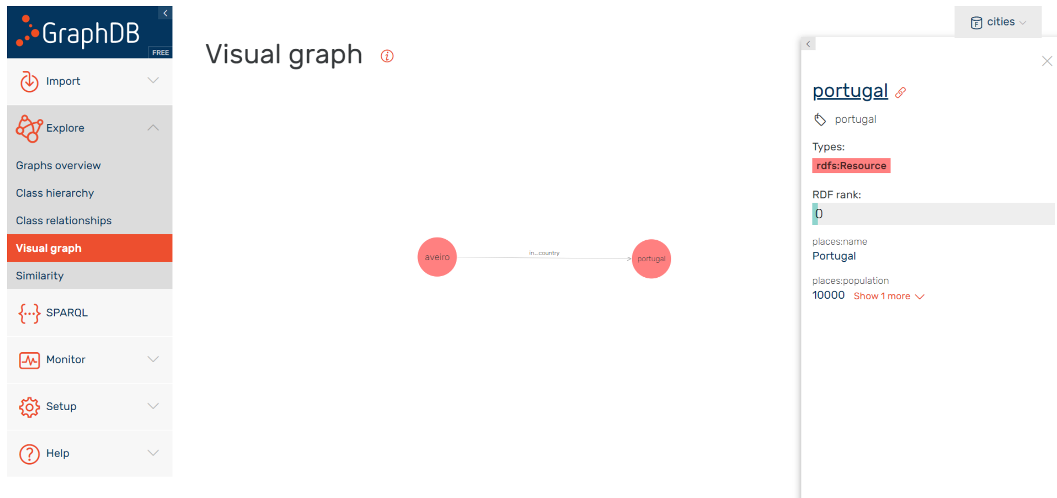


Figure 3.4: GraphDB's visual graph tool

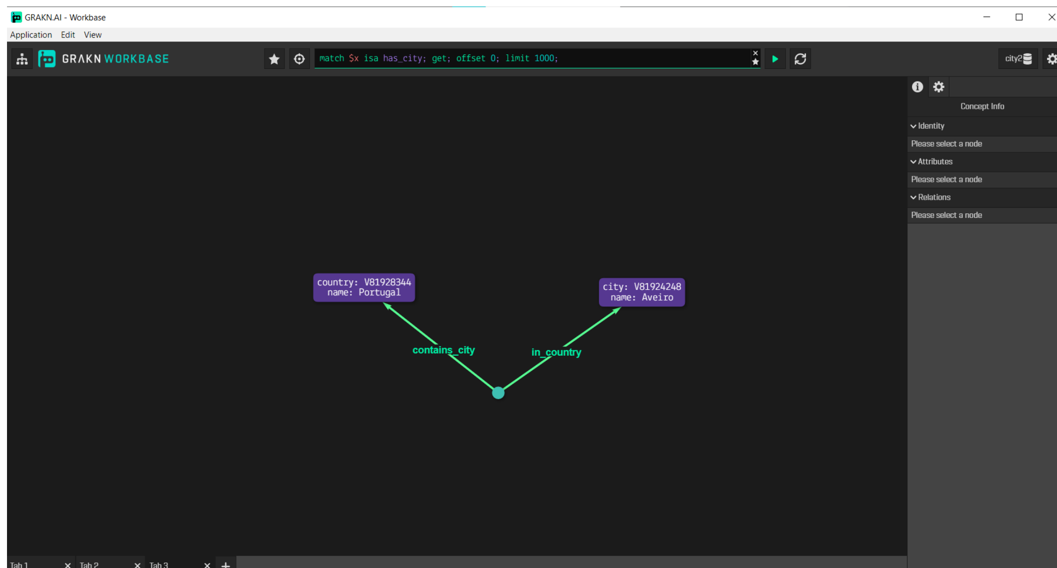
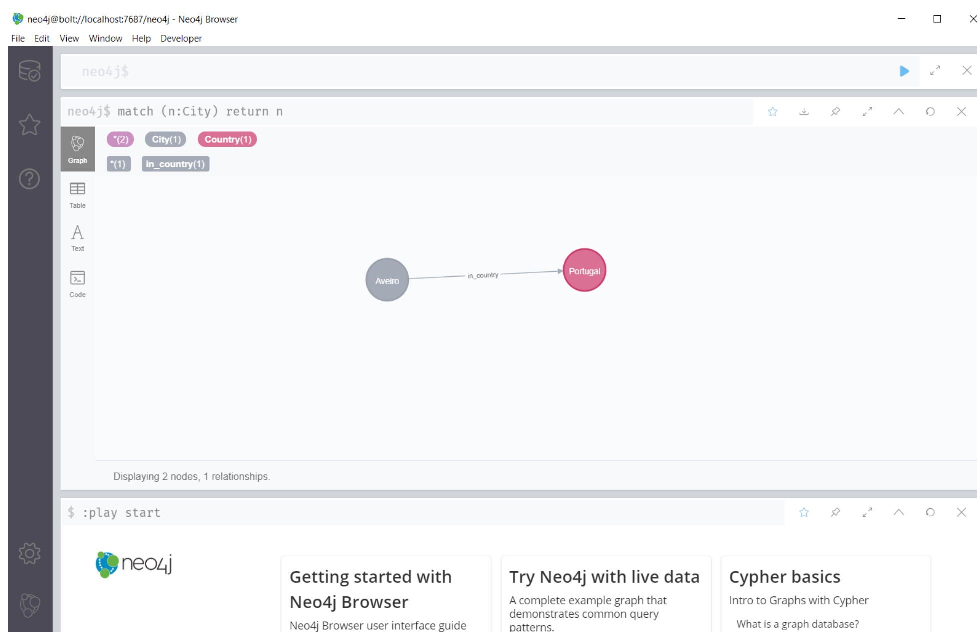
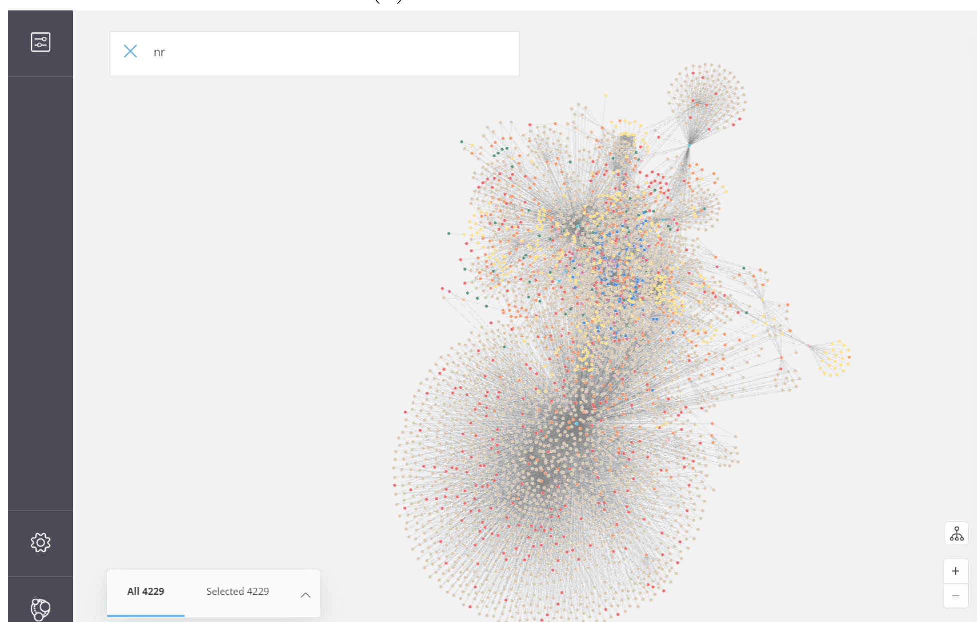


Figure 3.5: Grakn Workbase

Out of the three Neo4J's visualization capabilities have a slight edge, since its browser application functions as a Cypher console and visualization tool at the same time, allowing to visualize the result of the query executed on real time. Also, their more advanced "Bloom" visualization tool allows for the visualization of very large datasets with minimum resource consumption (Figure 3.6).



(a) Neo4J Browser



(b) Neo4J Bloom tool

Figure 3.6: Neo4J visualization tools

Discussion

Given that the objective of using a graph database in the context of this work is to load and alter pre-existing databases, Neo4J stood out as the ideal candidate, given its low learning curve, the power to write complex queries in a short and readable way as well as the similarity of its data model to the input of the used algorithms.

It is worth mentioning however that the other alternatives can be viable in different situations given their unique data models and features. An example would be Grakn and its hypergraph data model, that would have the advantage when working with ML models based on hypergraphs.

3.1.2 Pykg2vec library

Pykg2vec is a python library that provides implementations of multiple state-of-the-art knowledge graph embedding based link prediction models, being a central component in the realization of this work.

In light of the fact that the knowledge embedding field is not very mature, the number of support libraries is low. That number is further reduced when considering libraries that are not specific to one model. Three libraries were studied that adhere to these constraints: AmpliGraph [49], OpenKE[50], and pykg2vec[51].

Comparison with alternatives

Different factors were taken into consideration when choosing between the three studied alternatives.

Continued library support Taking into account the fact that the knowledge graph embedding field is still in development, with new models and variations of older models appearing frequently, it was important that the chosen library also adapted accordingly. All the three studied alternatives have good levels of support, all having seen updates in the last year.

Model diversity One of the possible byproducts of having continued support on these libraries manifests as the introduction of new models to the library. Despite this factor not being an indicator of the quality of the library, it portraits the authors' view on their idea of expansion. OpenKE seems to tend to only implement well established and baseline models such as TransE and focus more in terms of optimizing training times. Whilst the Pykg2vec approach leans more towards keeping up to date with the latest models in the field and trying to bring a working implementation of them into their library, having at the moment of writing twenty-three different knowledge-graph embedding models.

Since one of the objectives of this work is studying the results of applying different types of models to biomedical networks, as well as providing a "playground notebook" so that other researchers can also explore on their own, the approach of pykg2vec is the one that

best aligns with these objectives. However, in more business focused scenarios, OpenKE's approach of using well established and more time efficient implementations is most likely the best choice.

Time of training A small experiment was put in place in order to compare the three libraries in terms of training times. This experiment was performed under the most similar conditions possible. These tests were performed using the `12GB NVIDIA Tesla K80 GPU` (provided by the Google Colab tool), using the TransE model on the Wordnet18 dataset, with the following hyperparameter settings:

- **embedding size:** 20
- **epochs:** 100
- **batch size:** 128
- **learning rate:** 0.01
- **sampling:** uniform
- **loss function:** margin loss
- **negative rate:** 1
- **normalization:** L1
- **margin:** 2
- **optimizer:** SGD

It is also important to mention that the source code of pykg2vec had to be tinkered with, in order to make a fair comparison between these libraries. This is due to the fact that the `train` method it implements performs validation during training and ends with a full test on the test set at the end. In that regard, the validation and test steps were removed to perform this comparison, so that the closest conditions could be obtained. The results are presented in Table 3.1.

Library	Total Training Time (in seconds)	Seconds per epoch
OpenKE	253.7	2.5
AmpliGraph	366.7	3.7
Pykg2vec	882	8.8

Table 3.1: Comparison of training times of the different studied KGE libraries

In terms of training time OpenKE displays the best results. This edge in training time can be attributed to the fact that OpenKE implements heavy calculations at a lower level

using C++. Whilst the difference between Ampligraph and pykg2vec difference in training times, can result from differences between frameworks (PyTorch and Tensorflow) as well as differences in efficiency of the implementation.

Hyperparameter tuning In the field of machine learning, hyperparameters are special parameters that have a direct impact on the learning process of the corresponding model. In the case of knowledge graph embedding models, this can include: embedding size, learning rate, and many others. However, given the number of the hyperparameters the typical model has, and the range of values those can take, the search for a good combination can be time-consuming. As such, various approaches to search for such combinations were created. During the evaluation of the different libraries, the inclusion of hyperparameter search algorithms were considered as a positive bonus. In that aspect both `AmpliGraph` and `Pykg2vec` presented built-in tools to perform hyperparameter search. Ampligraph implements grid search and random search. Grid search is a simple hyperparameter search algorithm that performs an exhaustive search given a subset of options for each hyperparameter, making it very time-consuming if the training process is long. On the other hand, random search, randomly selects a set of parameters given a search space. Despite this randomness, it has been shown that this method can find sets of hyperparameters that result in models that preform similarly to the one found by grid search in a smaller amount of time [52].

Pykg2vec uses a probabilistic search algorithm instead, more concretely Bayesian optimization. This method allows to further reduce the search times, by minimizing the number of calls to the training process, which is the time bottleneck of the two previous methods. It does this by generating and updating a probabilistic model based on past calls to the training function, that closely mimics the real model aiming t´´o find best hyperparameter combinations in less time [53].

Given the training times of the used models in this work, the inclusion of these tools in the libraries was factored in when choosing from the libraries. However, it is to note that these search algorithms can be implemented in any library if needed.

Ease of extending One other aspect that was taken into consideration was how easy it was for developers to modify the library, for purposes of implementing new models, or simply adding or modifying tools the developer might need. OpenKE is the least flexible model when it comes to this aspect. This derives from the fact that it implements some of its operations at a lower level using C++, requiring the developer to have a much deeper understanding of the training process implementation, as well as demanding that the developer understands the compilation tools in place. On the other hand, Ampligraph and pykg2vec are programmed exclusively in python, providing well defined API structures easy to extend and modify.

Available Documentation Lastly, it was taken into account the quality of the documentation offered by each library. In this aspect, OpenKE shows the most problems, since

the only documentation provided in their official website is relative to an old version that has many dissimilarities to the recent release. This old version can still be found on their Github repository however it is discontinued. In terms of pykg2vec and Ampligraph, both have available documentation that is updated and detailed. However, it can be argued that Ampligraph's documentation is better structured and provides a bit more insight on their implementation.

Discussion

In the end, pykg2vec was the chosen library to use for building the pipeline. This choice stems from their exploratory philosophy (given the diversity of models available) which, as mentioned earlier in the "Model diversity" subsection aligns with the "educational" purposes of this work. This factor is translated into the implementation of the latest models, the flexible nature of their architecture, and tuning tools made available. However, such benefits come at the cost of time performance compared to OpenKE for instance.

3.2 Embedding and link prediction

In order to obtain good embeddings of the entities and relations of some dataset, it is necessary to follow a pipeline of steps. This pipeline starts with the selection of which models to use, followed by the preparation and loading of the data to embed, the optimization phase, the training phase, and finally the testing and metric collection phases. In this section, the different stages of the embedding process will be discussed, explaining the choices taken along the way.

3.2.1 Models used

As mentioned before, the first step to be taken is to decide which model to use in order to obtain the embeddings. The strategy adopted in this work was to use a range of models that cover the different families of models available, as such a baseline model was chosen from each of those families (TransE, DistMult and ConvE). This allows for an understanding of the impact that different strategies can have on the results of the different experiments made.

TransE

TransE [22] was the first **KGE** model to introduce a geometric interpretation of the latent space. The core idea behind this approach is that, for a given true fact/triple, and assuming that relations can be seen as translations on the embedding space, the result of applying such a transformation to the head of the known fact should be geometrically close to the tail. On the other hand, given a non-observed fact/triple, this should not occur. The geometrical closeness is obtained using a dissimilarity measure function d that takes the form of either L1 or L2 norm according to the original paper. Lastly, in order for the

model to learn the embeddings, the authors propose to minimize a pairwise margin-based loss function (Equation 3.1). This requires a set of known triples S and a set of corrupt triples S' obtained by replacing the head or tail by a random entity on known training triples.

$$L = \sum_{(h,r,t)\in S} \sum_{(h',r,t')\in S'} [m + d(h + r, t) - d(h' + r, t')], \text{ m is the margin hyperparameter} \quad (3.1)$$

Minimizing this function will ideally lead to the distance between $h + r$ and t to be close to zero, whilst the distance of $h' + r$ to be close to the margin value m (hyperparameter that can be changed). It is to note that given the relatively simple approach, the performance of TransE can be affected by the dataset it is applied to. A known problem of this model is connected to the rudimentary transformation it considers, leading to performance losses when dealing with one-to-many and many-to-one relations [23]. Looking at two triples such as `<Universidade de Aveiro, located_in, Aveiro>` and `<Glicinias, located_in, Aveiro>`, using TransE to obtain their embedding will result in them having very similar representations despite being very different entities.

DistMult

DistMult [17] can be argued to be one of the baseline models when it comes to tensor decomposition, given many other models derive from the concepts introduced by it. It forces relation embeddings to be diagonal matrices (of dimension $k \times k$, where k is the embedding size hyperparameter). This is done to allow the scoring function to be computed as a trilinear product, as shown in equation (3.2).

$$\phi(h, r, t) = \langle h\ r\ t \rangle = h^T\ diag(r)\ t \quad (3.2)$$

However, it can be seen that this scoring function is commutative, meaning it is the same for (h, r, t) and (t, r, h), which in turn means that all relations are considered as symmetric, which could be false and problematic in some datasets.

ConvE

Convolutional neural networks are a popular type of deep model, mainly encountered in the image classification field. The core idea behind them is to use several convolutional layers (hence the name) that apply a set of low-dimensional filters w to the input data. The product of such a stack of convolutional layers is a feature map that can be passed through a number of dense layers to obtain a fact score. One of the advantages of these types of networks is that, comparatively to the traditional deep fully connected architectures, convolutional networks require a lower number of parameters, making them less prone to over adapt to the data (overfit), as well as making them faster to train. As such, the authors of ConvE found this type of architecture to be a good middle ground to both the expressiveness and overfitting problems. ConvE is the simplest multilayer convolutional

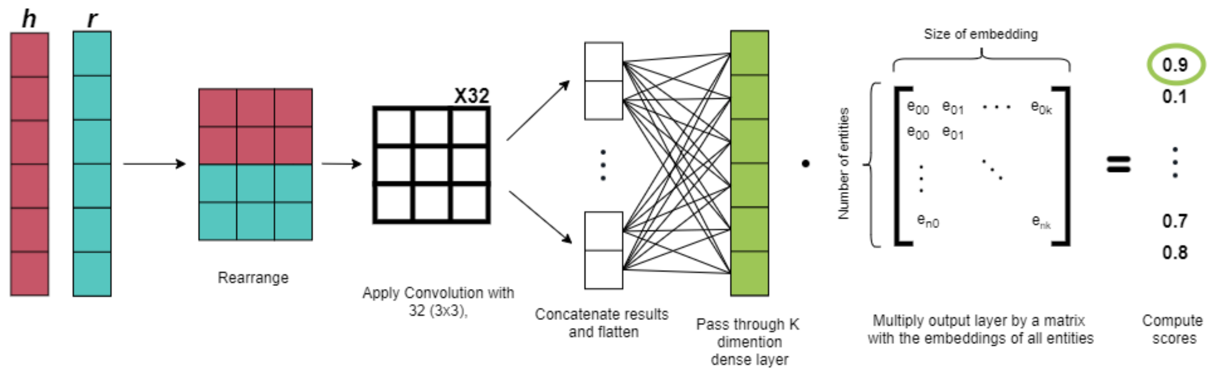


Figure 3.7: ConvE's architecture

architecture, using a single two-dimensional convolutional layer [26]. In order to calculate the score of a given triple, ConvE generates a two-dimensional matrix by reshaping and stacking the embedding of entity one with the embedding of the relation. That matrix is then used as the input for the two-dimensional convolutional layer. This process results in a feature map that is flattened and passed through a dense layer with k neurons. The score is finally calculated by applying the dot product between the output of the dense layer (a vector of dimension k) and the embedding of the second entity of the triple.

It is to note that in this architecture dropout layers are used as a form of regularization to further avoid overfitting scenarios. A general view of the architecture is shown in Figure 3.7.

3.2.2 Data preparation

Following the decision of which models to use, it is necessary to prepare the data before passing it into said models for training. As mentioned in the technologies section, the KGE library chosen for this work was pykg2vec, and as such, the format in which the data is passed to it, has to obey its restrictions. Pykg2vec requires user defined datasets to be in the form of three `.txt` files (`<DatasetName>-train.txt`, `<DatasetName>-valid.txt` and `<DatasetName>-test.txt`), each of them having one triple/fact per line with tab-separated values (TSV) ie. `<head entity>\t<relation>\t<tail entity>`. It is also worth noting that all of these files need to be under the same directory.

3.2.3 Training pipeline

After having the data prepared and having decided the models to use, it is possible to start the training process. The approach taken to obtain the embeddings and a model to perform link prediction based on them can be summarized with the pipeline presented in Figure 3.8.

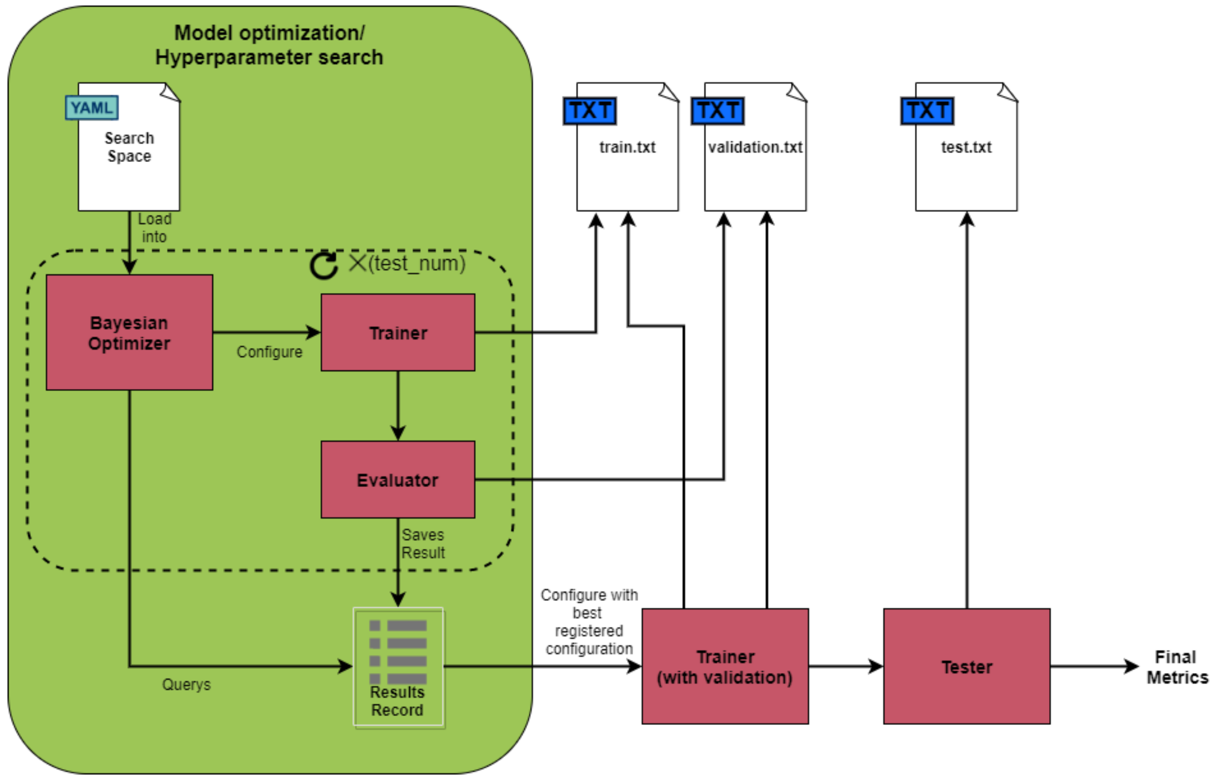


Figure 3.8: Pipeline used in order to obtain, train and test the models used

3.2.4 Training Methods

In Figure 3.8 it is possible to identify two different trainer components. Despite having very similar roles, they vary slightly in their implementation.

In machine learning, training refers to the process of providing the learning model with training data from which this can learn. During this training process, the model is shown the entirety of each training example, such that it can compare its current predictions with the truth (based on some loss function) and adjust its parameters/weights accordingly. In the case of KGE models, these parameters are mainly the embeddings of the entities and relations present in the graph. However, some models such as ConvE and ConvKB, which are built using convolutional neural networks as well as fully connected layers, have additional parameters to tweak.

Out of the box pykg2vec offers two alternative training processes, the first one being slower but more complete. In this case, besides updating the model parameters based on the training set and the values of the loss function across a certain amount of epochs, it also periodically evaluates the current model against a validation set which contains triples that the model has not seen before. This validation allows the training process to understand if the model is still generalizing, by verifying if the decreasing values of loss it observes correspond in fact to gains in performance on unseen data, or if otherwise the model is adapting to the training data, which is commonly referred as overfitting. In the

case this training process detects in fact a decreasing performance on the validation set over several epochs, this stops the training process early (early stopping). This process variant is described by the `train_model()` method. Other than training the model this method also tests the model at the end against the testing set. In order to complement the functionalities already provided by this training method, a `Tensorboard` logger was implemented in order to allow for the visualization of the training process. The second variant of training process that the library offers does not perform successive validations, it does not implement an early stopping mechanism and it does not perform a test at the end. The main objective of this training process is to be as fast as possible. The method that implements this type of training process is the `tune_model()` method, that given its characteristics is used in the trainer inside the hyperparameter search stage of the pipeline. Independently of the used method, it is necessary to configure some parameters so that the training process can occur. This ranges from general variables, such as the name of the model, or the path to the directory the dataset is located in, to the specific hyperparameters of the model in question. Below, a list of global configuration parameters and model hyperparameters relevant for this work is presented.

Relevant Global Configuration Parameters

Model Name (mn): As the name suggests, it is the name of the model the user intends to use on the dataset.

Experimental Setting (exp): This parameter is a flag that indicates if the user wants to use their own set of hyperparameters to train the dataset (true), or if he pretends to use the default hyperparameter values (available for specific datasets such as FB15k).

Dataset Name (ds) and Dataset Path (dsp): These arguments describe the name of the dataset to use. In case it is a user defined dataset (not included with the library), the path where the train, test and validation files reside.

Number of test triples (tn): Despite the name, this refers to the number of triples to use in the validation steps (referred to as mini-tests in the source code).

Test step (ts): It defines the epoch periodicity in which validation (mini-test) is performed, (i.e. if test_step = 10, every 10 epochs validation is performed).

Hyperparameter Absolute Path (hpf): The pykg2vec library allows the user to define the hyperparameters of the models in YAML files. This configuration parameter defines the path where this YAML file resides.

Device (device): This configuration variable tells the library if the user pretends to make use of the GPU to train the model. It can take the value of "cuda", if the GPU is to be used, or "cpu" if the user pretends to only make use of the CPU. This variable can affect the training times considerably.

Relevant Model Hyperparamters

When it comes to hyperparameters, despite the configuration object (`Config`) being model agnostic, not all hyperparameters are used by every model. This factor can be seen as a poor implementation choice, since it requires the user to have some prior knowledge of the models used in order to not "waste" time over-configuring. The following hyperparameters presented are the ones used at least by one of the used models.

k (hidden size): `k` represents the hidden size, which in other terms means the desired size of the output embeddings. Normally, and especially in shallower models, larger embeddings can allow obtaining more expressive embeddings, leading to better link prediction performances, but on the other hand larger embeddings lead to larger training times, hindering the scalability to larger datasets.

epochs: In machine learning an epoch corresponds to the passing of the entire training dataset through the model. The more epochs the model trains for, the better it will usually get. However, this gain in performance is not infinite, the model will reach a plateau in its training, in which giving it more epochs to train will not result in gains in performance. In the worst-case scenario, it can even start to overfit to the training dataset.

opt: `opt` refers to the optimizer algorithm used to find a minimum in the loss function. This optimizer depends on other hyperparameters such as the learning rate. The choice of optimization algorithm can have a great impact on the results obtained from training.

batch_size: The batch size refers to the number of training examples (in this case triples) that are passed through the model before updating the internal its weights/parameters. This batch size can range between two extremes. On the one hand, having the batch size of one, meaning that the new values of the weights are calculated after each triple of the training dataset is passed through the model. This will inevitably turn the process slower. Using this approach will also introduce instability in the convergence of the loss function into a minimum, given that less information is given on the best direction it should proceed. However, this instability is not entirely undesirable, since it allows escaping non-optimal shallow valleys in the loss function. On the other hand, it is possible to have the batch size equal to the number of training examples, this makes the process faster, but requires a large amount of memory as a trade-off, it is also more prone to be stuck in a locally optimal solution. It is a best practice to settle on a middle ground, where some stability and time can be gained, while being able to find the best loss minimum possible. As such, the best value can be found with a trial-and-error approach.

lr: `lr` refers to learning rate. As mentioned before, the learning rate is a hyperparameter that controls the process of optimization. It determines the step size taken when moving towards the minimum. When choosing the learning size, it is important to have into account the effects it can have on the task of achieving the best minimum possible. Too high a learning rate can lead to overshooting, meaning that the step taken is so large, that the minima is passed over. On the other hand, too low a learning rate makes the learning process slower and can lead to the training process to get stuck on a local minimum.

neg_rate: The negative rate dictates the number of corrupt triples generated for each positive triple. If the negative rate is one, it means that for each positive triple a negative one is generated.

- **The previous hyperparameters are relevant to the following models:** TransE, DistMult, ConvE

sampling: It refers to the strategy to be used when choosing which part of the triple to corrupt. It can be either "uniform", in which case the probability of corrupting either the head or tail of the triple is 50%. Or it can be "bern" in which case a Bernoulli sampling strategy is adopted. Exceptionally it can take the value of "adversarial_negative_sampling" when dealing with the RotatE algorithm, that is a sampling method proposed by the authors of that model, and which they argue boosts the model efficiency.

- **The previous hyperparametes is relevant to the following models:** TransE, DistMult

margin: This parameter is used on models implemented using a margin based loss function, in this case TransE and RotatE. It represents the distance that corrupt the triple should have between the projection of the head embedding given a relation and the tail embedding. The higher the value, the further away they are from each other.

l1_flag: If the flag is "true", L1 norm (also known as Manhattan distance) is used when calculating the distance between $h + r$ and t in TransE; if "false", L2 norm (Euclidean distance) is used instead. Figure 3.9 shows the intuition behind both of these strategies.

- **The previous hyperparameters are relevant to the following model:** TransE

lmbda: The lambda parameter is the value that controls the level of regularization applied to some models.

- **The previous hyperparameter is relevant to the following model:** DistMult

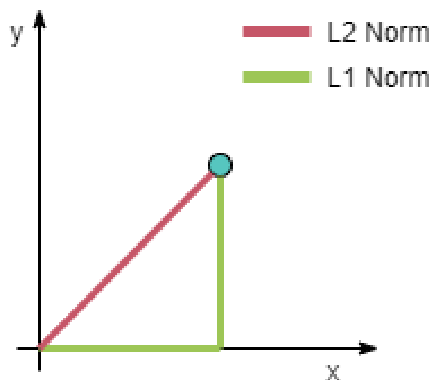


Figure 3.9: L1 vs L2 norm

---

hidden_size_1: As mentioned previously, ConvE reshapes and concatenates the head and relation embedding to obtain a matrix to pass as input to the convolutional layer. The `hidden_size_1` influences this reshape. Giving a more concrete example, in case that the embeddings of both the head and relation are vectors with size 1×100 (k=100), and considering a `hidden_size_1` of 10, a matrix of shape $(1 * hidden_size_1) \times (\frac{100}{hidden_size_1})$ will be the input of the convolutional layer. As such, it is imperial for the `hidden_size_1` to be a divisor of `k`, or it will create problems with the shape of the input matrix.

input_dropout, feature_map_dropout and hidden_dropout: This set of parameters defines the probability of dropout in different stages of the network. Dropout is a common regularization technique used on neural networks to prevent overfitting. The main idea behind dropout is that by randomly dropping units of the network, this is more prone to generalize [54]. In the case of ConvE the authors propose the use of dropout at the input matrix level (dropping some units of the matrix), at the feature map level (dropping some of the 32 feature maps) and at the fully connected hidden layer level (dropping some neurons).

label_smoothing: Label smoothing is a regularization technique used in multi-class neural networks to avoid overconfidence (meaning the predictions are consistently higher than the real accuracy).

- **The previous hyperparameters are relevant to the following model:** ConvE

As mentioned before, a practical feature made available by the library is the ability to store configurations of various models, for various datasets in a YAML file and load the model configuration programmatically from those files. An example of a TransE configuration YAML file for the FB15K (freebase15k) and WN18 (wordnet18) datasets is presented below:

```
# TransE.yaml
model_name: "TransE"
datasets:
  - dataset: "wordnet18"
    parameters:
      learning_rate: 0.01
      l1_flag: True
      hidden_size: 20
      batch_size: 128
      epochs: 1000
      margin: 2.00
      optimizer: "sgd"
      sampling: "uniform"
      neg_rate: 1

  - dataset: "freebase15k"
    parameters:
      learning_rate: 0.01
      l1_flag: True
      hidden_size: 50
      batch_size: 128
      epochs: 1000
      margin: 1.00
      optimizer: "sgd"
      sampling: "bern"
      neg_rate: 1
```

Running the trainer

To summarize, in order to train a model with pykg2vec, first the configuration has to be prepared and then passed to the model. The library offers an `Importer` object, that given an algorithm name, returns the necessary configuration and model class objects. It also makes available an argument parser that given a set of arguments generates the necessary objects to pass to the configuration object. Finally, it is only necessary to build the trainer with the model and configuration objects and start training.

The complete process can be done with the following code below:

```
from pykg2vec.common import Importer, KGEArgParser
from pykg2vec.utils.trainer import Trainer

#Define arguments
args = KGEArgParser().get_args(['-mn', <model_name>, '-ds',
<dataset_name>, "-dsp", <dataset_path>,"-tn" ,0 ,"-device", "cuda"])
#OR
#args = KGEArgParser().get_args(sys.argv[1:])


#Obtain config and model objects
```

```
c_def, m_def = Importer().import_model_config(args.model_name.lower())
config = c_def(args)
model = m_def(**config.__dict__)

#Build and start training
trainer = Trainer(model, config)
trainer.build_model()
trainer.train_model()
```

3.2.5 Hyperparameter optimization

In order to get the most out of the chosen model, it is necessary to find a set of hyperparameters that maximize its performance. A common practice to accomplish this goal is to iteratively test out different sets of hyperparameters and, at the end, choose the set that gives the best performance.

The most simple way of doing this is using the `Grid Search` method. This consists of creating a list of possible values for each hyperparameter and testing all the possible combinations of those values. However, this method can be very costly in terms of time since the number of possible combinations can grow exponentially. For example, given a set of 6 hyperparameters, if a user defines 3 possible values for each of them, that would represent a total of 729 (3^6) possible combinations. In order to counter such an effect, a `Random Search` method can be applied to reduce the number of combinations tested. This method randomly picks from the possible values for the different hyperparameters to create a defined number of sets to test. While it is possible this method may miss the optimal combination of hyperparametets when compared `Grid Search`, it has been proven that for a large number of combinations this method can find good models within a fraction of the time of `Grid Search` [52]. Pykg2vec however provides an even more sophisticated method of searching for the optimal model, `Bayesian Optimization`. In contrast to the previous methodologies, it is based on probabilistic models in order to try to reduce the number of calls to the training process.

It makes use of a surrogate model [55], which is a probability model tries to mimic the objective function (the training and validation process) based on prior calls to it. The main advantage is that the surrogate function is computationally cheaper than the true objective function. As such, it is possible to more efficiently evaluate larger search spaces and within smaller amounts of time. This process can be summarized in the following steps:

1. Generate initial surrogate model
2. Find the combination of hyperparameter that best perform on the surrogate model
3. Test this set of hyperparameters on the objective function
4. Update surrogate model taking in consideration the metrics obtained

5. Repeat from steps 2-4 until the number of predefined test steps is achieved

As mentioned before, pykg2vec uses this method, more concretely the Tree-structured Parzen Estimator (**TPE**) variation, which has a specific surrogate model. Internally the library makes use of the implementation provided by the hyperparameter optimization library `hyperopt` [56]. The optimization tools provided by pykg2vec can be found under the `BaysOptimizer` class.

The initialization of this class requires a set of arguments, as for the `Trainer` class. The arguments that are needed for the `Trainer` class, with the exception of the hyperparameter file path, are also required for the `BaysOptimizer`, since this class needs to instantiate a `Trainer` object. Furthermore, an additional set of arguments can be configured.

Additional arguments used in BaysOptimizer

Maximum number of trials (mt): Represents the maximum number of tests performed on the objective function before returning the best set of hyperparameters found.

Search space file (ssf): The path to a YAML file containing the possible values of each hyperparameter of the model to be optimized. Those values can be in the form of a range, in which case the Bayesian algorithm picks a value within that range according to a defined distribution (for example, normal or log-normal). Those values can also be a list of "choices" instead of a range, and in this case the value picked is a random value from this list of options. An example is shown below.

```
model_name: "TransE"
dataset: "freebase15k"
search_space:
  learning_rate:
    min: 0.00001
    max: 0.1
  l1_flag:
    - True
    - False
  hidden_size:
    min: 8
    max: 256
  batch_size:
    min: 8
    max: 4096
  margin:
    min: 0.0
    max: 10.0
  optimizer:
    - "adam"
    - "sgd"
    - "rms"
  epochs:
    - 100
```

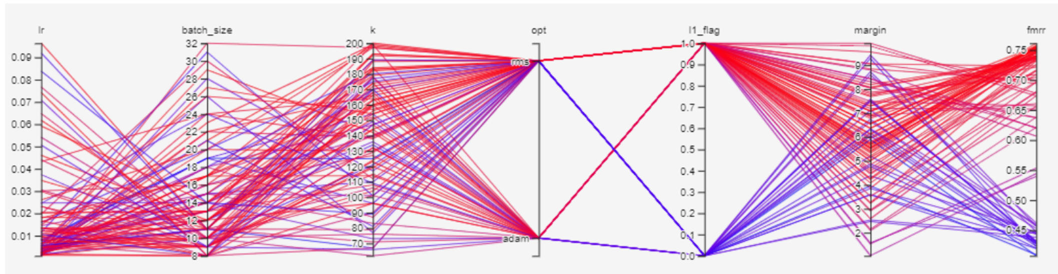
Running the optimizer

Given the arguments, it is possible to create the `BayesianOptimizer` object to start the optimization process. This can be easily done by calling the `optimize()` method shown in the code snippet below:

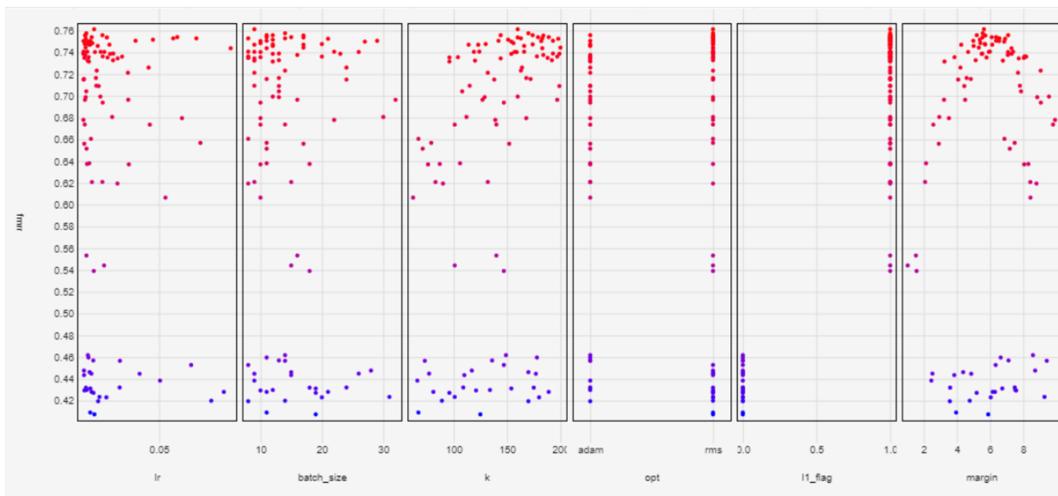
```
from pykg2vec.common import KGEArgParser
from pykg2vec.utils.bayesian_optimizer import BaysOptimizer

#Define the arguments
args = KGEArgParser().get_args(['-mn', <model_name>,
'-ds', <dataset_name>,"-dsp",<dataset_path> ,
"-ssf", <search space file path>, "-mt", "100","-tn",0,"-device", "cuda"])
#OR
#args = KGEArgParser().get_args(sys.argv[1:])

#Create Optimizer object
bays_opt = BaysOptimizer(args=args)

#Start optimization
bays_opt.optimize()

#Obtain the best hyperparameters
best = bays_opt.return_best()
```

However, in this work the `optimize()` method as well as the methods involved in the optimization function called by it, `_get_loss()` (wrapper that configures the "Trainer" object) and most notably `tune_model()` (mentioned in the "Training" subsection) were not used, given that their implementations did not align with the practices followed in this work. The main problem found in the implementation was the fact that the authors of pykg2vec used the training loss as the optimization objective passed to the Bayesian optimizer, while such metric would be able to identify sets of hyperparameters that presented the apparent best performance, it is not a good practice to do so. The main reason behind that is the fact that training loss is blind to the model generalization, meaning that it does not factor the possibility of the model overfitting to the training data. While this can be a lesser problem in some shallow models such as TransE, deeper models like ConvE tend to easily overfit, especially with small datasets. As such, new methods were implemented to substitute the existing ones (`optimize_mrr()`, `_get_imrr()` and `tune_model_mrr()`). The main point of difference being the use of the validation filtered mean reciprocal as the metric to optimize instead of the training loss. The choice to use a validation metric has the purpose of helping prevent the optimizer from choosing sets of hyperparameters that lead to overfitting.

Taking advantage of the fact that the method was being re-implemented, a visualization tool based on `Tensorboard` that registers information about the results of each run of the optimizer was included. This tool has the goal of helping researchers figure out the impact of the different hyperparameters on the performance of the model so that they can adjust the searching space accordingly (Figure 3.10).



(a) Parallel coordinates view of the hyperparameters in relation to the validation fMRR



(b) Scatter plot view of the hyperparameters in relation to the validation fMRR

Figure 3.10: Tensorboard tool for visualizing effects of different sets of hyperparameters

3.2.6 Testing and Metrics

As mentioned in the "Trainer" subsection, the `train_model()` method, used as the final training when the hyperparameters have already been chosen, performs a test using the test set. This test set is not seen by the model during training, and as such is a good representation of the actual performance of the model.

In the knowledge graph embedding literature, the measuring of performance of link prediction models is usually made based on three categories of metrics: mean rank, mean reciprocal rank and hits@X. All of these metrics are based on ranking. During the testing phase, each test triple is passed incomplete to the model by hiding one of the entities of the triple at a time, $< h, r, ? >$ in the case of tail prediction and $<?, r, t >$ in the case of head prediction. The model then returns a ranking of entities (with size equal to the number of entities existing on the original graph), built based on the level of confidence the model thinks said entity fits into the incomplete triple. The ideal scenario is for the target entity to be the prediction with the higher score (higher rank). However, there are two variants when it comes to computing the rank of the target, given the existence of other valid predictions fro which the model attributed a higher confidence level: raw and filtered.

Raw vs Filtered Ranking

To better understand the difference between these two scenarios, the following example prediction will be used:

- **Test Triple:** <University of Aveiro, located_in, Aveiro>
- Results of head prediction (<?,located_in, Aveiro>)
  1. <Glicínias, located_in, Aveiro>
  2. <Belém Tower, located_in, Aveiro>
  3. <University of Aveiro, located_in, Aveiro>

Raw Ranking With this type of ranking, the ranks are attributed as is, which means that valid predictions that score better than the target one still contribute to lower the ranking of the target. Looking at the example prediction proposed above, the "University of Aveiro" head prediction would be attributed a rank of 3 (third place) despite the fact that the first prediction made by the model is in fact true (being true implies being present in the original dataset before splitting).

Filtered Ranking On the other hand, the filtered ranking strategy, would attribute rank 2 (second) to the "University of Aveiro" head prediction, since in this strategy only the incorrect predictions (not present on the dataset) are used to lower the ranking of the target prediction. It is common in the literature to focus on this type of ranking instead

of the raw one, since this way the model is not being penalized for making other correct predictions.
Given these two ranking strategies, all the metrics used will have their raw and filtered counterpart.

Mean Rank (MR)

Mean rank as the name suggests is an indicative of the average rank in which the target predictions are placed by the model. Given a list containing the ranking of all the predictions made by the model on the testing data R, the mean rank is obtained as following:

$$MR = \frac{1}{|R|} \sum_{r \in R} r \tag{3.3}$$

A lower MR is correlated with a better model at predicting links. This metric is however unreliable given it is sensitive to outliers (a bad prediction that ranks the target in last place drastically increases the value of MR). As such, Mean Reciprocal Rank (MRR) is more commonly used as a more reliable alternative.

Mean Reciprocal Rank (MRR)

As mentioned above the MRR metric is a more stable variant of MR, the main difference is that it uses the inverse of the rank instead of the actual rank, which makes it more resilient to outliers.

$$MRR = \frac{1}{|R|} \sum_{r \in R} \frac{1}{r} \tag{3.4}$$

Its values vary from 0 to 1, and values closer to 1 indicate a better model at predicting missing links.

Hits@X

The hits metric gives the ratio of test predictions where the rank of the target was equal or lower than the X threshold. It is common to see X taking the values of 1,3,5 or 10.

$$Hits@X = \frac{r \leqslant X}{|R|}, r \in R \tag{3.5}$$

3.3 Limitations of the methodology used

As mentioned in the technologies section, the pykg2vec library has its own share of shortcomings, most notably the increased training time compared to the other options. In terms of the training methodology adopted, the main shortcomings are attached with time

constraints that such iterative methods bring. Despite the Bayesian optimizer being one of the most time efficient hyperparameter optimization strategies, finer searches still involve human interaction in order to verify that the training process occurs as expected, and alter the search space accordingly in order to obtain the best possible models. In some bigger datasets this process can prove to be time-consuming.

Chapter 4

Experiments and Results

Given the definition of the methodology to adopt, a sequence of experiments were executed in order to test the initial hypothesis that link prediction based on multi-relational graph embedding could be a positive asset when dealing with different biomedical datasets. In total, three experiments were put in place. The first of them was done over the semantic network of the "Unified medical Language System"(UMLS) [57], while the second and third experiments were done on different variations of the "Comprehensive Antibiotic Resistance Database"'s (CARD) "Antibiotic Resistance Ontology" (ARO)[58].

4.1 Experimental playground

The models used in the experimental setup (TransE, DistMult and ConvE) are computationally costly models. As such, it was necessary to have an environment in which such models could be run under an acceptable time frame. Pykg2vec allows for the use of a graphical processing unit (GPU) in order to drastically decrease the training times of the models it makes available. These gains on performance are however tied to the capabilities of the GPU. For this work, Google's Colaboratory tool was used. This tool, most commonly referred to as Google Colab is a free cloud service hosted on Google servers that allows the user to run "python notebooks", and to interact with the underlying Linux machine. The most notable feature of this tool that led to its utilization was the fact that it provides a `12GB NVIDIA Tesla K80` GPU, which far outperforms the other systems that were available for this research. The caveat, is that it can lock running times depending on the usage frequency which influenced some of the decisions made on each experiment.

4.2 Experiment One: UMLS' Semantic Network

4.2.1 Dataset

The Unified Medical Language System (UMLS) [57], is a well established compendium of multiple biomedical vocabularies. Its main selling point is the provision of a uniform

and seamless mapping between terms from different sources, which allows for an easier interoperability between clinical coding systems.

The UMLS is composed of three main components:

- **Metathesaurus:** The core of the tool, which contains said collection of terms from different biomedical vocabularies mentioned above as well as their relationships.
- **SPECIALIST Lexicon:** A lexicon of English terms as well as specialized biomedical terms present in the Metathesaurus. It includes syntactic, orthographic, and morphological information about the terms.
- **Semantic Network:** A network of the categories of the terms in UMLS, and the relations between them.

The semantic network of UMLS was chosen as the target dataset for this first experiment. This network was not only chosen given the fact that it is easily accessible but also because other works in the field of knowledge graph link prediction use this dataset as a baseline. This is relevant given that it allows for the comparison of the chosen models with other link prediction methods, based on graph embeddings or not. Another benefit of such fact is that it can serve as validation for the chosen methodology, most prominently the hyperparameter search approach taken, as well as validation for pykg2vec's implementation of these same algorithms, since other works used report their obtained performance using their implementation of some of these models ([59], [26]).

The pykg2vec library offers access to this dataset out of the box. The characteristics of the dataset are the following:

- **Number of entities:** 135
- **Number of relation types:** 46
- **Total number of triples:** 6529
- **Number of triples on the training set:** 5216
- **Number of triples on the validation set:** 661
- **Number of triples on the test set:** 652

4.2.2 Hyperparameter search and training of the final model

The first step to begin the experiment was to explore the dataset and understand how the tweaking of the different hyperparameters of each model affected the learning process. For such a task the `Tensorboard` tools added to the base library were important assets, as they allowed to identify models where the decrease in the training loss function corresponded to gains in the link prediction capabilities of the model on a validation set, as well as identifying models that tended to overfit. Given a good understanding of the

adjustable hyperparameters in relation to the performance on the dataset, a search space of hyperparameters that surrounded such values was created for each model in order to start the Bayesian optimizer process (each search process ran for 50 iterations). After the searching process finished, a set of hyperparameters was found for each model (Table 4.1), and as such the training process started. For the training of the final models, each of the models was given a thousand epochs to train with validation steps occurring in intervals of ten epochs. As mentioned before, these validation steps allow the training method to understand if performance is being lost and if so, stop the training process early, which made it so that not all the models ran for the initial defined thousand epochs. Figure 4.1 presents the validation fMRR and training loss of the final models.

	TransE	**DistMult**	**ConvE**
Batch Size	32	25	16
Embedding Size	183	169	100
Optimizer	ADAM	RMS	RMS
Learning Rate	0.005372	0.001120	0.001047
Negative Rate	1	1	0
Normalization Type	L1	–	–
Margin	6.057259	–	–
Regularization (Lambda)	–	0.000186	–
Reshape Divisor	–	–	20
Input Dropout	–	–	0.2
Feature Map Dropout	–	–	0.2
Hidden Layer Dropout	–	–	0.4
Label Smoothing	–	–	0.2

Table 4.1: Hyperparameters used in the last version of each model for the UMLS dataset

Timings

The `Tensorboard` tool introduced in the `train_model()` allowed to easily log the training time of each model. However, as mentioned before not all the models trained for the same amount of epochs due to the early stopping mechanism in place, as such on Table 4.2 are registered not only the total training times in seconds of each model but also the time per epoch ($\frac{\text{(total train)}}{\text{number of training epochs}}$). Despite providing this relative metric, the timings obtained should not be used as an accurate comparison measure between the different model variants, given that the different sets of hyperparameters used for each of them can have an impact in the training times, most notably the batch size.

Testing Metrics

The final trained models were tested against a testing set which the models had never seen before as described in the proposed methodology. Table **??** is a compendium of the

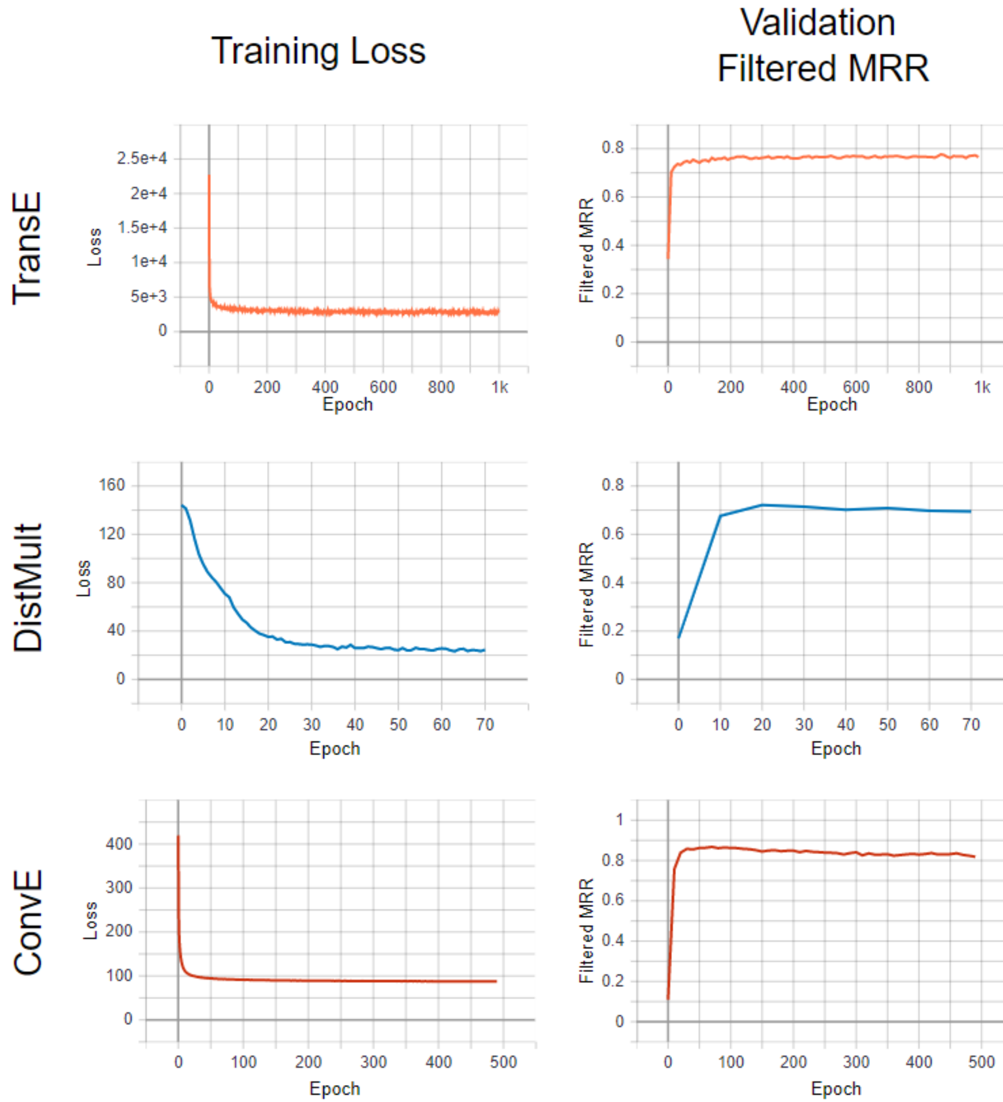


Figure 4.1: UMLS's training Loss and validation fMRR during the training process for the different models

TransE		DistMult		ConvE	
Total Train Time	Time per epoch	Total Train Time	Time per epoch	Total Train Time	Time per epoch
763	0.76	42	0.6	1412	2.88

Table 4.2: Total training time and training time per epoch of the models on the UMLS dataset in seconds

values of the metrics obtained in the testing phase. Both the raw and filtered variants of each metric are also reported, as well as the reported values of other works using their own implementations of the models on the dataset.

Model	Work	MR		MRR		Hits							
						@1		@3		@5		@10	
		Raw	**Filt.**	Raw	**Filt.**	Raw	**Filt.**	Raw	**Filt.**	Raw	**Filt.**	Raw	**Filt.**
Transe	SoA[59]	–	**1.84**	–	–	–	–	–	–	–	–	–	**0.99**
	our	14.15	**2.15**	0.18	**0.75**	0.04	**0.57**	0.16	**0.92**	0.29	**0.96**	0.56	**0.98**
DistMult	SoA[59]	–	**5.52**	–	–	–	–	–	–	–	–	–	**0.85**
	our	21.89	**6.2**	0.12	**0.65**	0.02	**0.56**	0.08	**0.7**	0.14	**0.77**	0.35	**0.84**
ConvE	SoA[26]	–	**1**	–	**0.94**	–	**0.92**	–	**0.96**	–	–	–	**0.99**
	our	19.16	**2.96**	0.15	**0.81**	0.04	**0.72**	0.12	**0.88**	0.20	**0.91**	0.38	**0.95**

Table 4.3: Testing results of the different models on the UMLS dataset

4.2.3 Discussion

An initial look at the results shows a large discrepancy between the raw and filtered versions of the metrics. This can most likely be attributed to the high connectivity of this network. With only 135 entities and 6529 relationships, the ratio between number of relations and entities is 48. Additionally, the number of occurrences of each relation type in the network is skewed, being verified a higher concentration of some specific types of relations in comparison with others (Figure 4.2). This can lead to a large number of possible correct predictions for an incomplete triple ($< h, r, ? >$ or $<?, r, t >$) where the relation type has a high number of occurrences, and as such the filtered approach which does not penalize the ranking of a prediction given other correct predictions, will show much higher results. In this work, and similarly to other works in the area, a larger importance is put into the filtered metrics.

A second observation that can be made is the slight underperformance of DistMult compared to the other two models. The main hypothesis for this occurrence stems from the scoring function this model employs. The score of each triple is given by the bilinear product between the embeddings of all elements of the triple ($h^T\ diag(r)\ t$). This operation is commutative, and as such will score triple $< h, r, t >$ the same as $< t, r, h >$ which is not always true in directed graphs.

Overall, this experiment allowed the validation of both the implementations of the pykg2vec library in use as well as the pipeline defined in order to find good models. This is corroborated by the fact that the obtained results are relatively close to ones reported on other works in the area, with ConvE being the only one that presented some underperformance relative to the state of the art on some of the metrics (Hits@1 and MRR).

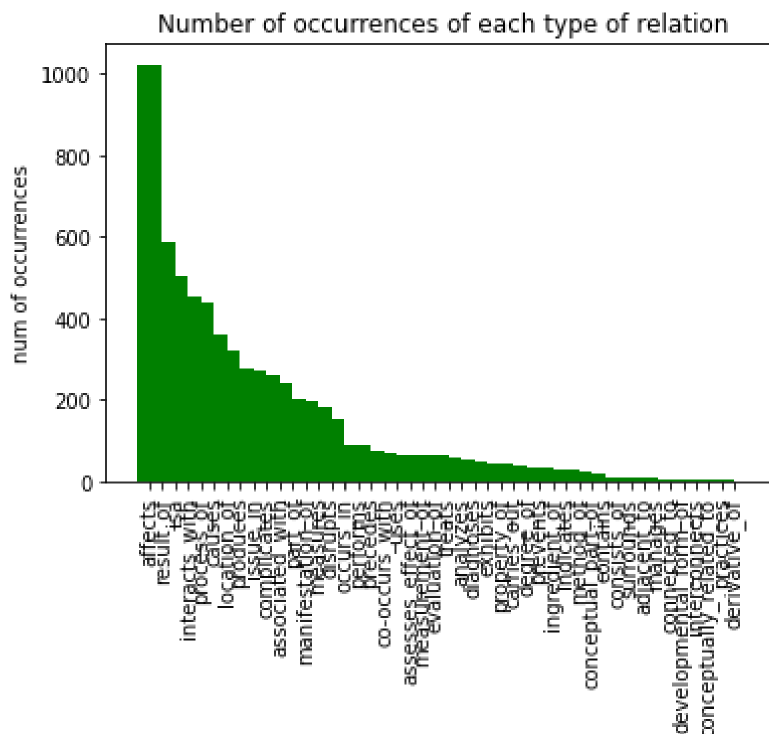


Figure 4.2: Distribution of relation types on the UMLS dataset

---

4.3 Experiment Two: Comprehensive Antibiotic Resistance Database

4.3.1 Dataset

The main objective of the second experiment was to test the use of link prediction capabilities of knowledge graph embedding models, on a general network that fell under the biomedicine scope and that was not already being used by other works in the area. The main idea behind such a decision was to test if any kind of network could benefit from these types of methods, without the need for the user to do any kind of data treatment.

The Comprehensive Antibiotic Resistance Database (CARD) is a project run by laboratories belonging to McMaster University. It integrates data from different fields such as molecular biology and biochemistry in order to provide a standardized and central database of antimicrobial resistance (AMR) sequences and mutations. To provide such standardization, all the data in the database follows a set of controlled vocabularies (ontologies), the main one being the "Antibiotic Resistance Ontology" (ARO), which was used as the data set for this experiment. This ontology comprehends entities such as antibiotic molecules, resistance genes, mechanisms of resistance, as well as entities that encode higher level concepts. These entities are connected by a set of relations that fall under one of the following eleven relation types:

- **is_a:** A hierarchical relation type, that indicates that entity A is a "subclass" of entity B;
- **part_of:** Relation type that shows composition. Indicates that entity A is a component that constitutes class B;
- **has_part:** Is the inverse of the `part_of`. It indicates that class A has a component B;
- **participates_in:** Relation that indicates that entity A is involved in process B;
- **regulates:** Relation that indicates a regulatory role that entity A has in relation to entity B;
- **derives_from:** A type of relation between class A and B where A is a derivation of B;
- **evolutionary_variant_of:** A relation where gene A is a variant of gene B;
- **confers_resistance_to_drug_class:** Indicates that the presence of entity A confers resistance to a certain drug class;
- **confers_resistance_to_antibiotic:** Indicates that the presence of entity A is associated with the conferring of resistance to a specific antibiotic;
- **targeted_by:** A relation where molecule A is the target of drug class B;
- **targeted_by_antibiotic:** A relation where molecule A is the target of a specific antibiotic B.

Additionally to having a native network structure, which makes it suitable for the context of this work, CARD's ARO also has other properties that led to it being chosen. Firstly, the fact that the scope of the dataset is an interesting target for link prediction tasks, especially in the context of drug repositioning. Secondly, the fact that at the time of writing no other work has been found that performs link prediction over the ARO ontology network, making it a good dataset to experiment the hypothesis of these methods being applicable to any sort of network in the biomedical field.

The ARO network is provided in the official website of the project[1]. The dataset is made available in numerous ontology representation formats. The "Open Biological and Biomedical Ontologies" (OBO) format was the one chosen as the starting point for the process of transforming the dataset into a format that the pykg2vec library could ingest. The first step taken was to translate this OBO format file into a graph like structure, which was done with the help of the open-source python library `obonet`[2]. With some scripting this structure was then loaded into a Neo4J database instance in order to allow the visualization of the dataset. Figure 4.3 shows an overview of the entire network.

[1]https://card.mcmaster.ca/download
[2]https://pypi.org/project/obonet/

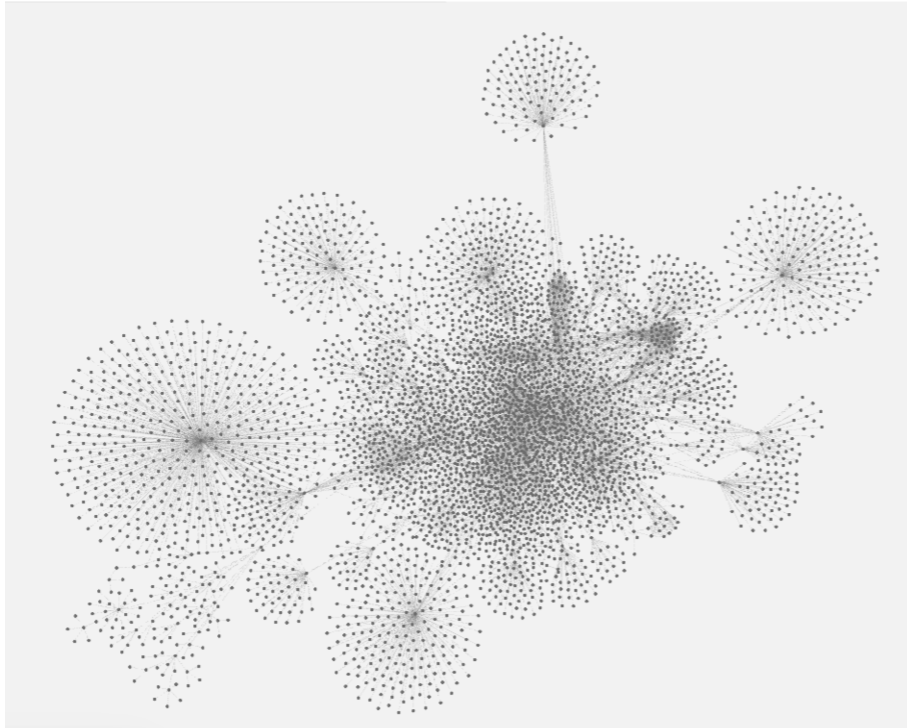


Figure 4.3: Overview of the Original CARD Network

After the data was explored, the training, validation and test files required by the pykg2vec library needed to be created. The first step was to obtain a list with all the triples using the following Cypher query:

```
MATCH (n)-[r]->(m)
RETURN n.ARO_ID as h, Type(r) as r, m.ARO_ID as t
```

Given this list it was possible to generate the aforementioned training, testing and validation sets using a simple holdout method. However, in the scope of this work a different approach was taken in the case of the splitting process in order to test a variable that could impact performance of these methods. This variable was the number of unique entities in the training set. The reason behind testing such variables stems from a problem commonly found in recommender systems literature, the "Cold start problem". In the case of recommender systems, it refers to the concern of a system not being able to draw information for entities (users or items) which it has not gathered sufficient information from. Given that the models used in this work attribute an embedding to each entity of the network, which are then refined during the training process based on the triples they appear on, these are prone to face similar problems. An extreme case scenario would be the testing and/or validation set having a large number of triples containing entities not seen in the training set. This would likely lead to underperformance comparatively to a more balanced set. In order to understand if this variable could affect the performance of

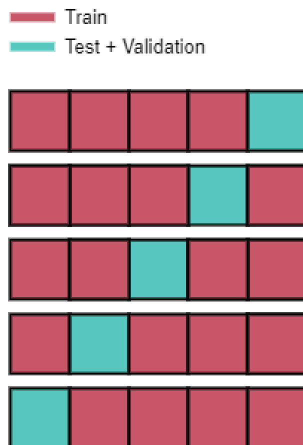


Figure 4.4: Leave-one-out 5 fold cross validation split strategy

---

the models on this particular dataset, an extra layer was added to the splitting process. Firstly the dataset was split using a holdout method (80% train, 10% validation and 10% test) in a way that maximized the number of unique entities on the triples of the training set (“ideal” split), by iteratively testing a large number of random splits and choosing the best one. Next, a set of other five splits were generated where this maximization process was not performed. The generation of these five splits followed the strategy used in “leave-one-out cross-validation”, with 80% of the total dataset for training and the other 20% to divide between validation and testing (10%+10%) as seen in Figure 4.4.

Both the “ideal split” and the five splits from cross validation ended up with the following characteristics:

- **Number of entities:** 4550
- **Number of relation types:** 11
- **Total Number of triples:** 8550
- **Number of triples on the training set:** 6840
- **Number of triples on the validation set:** 855
- **Number of triples on the test set:** 855

Given this set of splits it was possible to compare the performance of the models on the “ideal” split against the average performance on the other five “random” splits.

4.3.2 Hyperparameter search and training of the final model

Similar to what was done for the UMLS experiment, a series of manual experiments were made in order to understand the impact of different hyperparameters on the performance of the model. Afterwards, a search space was produced for each of the types of

model used. This search space was then used to conduct a search for a good performing set of hyperparameters on each of the splits generated (Table 4.4 presents the best hyperparameters for the "ideal" split and in "Appendix I" tables for each of the cross validation splits can be found). Finally, using the set of hyperparameters found for the different models on each of the splits, a final version of each model was trained. Figure 4.5 presents the evolution of the training loss and validation fMRR of each of the models on the different splits.

	TransE	**DistMult**	**ConvE**
Batch Size	23	105	16
Embedding Size	82	116	160
Optimizer	RMS	ADAM	RMS
Learning Rate	0.017058	0.002292	0.001109
Negative Rate	1	1	0
Normalization Type	L1	–	–
Margin	9.752415	–	–
Regularization (Lambda)	–	0.0006860	–
Reshape Divisor	–	–	20
Input Dropout	–	–	0.3
Feature Map Dropout	–	–	0.3
Hidden Layer Dropout	–	–	0.4
Label Smoothing	–	–	0.2

Table 4.4: Best hyperparameters found for the "ideal" split of the original CARD's ARO dataset

Timings

Regarding timing measurements, the total train time as well as the training time per epoch were registered for every model on each of the splits (Table 4.5). Additionally, the table also presents the average train time and time per epoch across the different splits.

Testing Metrics

With the final versions of the models obtained, each of them was tested against their corresponding testing set. Table 4.6 shows the results obtained for the different models applied on each of the splits.

4.3.3 Discussion

Firstly, it is important to mention that in order to try to obtain better results, the number of search iterations was increased for the TransE and DistMult models (100 instead of the 50 used on the UMLS dataset). It was not possible to do the same for the ConvE

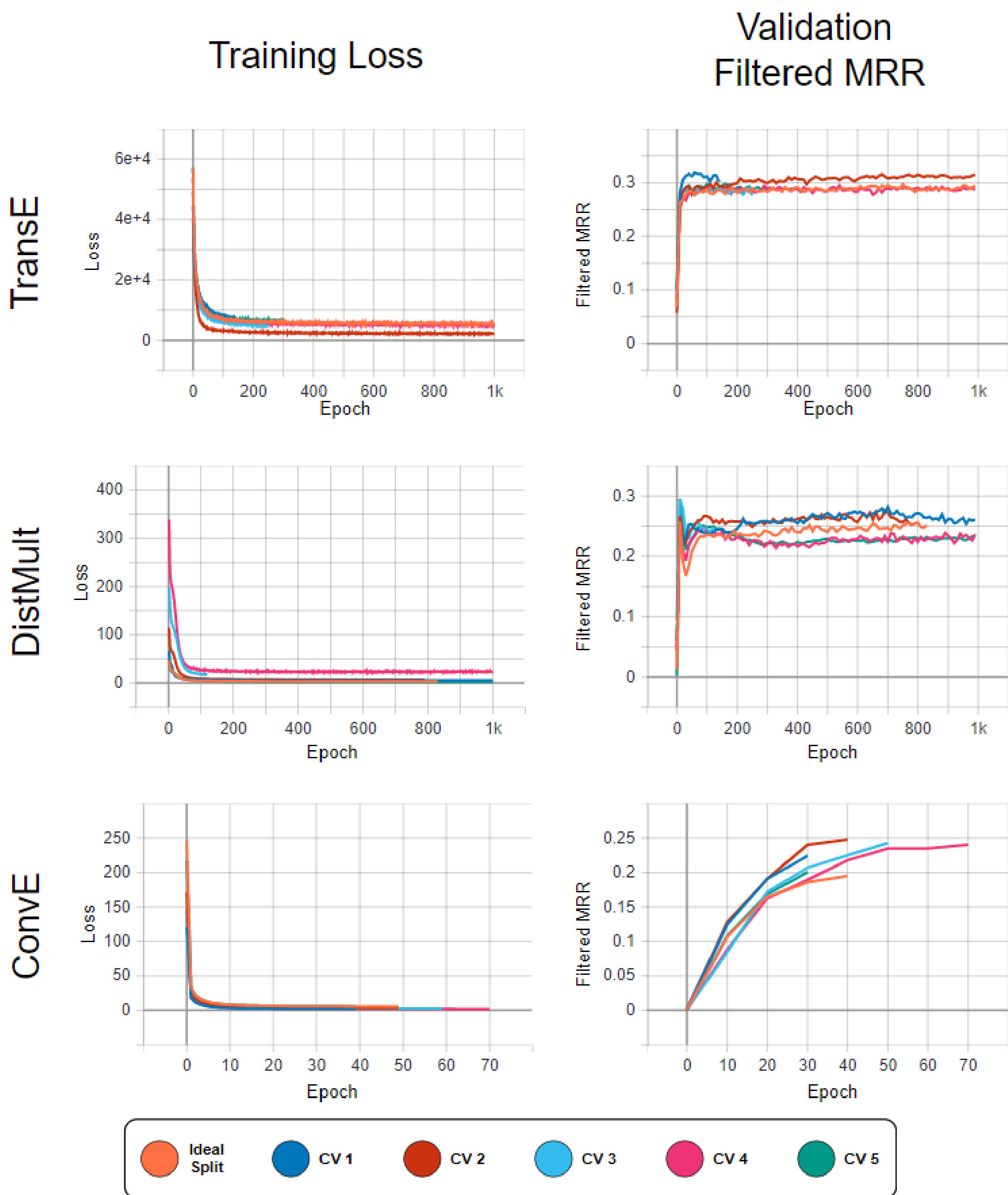


Figure 4.5: Training loss and validation fMRR of the different splits along the epochs on the original ARO dataset

	TransE		DistMult		ConvE	
	Total Train Time	Time per epoch	Total Train Time	Time per epoch	Total Train Time	Time per epoch
Ideal Split	1484	1.48	421	0.51	216	4.32
CV Split 1	150	1.07	683	0.68	105	2.63
CV Split 2	841	0.84	590	0.75	256	5.12
CV Split 3	470	1.88	136	1.13	157	2.62
CV Split 4	1846	1.85	1600	1.6	152	2.17
CV Split 5	744	2.48	615	0.615	178	4.45
Mean	**922.5**	**1.6**	**674.17**	**0.88**	**177.33**	**3.55**

Table 4.5: Total training time and train time per epoch metrics of the different models for the original ARO dataset. All the values presented are in seconds

Type	Split	MR		MRR		Hits							
						@1		@3		@5		@10	
		Raw	Filt.	Raw	Filt.	Raw	Filt.	Raw	Filt.	Raw	Filt.	Raw	Filt.
TransE	Best	518.61	**491.28**	0.14	**0.29**	0.07	**0.23**	0.15	**0.31**	0.19	**0.35**	0.28	**0.4**
	CV5	550.95	**523.02**	0.14	**0.28**	0.07	**0.22**	0.15	**0.31**	0.21	**0.35**	0.29	**0.4**
DistMult	Best	896	**867.09**	0.1	**0.25**	0.04	**0.2**	0.1	**0.26**	0.14	**0.29**	0.2	**0.33**
	CV5	971.18	**942.11**	0.1	**0.24**	0.05	**0.19**	0.11	**0.26**	0.15	**0.29**	0.22	**0.33**
ConvE	Best	1139.88	**1110.75**	0.09	**0.2**	0.04	**0.15**	0.1	**0.23**	0.14	**0.26**	0.19	**0.3**
	CV5	842	**812.24**	0.11	**0.24**	0.05	**0.18**	0.12	**0.26**	0.17	**0.3**	0.24	**0.35**

Table 4.6: Testing results of the different models on the original CARD's ARO dataset

given its higher training time, that led Google's Colab platform to terminate the search process before the search process finished. However, it is also to note that the increase in search iterations did not always find better models, and when it did, this only outperformed the ones found using 50 iterations by a small amount.

Based on the obtained results, it appears that there is no clear advantage in trying to create a split that maximizes the numbers of unique entities versus using a random split. Upon this observation a closer look was taken at the splits and it was verified that the difference in number of unique entities between the "ideal" split and the "worst" of the cross validation splits is only 93 entities ($4085 - 3992$). Despite that, it could still be argued that the five random splits were outliers and that the typical split could have an even lower number of unique entities in the training set. To verify this hypothesis, a hundred thousand splits were generated and the average number of unique entities was obtained. The results showed that on average each split has 4014 unique entities, with the lowest value obtained being 3943. With this information it is possible to confirm that the cross validation splits were not outliers and are a good representation of the average random split.

In terms of performance and comparing it to the one obtained in the UMLS dataset, a large difference can be observed. While there are multiple factors that can lead to such results, some of which are inherent to the domain the datasets represent, a few hypotheses for this underperformance can be proposed. The first hypothesis is based on the connectivity of the datasets. While the UMLS dataset is highly connected, with an average of 48 relations per entity, CARD's ARO dataset only has approximately 2 relations per entity. This sparsity makes it so that each entity embedding will not be as "refined" comparatively to more connected datasets. The second hypothesis is related to the unbalance of the dataset relations. Both UMLS and the ARO dataset are hierarchical networks and as such their entities have a large number of hierarchical relations ("isa" relations). However, this unbalance is even higher in the ARO network, with 4731 relations out of the 8550 total triples being `is_a` relations which in conjunction with the sparse nature of the network seems to lead to a bad performance.

4.4 Experiment Three: Altered Comprehensive Antibiotic Resistance Database

Given the potential problems mentioned in the previous experiment, namely the large prominence of hierarchical relations (`is_a` relations) as well as the sparsity of the dataset, a variation of the original dataset was generated in order to try and address these issues. This variation prunes the dataset from some of the hierarchical entities (entities that only are associated with others relations by `is_a` relations), while also adding some new relations derived from the already existing knowledge in the dataset, and from extra data available in the CARD's official website which is not present in the current downloadable dataset. The objective of these changes is to understand if addressing some of the hypothesized problems

with the previous dataset could translate into better link prediction performances. In principle these changes do not alter the inherent validity/truth of the dataset, since most of the changes made are deletions of existing relations and entities, and the only added new relations are extracted from official sources.

4.4.1 Dataset

The main tool used to execute such restructuring was the Neo4J Browser tool which allowed to query specific parts of the dataset in order to understand its structure, and allowed to easily transform it.

As mentioned before, ARO has a hierarchical nature and as such it has a root node ("process or component of antibiotic biology or chemistry") from which all the other nodes derive. Connected to this node there exist seven other nodes that are the roots for the seven main branches of the graph, each one containing a specific type of data under it. The seven nodes are the following:

- **antibiotic molecule**: Parent node for all the entities relative to the chemicals of the network (antibiotics);
- **resistance-modifying agents:** Parent node for adjuvants and other potentiators of antibiotic effectiveness nodes;
- **mechanism of antibiotic resistance:** Contains the different antibiotic resistance strategies;
- **determinant of antibiotic resistance:** Contains the genes, gene products and other entities that confer antibiotic resistances to certain organisms. Highly connected with the mechanism of antibiotic resistance branch;
- **antibiotic target:** Parent node for all the entities that are in someway targeted by an antibiotic or drug class;
- **antibiotic biosynthesis:** Root for the branch containing the names of different types of antibiotic biosynthesis;
- **component of AMR phenotype terminology:** Root node for a collection of terms related to the AMR field.

`Antibiotic molecule`, `Mechanism of antibiotic resistance` and `Determinant of antibiotic resistance`, are the three main branches of the dataset containing a big part of the entities in the dataset [58]. The main source of knowledge about the antibiotic resistance processes comes from the relations between them. Other branches provide additional entities that can interact with entities in these three main branches in order to provide extra context and knowledge. However, two of these other branches do not provide useful information to the context of the problem, given that their entities do not interact with entities from other branches, as it can be verified by executing the following `Cypher` query:

```
//Get all the 7 types of main "parents" as a collection
match (n)-[:is_a]->({name:"process or component of antibiotic biology or chemistry"})
with collect(n.name) as c

// From this 7 types get the ones whose children have interactions with the children
//of any other of the 6 remaining types
match (k)-[:is_a]->({name:"process or component of antibiotic biology or chemistry"})
match (y)-[*]->(k)
match (m)-[*]->(x)
match (m)-[r]-(y)
where x.name in c and x.name<>k.name
with k,count(distinct r)>0 as cnt
return k.name


//Result:
// -"antibiotic target"
// - "determinant of antibiotic resistance"
// - "antibiotic molecule"
// - "mechanism of antibiotic resistance"
// - "resistance-modifying agents"
```

As it can be seen, all the branches except `antibiotic biosynthesis` and `component of AMR phenotype terminology` have at least one relation to an entity below another branch. As such, and given the fact that those branches are mainly composed of `is_a` relations, they were prime candidates to be removed in order to reduce the concentration of those specific relations.

After removing those two branches, an effort was made in order to make sense of the remainder of the dataset, which was done by classifying the most entities in the dataset as possible. As mentioned before, the CARD's official website contains extra information not present in the original dataset, one such information being a recently added classification of some of the entities of the graph, reported in the 2020 CARD report paper [58]. Those categories are the following:

- **Drug_Class:** Entities that represent a general class of drugs, that are the parents to antibiotic entities that share similar chemical structures;
- **Antibiotic:** Antimicrobial entities;
- **Adjuvant:** Chemical entities that can be combined with antibiotics in order to enhance the effectiveness of those;
- **AMR_Gene_Family:** Parents to gene entities that share common biochemical functions;
- **Resistance_Mechanism:** Strategy used in order to obtain resistance to antibiotics;

- **Efflux_Component** and **Efflux_Regulator:** Entities involved in antibiotic efflux pump processes. Unlike the previous categories, the authors of CARD only attributed this category to the parent entities from which the real components and regulators derive from.

Beside the previous categories, one more classification was indirectly derived from the website, that being the `Gene` class. Despite the class not being directly mentioned, it was possible to identify `Gene` entities by verifying if the website made available their DNA sequence as well as their protein homolog sequence. Additionally to the classifications obtained using the website, a couple more entities were classified given their outgoing or incoming relations. The first of them was the `Target` class, that was attributed to all entities that had an outgoing relation of either `targeted_by` or `targeted_by_antibiotic`. The second one being the `Antibiotic_Mixture` class, that was given to nodes under the "antibiotic target" branch that had outgoing `has_part` relations to `Antibiotics` or `Adjuvants`. In the context of the domain, antibiotic mixtures are aggregations of different antibiotics (with the possibility of having some adjuvant) that together target some component of a bacterial agent. The `Regulator` class was attributed to any child of the `Efflux Regulator` entity that had an outgoing `regulates` relation. And finally, the classes `Efflux_Pump` and `Efflux_Pump_Subunit` were given to entities under the `Efflux Component` "sub-branch". More concretely the `Efflux_Pump` class was attributed to all entities that derived from one of the five efflux pump superfamilies [60] (Figure 4.6):

- `MFS`: Major facilitator superfamily ("ARO:0010002")
- `ATP`: ATP-binding cassette superfamily ("ARO:0010001")
- `SMR`: Small multidrug resistance superfamily ("ARO:0010003")
- `RND`: Resistance-nodulation-cell division superfamily ("ARO:0010004")
- `MATE`: Multi antimicrobial extrusion protein superfamily ("ARO:3000112")

The `Efflux_Pump_Subunit` class on the other hand was attributed to all entities that made `part_of` an `Efflux_Pump` entity.

The next step taken was the addition of the some relations that could be inferred form the official website, those being the `belongs_to_drug_class` which connected directly each `Antibiotic` entity with its respective `Drug_Class` ignoring possible hierarchical chains existing in between. Similarly the `associated_with_resistance_mechanism` was added in order to connect entities involved in conferring antibiotic resistance, and the `Resistance_Mechanism` they are involved in.

After all the classifications were attributed, it was then easier to better understand the dataset and some more pruning was made. In the antibiotic branch, every node that was not a `Drug_Class` or `Antibiotic` was removed. Under the `determinant of antibiotic resistance` and `mechanism of antibiotic resistance` branches (which for the most part share their nodes), the `Efflux Component` and `Efflux Regulator` entities where

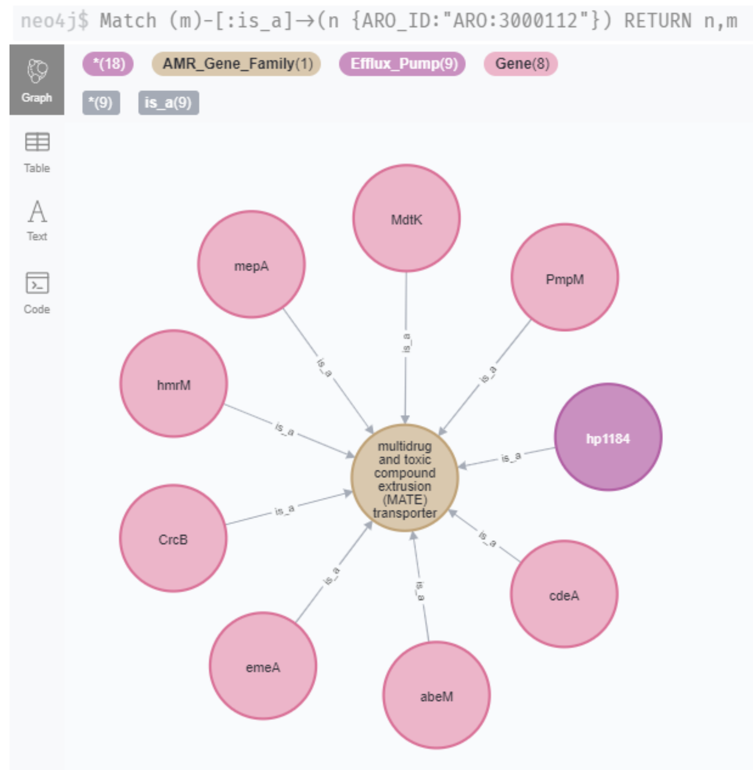


Figure 4.6: Efflux pumps from the MATE superfamily

---

deleted since they were only used as parent nodes for the actual data entities. Also under this branch every node that was hierarchically above an `AMR_Gene_Family` one, and that did not contain any incoming or outgoing relation besides hierarchical ones was removed.

In terms of relations, the `associated_with_resistance_mechanism` and `belongs_to_drug_class` were added in order to directly connect all `Antibiotic` entities with their respective `Drug_Class` and connect all entities involved in providing antibiotic resistance with their correspondent `Resistance_Mechanism` respectively, information that was taken from the website. It is to note that cases in which an `is_a` relation already existed between those nodes it was substituted by these new relations.

At last, after all the transformations, the last step was to eliminate any small "island" sub-graphs that were left. For this operation, Neo4J's Bloom tool visually separates all the sub-graphs. Given that all the observed sub-graphs were weakly connected, the `Weakly connected component` procedure provided by the `Graph Data Science Library` was used to find these nodes and remove them.

```
CALL gds.wcc.stream("myGraph")
YIELD nodeId, componentId
WITH componentId, collect(gds.util.asNode(nodeId).name) AS libraries
where size(libraries)<4
with apoc.coll.flatten(collect(libraries)) as nodeList
CALL apoc.periodic.iterate(
```

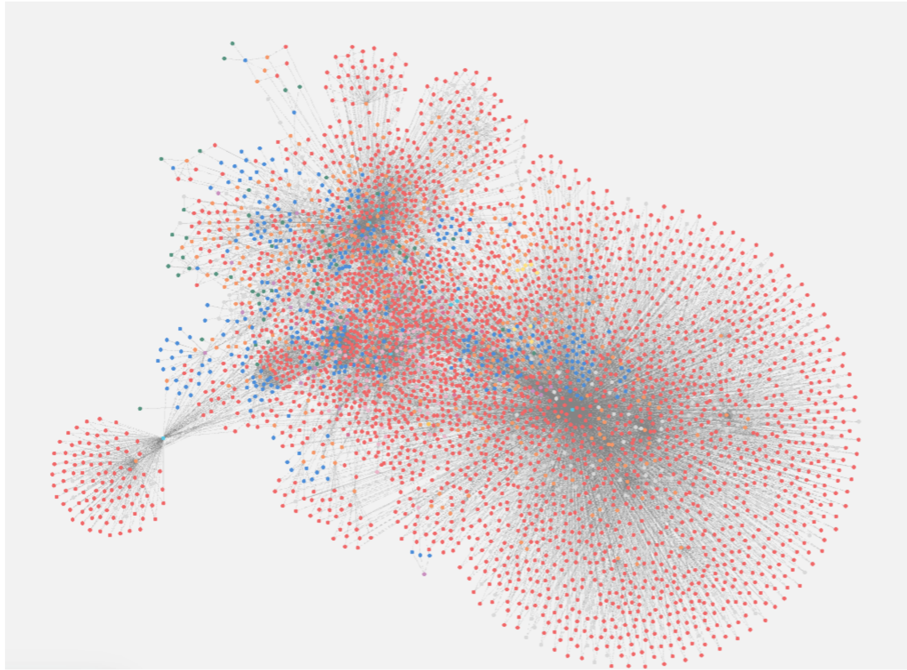


Figure 4.7: Visualization of the modified dataset

```
"Match (n) where n.name in $nodeList return n","detach delete n",
{batch_size:1000,params:{nodeList:nodeList}}) Yield batch, operations
return 1
```

The graph shown in Figure 4.7 is the end result after all the transformations. It can be observed that the majority of the entities are either Genes (Red) or Antibiotics (Blue). Additionally, looking at the count of each type of relation in the dataset, it can be verified that the number of `is_a` relations was reduced from 4731 to 3306, which attenuates the skewness mentioned before. Once again it is important to note that despite the number of transformations done, the "validity" of the data was not compromised. This is assured, given that these transformations were either the removal of entities, which can lead to the loss of some data, but does not alter the "truth" of the remainder of the dataset. While the addition of new relations was only made based on information, present on the official website.

With this final version of the dataset, this was exported and splitted in the same way as in "Experiment 2", meaning that an "ideal" split that maximized the number of unique entities in the training set, as well as five random splits obtained based on leave-one-out cross validation strategy were obtained. Each resulting split has the following characteristics:

- **Number of entities:** 4229
- **Number of relations types:** 14

- **Total Number of triples:** 11210
- **Number of triples on the training set:** 8968
- **Number of triples on the validation set:** 1121
- **Number of triples on the test set:** 1121

4.4.2 Hyperparameter search and training of the final model

The process used in order to search and train the final versions of each model was the same used in the previous experiments. That means that there was an initial search for a range of hyperparameters that seemed to perform well on the dataset. Then given that range, the search space `yaml` files were created and the searching processes started. At the end, given the best hyperparameter sets found by the Bayesian optimizer algorithm, the final versions of the models were trained. Table 4.7 contains the set of hyperparameters used to train the different models on the "ideal" split, for tables for the rest of the splits can be found on "Appendix I". Figure 4.8 presents the evolution of the metrics of the final training process of each of the models, where it can be observed that the learning process of all the models converged (training loss converges), and at the same time the validation mean reciprocal rank increased.

	TransE	**DistMult**	**ConvE**
Batch Size	21	95	55
Embedding Size	87	71	120
Optimizer	RMS	RMS	RMS
Learning Rate	0.002655	0.004570	0.001304
Negative Rate	1	1	0
Normalization Type	L1	–	–
Margin	7.446822	–	–
Regularization (Lambda)	–	0.000964	–
Reshape Divisor	–	–	20
Input Dropout	–	–	0.2
Feature Map Dropout	–	–	0.3
Hidden Layer Dropout	–	–	0.3
Label Smoothing	–	–	0.1

Table 4.7: Best hyperparameters found for the "ideal" split of the altered CARD's ARO dataset

Timings

The total training time and training time per epoch of each of the models are presented on Table 4.8.

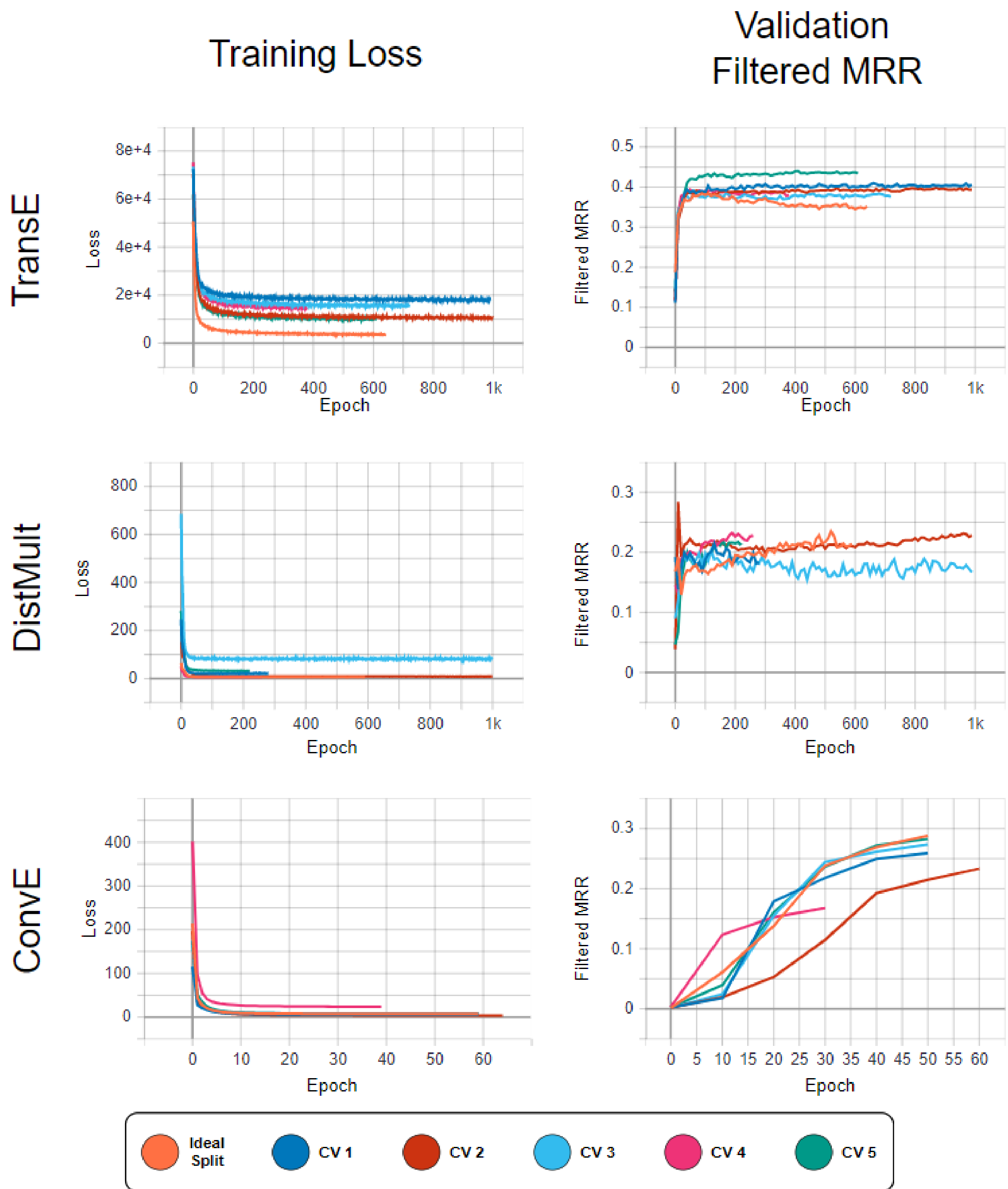


Figure 4.8: Training loss and validation fMRR of the different splits along the epochs on the altered ARO dataset

	TransE		DistMult		ConvE	
	Total Train Time	Time per epoch	Total Train Time	Time per epoch	Total Train Time	Time per epoch
Ideal Split	1362	2.13	410	0.69	506	8.43
CV Split 1	769	0.78	357	1.28	525	8.75
CV Split 2	1403	1.4	1235	1.24	549	8.45
CV Split 3	478	0.66	2884	2.88	497	8.28
CV Split 4	235	0.62	173	0.67	436	10.9
CV Split 5	481	0.79	330	1.5	494	8.23
Mean	**788**	**1.06**	**898.17**	**1.38**	**501.17**	**8.84**

Table 4.8: Total training time and train time per epoch metrics of the different models for the altered ARO dataset. All the values presented are in seconds

Type	Split	MR		MRR		Hits							
						@1		@3		@5		@10	
		Raw	Filt.	Raw	Filt.	Raw	Filt.	Raw	Filt.	Raw	Filt.	Raw	Filt.
TransE	Best	441.46	**173.56**	0.18	**0.36**	0.11	**0.29**	0.2	**0.4**	0.24	**0.46**	0.32	**0.53**
	CV5	485.83	**222.68**	0.19	**0.39**	0.12	**0.32**	0.21	**0.42**	0.25	**0.46**	0.31	**0.51**
DistMult	Best	794.95	**525.88**	0.14	**0.22**	0.09	**0.15**	0.15	**0.25**	0.19	**0.3**	0.26	**0.36**
	CV5	883.54	**597.67**	0.12	**0.2**	0.07	**0.14**	0.12	**0.22**	0.16	**0.26**	0.23	**0.31**
ConvE	Best	937.9	**667.46**	0.2	**0.29**	0.13	**0.21**	0.24	**0.34**	0.29	**0.39**	0.34	**0.44**
	CV5	955.34	**679.91**	0.18	**0.26**	0.09	**0.18**	0.23	**0.32**	0.27	**0.37**	0.33	**0.42**

Table 4.9: Testing results of the different models on the altered CARD's ARO dataset

Testing Metrics

All the models were tested against their corresponding testing sets obtaining the performance results shown in Table 4.9.

4.4.3 Discussion

Analogously to what was done on "Experiment 2", the number of search iterations given to the bayesian optimizer processes of the TransE and DistMult models was increased from fifty to a hundred in order to push the search for a set of hyperparameters that could maximize the performance of the models. Once again it was not possible to have the same increase to the ConvE model given its higher training time in conjunction with the restrictions imposed by the Colab tool used to run the experiment.

Based on the testing metrics obtained on this experiment, it can be observed that similarly to what was happened in the previous experiment, the difference in the number of unique entities on the training set between the "ideal" split and the cross validation ones did not have a significant impact. Once again, a look at the dataset shows that by running

a hundred thousand random splits the worst value for the number of unique entities on the training set found was 4054 and an average of 4100, whilst on the "ideal" split the number of unique entities on the training set is 4142, which justifies the similarity in performance between the splits. Additionally to this observation, looking at the testing results, it appears that there were some improvements in performance when compared to the original version. These results are however not a final indicative of the capabilities of these models applied to this context, since that with more domain knowledge in the area of antibiotic resistance it would be possible to further curate the graph in order to filter out parts of the graph that are superfluous, and focus more on the entities that are involved in more interesting relations to perform link prediction , such as `confers_resistance_to_antibiotic` and `confers_resistance_to_drug_class`.

Chapter 5

Conclusion

The main objective proposed by this thesis was the exploration of the use of knowledge graph embedding methods in order to derive new knowledge from pre-existing biomedical multi-relational graphs. To that end a series of KGE models were studied and an unified pipeline was used in place in order to obtain and train models capable of performing link prediction tasks over a set of biomedical datasets. Based on the results obtained on the three experiments carried out, it can be concluded that these models can achieve good results on some biomedical datasets such as the UMLS semantic network, where for the task of ranking the entities of the graph based on the likelihood of them being the correct missing part of an incomplete triple (which are obtained from the testing set triples), the model places the correct within the first ten predictions 98% of the time (Hits@10). Despite that, this level of performance was not verified across all the datasets, more concretely when using the models over the antibiotic resistance ontology (ARO) graph of the "Comprehensive antibiotic resistance" database, the models did not perform as well. However it was verified in a third experiment that by filtering down this dataset and making it more connex it was possible to increase the performance obtained. As such, taking all these experiments into account, the potential of using these methods over biomedical networks is variable, it is not a "silver-bullet" for each and every situation, but within certain datasets or in some cases, given certain adjustments to the original graph, they can be valuable assets for a multitude of problems.

In view of these conclusions, potential users of these models should take into account the use case in which they are planning to apply the model. Given the fact that the predictions made by these models are not always true, they should not be used as the determinant factors in crucial decision making software namely assisted medical decision making software. However, given a scenario where an expert in the area can review the predictions made, these models can be helpful in tasks such as the quick generation of hypotheses to be tested, for example in drug repurposing or drug adverse reaction contexts. Additionally, given the fact that only baseline models were used in the scope of the experiments made on this work, it is worth mentioning that the use of more recent models should be considered, since it is likely that they could present better performance levels. Finally users interested in using these models should also consider the use of alternative libraries,

namely OpenKE, if their use case requires the training process to be fast. Otherwise if their main goal is to explore with different and more recent models Pykg2vec is a better option.

Future research can expand on the work done on this thesis by exploring strategies that could better model the hierarchical nature of the datasets used here, and also devise or find strategies that can reduce the perceived impact of connection sparsity on the performance of these embedding methods.

Furthermore future works can also explore additional uses for the obtained embeddings, being it using them on other downstream tasks, such as clustering, or use them to refine pre-existing feature vectors obtained by some other methods such as text-mining embeddings or even manually obtained sets of features.

Additionally to the aforementioned objective, the concepts learned during the execution of this work were compiled and used to develop a python notebook that aims to serve as an introduction to the use of knowledge graph embeddings for link prediction tasks, providing a generalized training pipeline that could be used by researchers or other users interested in applying these models to their own datasets (`https://colab.research.google.com/drive/10CS1n9_wLz0z75jvRDiK1Z3syvWq5met?usp=sharing`). This notebook follows the following structure:

- Installing the pykg2vec library;
- Training a model with a predetermined set of hyperparameters;
- Hyperparameter search;
- Using custom datasets;
- Using trained model to make inferences.

Bibliography

[1] Leonardo F.R. Ribeiro, Pedro H.P. Saverese, and Daniel R. Figueiredo. Struc2vec: Learning node representations from structural identity. *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, Part F1296:385–394, 2017.

[2] Michel Dumontier, Alison Callahan, Jose Cruz-Toledo, Peter Ansell, Vincent Emonet, François Belleau, and Arnaud Droit. Bio2RDF release 3: A larger connected network of linked data for the life sciences. Technical report, 2014.

[3] Nicole Redaschi and UniProt Consortium. UniProt in RDF: Tackling Data Integration and Distributed Annotation with the Semantic Web. *Nature Precedings*, pages 1–1, apr 2009.

[4] S. T. Roweis and L. K. Saul. Nonlinear dimensionality reduction by locally linear embedding. *Science*, 290(5500):2323–2326, dec 2000.

[5] Yoshua Bengio, Réjean Ducharme, and Pascal Vincent. A neural probabilistic language model. In *Advances in Neural Information Processing Systems*, volume 3, pages 1137–1155, 2001.

[6] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. In *1st International Conference on Learning Representations, ICLR 2013 - Workshop Track Proceedings*, 2013.

[7] Albert László Barabási. *Network science*, volume 371. 2013.

[8] Palash Goyal and Emilio Ferrara. Graph embedding techniques, applications, and performance: A survey. Technical report, 2018.

[9] Mikhail Belkin and Partha Niyogi. Laplacian eigenmaps for dimensionality reduction and data representation. Technical Report 6, 2003.

[10] Shaosheng Cao, Wei Lu, and Qiongkai Xu. GraRep: Learning graph representations with global structural information. *International Conference on Information and Knowledge Management, Proceedings*, 19-23-Oct-:891–900, 2015.

[11] Mingdong Ou, Peng Cui, Jian Pei, Ziwei Zhang, and Wenwu Zhu. Asymmetric transitivity preserving graph embedding. In *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, volume 13-17-Augu, pages 1105–1114, 2016.

[12] Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. DeepWalk: Online learning of social representations. *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 701–710, 2014.

[13] Aditya Grover and Jure Leskovec. Node2vec: Scalable feature learning for networks. *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 13-17-Augu:855–864, 2016.

[14] Daixin Wang, Peng Cui, and Wenwu Zhu. Structural deep network embedding. *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 13-17-Augu:1225–1234, 2016.

[15] Brasil Capítulo, Daniel N R da Silva, and Artur Ziviani Fabio Porto. Aprendizado de máquina e inferência em Grafos de Conhecimento. 2019.

[16] Andrea Rossi, Antonio Matinata, Paolo Merialdo, Denilson Barbosa, and Donatella Firmani. Knowledge Graph Embedding for Link Prediction: A Comparative Analysis. 2016.

[17] Bishan Yang, Wen tau Yih, Xiaodong He, Jianfeng Gao, and Li Deng. Embedding entities and relations for learning and inference in knowledge bases. Technical report, 2015.

[18] Theo Trouillon, Johannes Welbl, Sebastian Riedel, Eric Ciaussier, and Guillaume Bouchard. Complex embeddings for simple link prediction. In *33rd International Conference on Machine Learning, ICML 2016*, volume 5, pages 3021–3032, 2016.

[19] Théo Trouillon and Maximilian Nickel. Complex and holographic embeddings of knowledge graphs: A comparison, 2017.

[20] Hanxiao Liu, Yuexin Wu, and Yiming Yang. Analogical inference for multi-relational embeddings. In *34th International Conference on Machine Learning, ICML 2017*, volume 5, pages 3422–3432, 2017.

[21] Seyed Mehran Kazemi and David Poole. Simple embedding for link prediction in knowledge graphs. In *Advances in Neural Information Processing Systems*, volume 2018-Decem, pages 4284–4295, 2018.

[22] Antoine Bordes, Nicolas Usunier, Alberto Garcia-Durán, Jason Weston, and Oksana Yakhnenko. Translating embeddings for modeling multi-relational data, 2013.

[23] Zhen Wang, Jianwen Zhang, Jianlin Feng, and Zheng Chen. Knowledge graph embedding by translating on hyperplanes. Technical report, 2014.

[24] Hailun Lin, Yong Liu, Weiping Wang, Yinliang Yue, and Zheng Lin. Learning Entity and Relation Embeddings for Knowledge Resolution. Technical report, 2017.

[25] Zhiqing Sun, Zhi Hong Deng, Jian Yun Nie, and Jian Tang. Rotate: Knowledge graph embedding by relational rotation in complex space, 2019.

[26] Tim Dettmers, Pasquale Minervini, Pontus Stenetorp, and Sebastian Riedel. Convolutional 2D knowledge graph embeddings. Technical report, 2018.

[27] Dai Quoc Nguyen, Tu Dinh Nguyen, Dat Quoc Nguyen, and Dinh Phung. A novel embedding model for knowledge base completion based on convolutional neural network. *NAACL HLT 2018 - 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies - Proceedings of the Conference*, 2:327–333, 2018.

[28] Xiaotian Jiang, Quan Wang, and Bin Wang. Adaptive convolution for multi-relational learning. In *NAACL HLT 2019 - 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies - Proceedings of the Conference*, volume 1, pages 978–987, 2019.

[29] Efpia. The Pharmaceutical Industry in Figures: Key Data 2014. *European Federation of Pharmaceutical Industries and Associations (EFPIA) Brussels Office*, page 28, 2014.

[30] Ted T. Ashburn and Karl B. Thor. Drug repositioning: Identifying and developing new uses for existing drugs. *Nature Reviews Drug Discovery*, 3(8):673–683, 2004.

[31] Chang Su, Jie Tong, Yongjun Zhu, Peng Cui, and Fei Wang. Network embedding in biomedical data science. *Briefings in Bioinformatics*, 21(1):182–197, dec 2018.

[32] Daniel N Sosa, Alexander Derry, Margaret Guo, Eric Wei, Connor Brinton, and Russ B Altman. A Literature-Based Knowledge Graph Embedding Method for Identifying Drug Repurposing Opportunities in Rare Diseases, 2019.

[33] Daniel M. Bean, Honghan Wu, Olubanke Dzahini, Matthew Broadbent, Robert Stewart, and Richard J.B. Dobson. Knowledge graph prediction of unknown adverse drug reactions and validation in electronic health records. *Scientific Reports*, 7(1):1–11, dec 2017.

[34] Barry Causier. Studying the interactome with the yeast two-hybrid system and mass spectrometry. *Mass Spectrometry Reviews*, 23(5):350–367, sep 2004.

[35] Brad Melis. Electronic health records, 2011.

[36] Matthew Wai Heng Chung, Jianyu Liu, and Hegler Tissot. Clinical knowledge graph embedding representation bridging the gap between electronic health records and prediction models. In *Proceedings - 18th IEEE International Conference on Machine Learning and Applications, ICMLA 2019*, pages 1448–1453. Institute of Electrical and Electronics Engineers Inc., dec 2019.

[37] Nansu Zong, Hyeoneui Kim, Victoria Ngo, and Olivier Harismendy. Deep mining heterogeneous networks of biomedical linked data to predict novel drug-target associations. *Bioinformatics*, 33(15):2337–2344, aug 2017.

[38] Pengwei Wang, Tianyong Hao, Jun Yan, and Lianwen Jin. Large-scale extraction of drug–disease pairs from the medical literature. *Journal of the Association for Information Science and Technology*, 68(11):2649–2661, nov 2017.

[39] Guanghui Li, Jiawei Luo, Qiu Xiao, Cheng Liang, Pingjian Ding, and Buwen Cao. Predicting MicroRNA-Disease Associations Using Network Topological Similarity Based on DeepWalk. *IEEE Access*, 5:24032–24039, oct 2017.

[40] Sezin Kircali Ata, Le Ou-Yang, Yuan Fang, Chee Keong Kwoh, Min Wu, and Xiao Li Li. Integrating node embeddings and biological annotations for genes to predict disease-gene associations. *BMC Systems Biology*, 12(9):31–44, dec 2018.

[41] Bo Ya Ji, Zhu Hong You, Li Cheng, Ji Ren Zhou, Daniyal Alghazzawi, and Li Ping Li. Predicting miRNA-disease association from heterogeneous information network with GraRep embedding model. *Scientific Reports*, 10(1):1–12, dec 2020.

[42] Wonjun Choi and Hyunju Lee. Inference of Biomedical Relations among Chemicals, Genes, Diseases, and Symptoms Using Knowledge Representation Learning. *IEEE Access*, 7:179373–179384, 2019.

[43] Qin Dai, Naoya Inoue, Paul Reisert, Ryo Takahashi, and Kentaro Inui. Distantly Supervised Biomedical Knowledge Acquisition via Knowledge Graph Based Attention. Technical report, 2019.

[44] Sameh K. Mohamed, Aayah Nounu, and Vít Nováček. Drug target discovery using knowledge graph embeddings. *Proceedings of the ACM Symposium on Applied Computing*, Part F1477:11–18, 2019.

[45] Soumya Sharma, T. Y.S.S. Santosh, Bishal Santra, Niloy Ganguly, Abhik Jana, and Pawan Goyal. Incorporating domain knowledge into medical NLI using knowledge graphs. In *EMNLP-IJCNLP 2019 - 2019 Conference on Empirical Methods in Natural Language Processing and 9th International Joint Conference on Natural Language Processing, Proceedings of the Conference*, pages 6092–6097, 2020.

[46] Byoung-Ha Yoon, Seon-Kyu Kim, and Seon-Young Kim. Use of Graph Database for the Integration of Heterogeneous Biological Data. *Genomics & Informatics*, 15(1):19, 2017.

[47] Bryce Merkl Sasaki. Graph Databases for Beginners: Other Graph Technologies, 2018.

[48] Joe Celko. *Graph Databases.* 2014.

[49] Luca Costabello, Sumit Pai, Chan Le Van, Rory McGrath, Nick McCarthy, and Pedro Tabacof. AmpliGraph: a Library for Representation Learning on Knowledge Graphs, March 2019.

[50] Xu Han, Shulin Cao, Xin Lv, Yankai Lin, Zhiyuan Liu, Maosong Sun, and Juanzi Li. OpenKE: An open toolkit for knowledge embedding. In *EMNLP 2018 - Conference on Empirical Methods in Natural Language Processing: System Demonstrations, Proceedings*, pages 139–144, 2018.

[51] Shih Yuan Yu, Sujit Rokka Chhetri, Arquimedes Canedo, Palash Goyal, and Mohammad Abdullah Al Faruque. Pykg2vec: A python library for knowledge graph embedding, 2019.

[52] James Bergstra and Yoshua Bengio. Random search for hyper-parameter optimization. Technical report, 2012.

[53] Lisha Li, Kevin Jamieson, Giulia DeSalvo, Afshin Rostamizadeh, and Ameet Talwalkar. Hyperband: A novel bandit-based approach to hyperparameter optimization. Technical report, 2018.

[54] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. Technical report, 2014.

[55] Ian Dewancker, Michael McCourt, and Scott Clark. Bayesian Optimization Primer. Technical report, 2015.

[56] J. Bergstra, D. Yamins, and D. D. Cox. Making a science of model search: Hyperparameter optimization in hundreds of dimensions for vision architectures. In *30th International Conference on Machine Learning, ICML 2013*, volume 28, pages 115–123, 2013.

[57] Olivier Bodenreider. The Unified Medical Language System (UMLS): Integrating biomedical terminology. *Nucleic Acids Research*, 32(DATABASE ISS.):D267, jan 2004.

[58] Brian P. Alcock, Amogelang R. Raphenya, Tammy T.Y. Lau, Kara K. Tsang, Mégane Bouchard, Arman Edalatmand, William Huynh, Anna Lisa V. Nguyen, Annie A. Cheng, Sihan Liu, Sally Y. Min, Anatoly Miroshnichenko, Hiu Ki Tran, Rafik E. Werfalli, Jalees A. Nasir, Martins Oloni, David J. Speicher, Alexandra Florescu, Bhavya Singh, Mateusz Faltyn, Anastasia Hernandez-Koutoucheva, Arjun N. Sharma, Emily Bordeleau, Andrew C. Pawlowski, Haley L. Zubyk, Damion Dooley, Emma Griffiths, Finlay Maguire, Geoff L. Winsor, Robert G. Beiko, Fiona S.L. Brinkman,

William W.L. Hsiao, Gary V. Domselaar, and Andrew G. McArthur. CARD 2020: Antibiotic resistome surveillance with the comprehensive antibiotic resistance database. *Nucleic Acids Research*, 48(D1):D517–D525, jan 2020.

[59] Liang Yao, Chengsheng Mao, and Yuan Luo. KG-BERT: Bert for knowledge graph completion. Technical report, 2019.

[60] Jared A. Delmar, Chih Chia Su, and Edward W. Yu. Bacterial multidrug efflux transporters. *Annual Review of Biophysics*, 43(1):93–117, 2014.

Appendix I

	TransE	**DistMult**	**ConvE**
Batch Size	35	72	36
Embedding Size	54	326	180
Optimizer	RMS	ADAM	RMS
Learning Rate	0.001904	0.002469	0.002017
Negative Rate	1	1	0
Normalization Type	L1	–	–
Margin	8.450403	–	–
Regularization (Lambda)	–	0.000788	–
Reshape Divisor	–	–	20
Input Dropout	–	–	0.2
Feature Map Dropout	–	–	0.3
Hidden Layer Dropout	–	–	0.4
Label Smoothing	–	–	0.1

Table 5.1: Best hyperparameters found for the 1st Cross Validation split of the original CARD's ARO dataset

	TransE	**DistMult**	**ConvE**
Batch Size	48	42	26
Embedding Size	113	100	140
Optimizer	RMS	ADAM	RMS
Learning Rate	0.034886	0.001664	0.00141
Negative Rate	1	1	0
Normalization Type	L1	–	–
Margin	9.3190745	–	–
Regularization (Lambda)	–	0.00537	–
Reshape Divisor	–	–	20
Input Dropout	–	–	0.3
Feature Map Dropout	–	–	0.3
Hidden Layer Dropout	–	–	0.4
Label Smoothing	–	–	0.1

Table 5.2: Best hyperparameters found for the 2nd Cross Validation split of the original CARD's ARO dataset

	TransE	**DistMult**	**ConvE**
Batch Size	17	24	32
Embedding Size	40	297	140
Optimizer	RMS	RMS	RMS
Learning Rate	0.00273	0.001041	0.001584
Negative Rate	1	1	0
Normalization Type	L1	–	–
Margin	6.957936	–	–
Regularization (Lambda)	–	0.000512	–
Reshape Divisor	–	–	20
Input Dropout	–	–	0.2
Feature Map Dropout	–	–	0.2
Hidden Layer Dropout	–	–	0.4
Label Smoothing	–	–	0.2

Table 5.3: Best hyperparameters found for the 3rd Cross Validation split of the original CARD's ARO dataset

	TransE	**DistMult**	**ConvE**
Batch Size	17	14	42
Embedding Size	47	132	180
Optimizer	RMS	ADAM	RMS
Learning Rate	0.003096	0.001016	0.001673
Negative Rate	1	1	0
Normalization Type	L1	–	–
Margin	7.593912	–	–
Regularization (Lambda)	–	0.000545	–
Reshape Divisor	–	–	20
Input Dropout	–	–	0.2
Feature Map Dropout	–	–	0.2
Hidden Layer Dropout	–	–	0.4
Label Smoothing	–	–	0.2

Table 5.4: Best hyperparameters found for the 4th Cross Validation split of the original CARD's ARO dataset

	TransE	**DistMult**	**ConvE**
Batch Size	12	123	16
Embedding Size	87	486	120
Optimizer	RMS	ADAM	RMS
Learning Rate	0.012973	0.001226	0.001211
Negative Rate	1	1	0
Normalization Type	L1	–	–
Margin	9.897107	–	–
Regularization (Lambda)	–	0.000533	–
Reshape Divisor	–	–	20
Input Dropout	–	–	0.2
Feature Map Dropout	–	–	0.2
Hidden Layer Dropout	–	–	0.4
Label Smoothing	–	–	0.2

Table 5.5: Best hyperparameters found for the 5th Cross Validation split of the original CARD's ARO dataset

	TransE	**DistMult**	**ConvE**
Batch Size	88	25	112
Embedding Size	49	102	120
Optimizer	ADAM	RMS	RMS
Learning Rate	0.001169	0.004380	0.002142
Negative Rate	1	1	0
Normalization Type	L1	–	–
Margin	9.106215	–	–
Regularization (Lambda)	–	0.000663	–
Reshape Divisor	–	–	20
Input Dropout	–	–	0.2
Feature Map Dropout	–	–	0.3
Hidden Layer Dropout	–	–	0.3
Label Smoothing	–	–	0.2

Table 5.6: Best hyperparameters found for the 1st Cross Validation split of the altered CARD's ARO dataset

	TransE	**DistMult**	**ConvE**
Batch Size	30	41	109
Embedding Size	75	409	200
Optimizer	SGD	ADAM	RMS
Learning Rate	0.0493355	0.001142	0.002659
Negative Rate	1	1	0
Normalization Type	L1	–	–
Margin	9.190434	–	–
Regularization (Lambda)	–	0.000600	–
Reshape Divisor	–	–	20
Input Dropout	–	–	0.2
Feature Map Dropout	–	–	0.2
Hidden Layer Dropout	–	–	0.3
Label Smoothing	–	–	0.1

Table 5.7: Best hyperparameters found for the 2nd Cross Validation split of the altered CARD's ARO dataset

	TransE	**DistMult**	**ConvE**
Batch Size	130	9	76
Embedding Size	69	70	160
Optimizer	RMS	ADAM	RMS
Learning Rate	0.004700	0.005523	0.001670
Negative Rate	1	1	0
Normalization Type	L1	–	–
Margin	9.857968	–	–
Regularization (Lambda)	–	0.001317	–
Reshape Divisor	–	–	20
Input Dropout	–	–	0.3
Feature Map Dropout	–	–	0.3
Hidden Layer Dropout	–	–	0.3
Label Smoothing	–	–	0.2

Table 5.8: Best hyperparameters found for the 3rd Cross Validation split of the altered CARD's ARO dataset

	TransE	**DistMult**	**ConvE**
Batch Size	173	120	18
Embedding Size	73	81	180
Optimizer	SGD	rms	RMS
Learning Rate	0.0314065	0.010802	0.001009
Negative Rate	1	1	0
Normalization Type	L1	–	–
Margin	9.878712	–	–
Regularization (Lambda)	–	0.001541	–
Reshape Divisor	–	–	20
Input Dropout	–	–	0.3
Feature Map Dropout	–	–	0.2
Hidden Layer Dropout	–	–	0.4
Label Smoothing	–	–	0.1

Table 5.9: Best hyperparameters found for the 4th Cross Validation split of the altered CARD's ARO dataset

	TransE	**DistMult**	**ConvE**
Batch Size	89	22	59
Embedding Size	63	76	100
Optimizer	RMS	ADAM	RMS
Learning Rate	0.001138	0.005391	0.001544
Negative Rate	1	1	0
Normalization Type	L1	–	–
Margin	8.365610	–	–
Regularization (Lambda)	–	0.001305	–
Reshape Divisor	–	–	20
Input Dropout	–	–	0.2
Feature Map Dropout	–	–	0.2
Hidden Layer Dropout	–	–	0.4
Label Smoothing	–	–	0.1

Table 5.10: Best hyperparameters found for the 5th Cross Validation split of the altered CARD's ARO dataset