POLITECNICO DI TORINO

Corso di Laurea Magistrale
in Ingegneria Matematica

Tesi di Laurea Magistrale

Towards Named Entity Disambiguation with Knowledge Graph Embeddings

Supervisors
Prof. Antonio Vetrò
Dr. Giuseppe Futia
Dott. Giovanni Garifo

Candidate
Felice Paolo Colliani

Anno Accademico 2022-2023

Ai miei genitori

Summary

Context: In recent years, the field of biomedicine has been experiencing a huge growth of interest, particularly in the application of knowledge-mining algorithms. Extracting knowledge from the scientific literature is valuable for assisting professionals in making well-informed decisions supported by relevant documents. This thesis discusses a novel approach for the Named Entity Disambiguation (NED) task, applied to the biomedical field. The proposed approach combines pre-trained language models and graph technologies for the NED task. It is worth noting that this methodology is not limited to the biomedical field, but it could be applied to various domains. However, the biomedical domain is employed as a case study, since it is one of the most complex due to the vast number of entities and a lack of sufficient clarity in the available literature.
State of the art: When dealing with a complex domain, such as the biomedical field, only relying on entity recognition is not sufficient. The same entities (e.g. diabetes) can refer to different concepts (e.g. type 2 diabetes). Assigning a unique identity to entities mentioned in text is referred to as NED. Previous NED frameworks mainly relied on the contextual information surrounding the entity to disambiguate, taking inspiration from human reasoning and attempting to find patterns between word embeddings. New approaches based on the integration of Knowledge Bases (KBs) allowed to add context to the word embeddings, increasing the accuracy of the disambiguation task. However, KBs often lack completeness, leading to unreliable results.
Method and contribution: This thesis will deal with annotated text from biomedical papers and proposes a novel approach that leverages a Siamese Neural Network (SNN) and integrates Knowledge Graph (KG) embeddings. For these reasons, the input of this NN consists of the concatenation of two ingredients: (i) the text embedding of an annotated sentence produced by a pre-trained language model in the medical domain; (ii) the KG embedding computed by graph learning methods on SNOMED, a well-known medical KG. The output is a score between 0 and 1, representing the probability that the SNOMED entity corresponds exactly to the medical entity annotated in the sentence. This novel approach delves beyond the mere context of individual words in a sentence, and it also exploits the topology of the KG. Furthermore, thanks to the Neo4j full-text search, an easier candidate selection can be performed to provide during the testing phase the best entities to the trained NN.
Results: This novel method has only recently been tested and has rarely been used in a complex domain such as biomedicine. However, it demonstrates acceptable accuracy on two famous challenging datasets of annotated documents, MedMentions and BC5CDR. Moreover, when considering a more relaxed constraint on accuracy it reaches an even higher score. Notably, this performance is also comparable with the ScispaCy models, which have been trained on a dataset containing a number of entities 70 times higher than the one used in this study.
Conclusions and Future work: The solution proposed, based on integrating KG embeddings and text embeddings, shows promising results comparable to the state of the art. However, comparable solutions achieved such results being trained on much bigger datasets. A future fully-integrated pipeline developed in this thesis could lead to better results and make it possible to apply this approach to other domains.

Acknowledgements

Starting with the people who assisted me technically throughout this thesis and this phase of my life, I would like to express my gratitude to Giuseppe Futia for introducing the main idea behind this work and providing me with valuable knowledge. Additionally, I appreciate the help of Giovanni Garifo, for consistently offering the needed engineering support. Thanks for all the Friday meeting done together.

I extend my thanks to the supervisor of this work, Prof. Antonio Vetrò, for his helpful comments and revisions.

I am also grateful to my parents for providing me with the opportunity to complete my studies without additional worries, asking me only to concentrate on pursuing this master's degree. Making my grandparents proud of this achievement brings me immense joy, as they were perhaps anticipating it even more than I was.

Special thanks to all my friends, especially those from the university group, who played a crucial role in helping me reach this milestone. I want to acknowledge Michele, the first person I met at Politecnico, and the Pinelle and Uscenti groups who supported me in maintaining a sense of calm during lighter moments.

Finally, I want to express my sincere affection to Héloïse, whom I am grateful to have met in my life and who helped me to go through during the final rush period of this thesis and the beginning of the working career.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation

The biomedical domain has seen a steadily increasing publication rate over the years due to the growth of scientific research, advances in technology, and the global emphasis on healthcare and medical research. Recent statistics about the literature show that the number of published scientific papers has climbed by 8–9% each year over the past several decades. If we only consider the biomedical field, more than 1 million papers are uploaded into the PubMed database yearly, which means about two papers per minute. Extracting knowledge from this enormous textual data is fundamental for research and healthcare. Applying Natural Language Processing (NLP) techniques in the biomedical domain represents a shift in analysing and interpreting the vast corpus of biomedical knowledge, enhancing our ability to derive meaningful insights from textual data. NLP empowers us to make informed decisions, uncover hidden connections, and drive innovation to pursue improved healthcare and scientific understanding.

In this context, it is not only necessary to recognise an entity in itself, such as genes, proteins, diseases, drugs, and clinical terms, but it is even more important to disambiguate the proper technical name of the entity, to understand better which is the proper subject of a sentence. Thanks to Name Entity Disambiguation (NED), we can disambiguate and link textual mentions of biomedical entities to their corresponding entries in biomedical knowledge bases. NED is a task in NLP that aims to assign the right and unique identity to entities in a mentioned text. It is crucial to accurately understand and extract knowledge from unstructured text, since these entities often have multiple possible interpretations or refer to distinct entities with similar names, introducing ambiguity and complexity into text analysis.

This task is important in biomedical text mining, as it enables researchers and healthcare professionals to extract valuable insights from vast amounts of biomedical literature and clinical records since it not only disambiguates the intended entity but also enriches the understanding of the text by associating it with structured external information, such as Knowledge Bases (KB).

The thesis discusses a novel approach for NED, with the help of Knowledge Graph (KG). The idea of integration with a KG is pretty recent, and it is difficult to find something at the state of art in the biomedical domain. Still, huge progress is in act in graph data science, thanks to the possibility of storing relevant information in a more structured way.

1.2 Why NED in the biomedical topic

In the biomedical domain, the challenges in NED are amplified due to the vast and continually expanding knowledge bases, complex terminologies, and the evolving nature of the field. In particular this is due to several reasons:

- the medical knowledge bases are usually coarse-grained or incomplete. In this thesis one of the most updated and used in many experiments, called UMLS, which stands for United

Medical Language System, will be used and it will be integrated with a support KG to retrieve the best entities and overcome to possible loss of information;

- the literature is full of entities with multiple meanings, and usually people who can understand the subject are not so clear on what they are writing about. So, for example, when in a paper it is written *p53*, this could refer to the gene or the protein that the gene produces. The complexity here is that only people with specialized knowledge can anticipate what they are reading;
- on the other hand, the opposite problem is when two different words have the same meaning and they express the same entity. Again with the gene *p53*, it is also known as *tp53*. It is a small change of only one letter, but for someone who doesn't know the subject perfectly and especially for a computer that is only reading text, it is impossible to know whether the two words correspond to the same entity or not. This is due to a lack of standardisation between scientists. Only large knowledge bases such as UMLS are trying to overcome this problem;
- it is really common to use acronyms when writing a scientific paper, and in the biomedical field they can relate to really different subjects. For example, *ACF* is sometimes used for Ambulatory Care Facilities, which refers to medical services, or Asymmetric Crying Facies, which is a neonate disease;
- it could be complicated to train a model due to rare entities that are not common to the basic scientific knowledge because the research on the field is not properly developed as in other sectors. MedMentions, which is one of the largest available biomedical NED annotated datasets, does not have a proportioned instance type. For example, there are less than thirty occurrences of entities with type "Drug Delivery Device". Instead, the most common type is "Disease or Syndrome", with more than 10,000 observed entities. It is for this reason that a model may have difficulties learning the right patterns for disambiguating entities.

In this context, distinguishing or disambiguating the correct entity is therefore challenging when not being an expert. Hence, a machine requires a robust knowledge base, which is why the primary concept of this work involves leveraging a knowledge graph.

1.3 Why Knowledge Graph in NED

To enable the machine to learn, in problems such as NED or name entity recognition (NER), it is really important to use a knowledge base (KB) to map the mentions found in the text. In the biomedical field, it is a critical preprocessing step to collect all the entities and properly store them with the most relevant information. In this case the KB chosen is a knowledge graph (KG), which we will see later, is a kind of database that can store information and more importantly relationships between the entities.

For example the following sentence, taken from the first abstract of MedMentions, can show how the pipeline of the disambiguation should work:

The purpose of this study was to investigate the role of DCTN4 missense variants on Pa infection incidence

To disambiguate the acronym *Pa*, which stands for *bacteria Pseudomonas aeruginosa*, the model in this work will use two key features from the KG:

- the graph embeddings will be used to train the model to understand the spatial position of the entities and how they are linked with the text embedding;
- the relationships, stored in the KG, that the Pa has with the gene *DCTN4* and the link with the word context given from *infection*.

Throughout this study we will focus on how the KG is built to support the NN within the binary classification task.

Furthermore, this model idea can also be applied outside the biomedical entity disambiguation problem. The same approach could be used in other fields to facilitate applications such as literature search, clinical decision-making, and relational knowledge discovery.

1.4 Related work

Finding similar work in biomedical literature for graph embedding in the NED task proves challenging, as this area hasn't been extensively discussed.

Indeed, the primary purpose of employing graph embedding has been to address the challenge of sparse data, which annihilates the efficient computation and management of information in extensive KG systems. These embeddings serve to translate entities and relationships within a KG into a low, dense, and continuous feature space, empowering the resulting model with capabilities for inferring and combining knowledge. Over recent years, considerable research focus has been directed towards this method.

Recent studies in utilizing graph embeddings have primarily focused on introducing advanced models to enhance original methods by capturing extra semantic insights via embeddings. In their work, Dai et al. [2020] categorize these methods into two groups: those relying on textual descriptions and those utilizing relation paths. The text-based model extends the traditional triplet-based approach, integrating additional text details to enhance its performance. Conversely, the other category involves methods that gather supplementary information to refine embedding models via multi-step relational paths, revealing one or more semantic relationships between entities.

Instead, this thesis proposes a novel method that it is not upgrading an old one, but want to compare the result obtained in past research that are not relying on graph embeddings. The main goal is to find a connection between the graph embeddings and the text embedding, trying to capture the relationship between these two features.

In fact, the work of Varma et al. [2021] proposes the NED tasks, also known as entity linking problem, with the use of only text embedding. Past approaches were limited by the poor quality of the resources in biomedical knowledge bases, since they were coarse-grained without reporting all the necessary information. They upgraded the medical domain KB used, integrating structural knowledge from a general text knowledge base, such as Wikipedia. Thanks to that, they were able to obtain a huge training set to overcome the problem of uncommon entities with low coverage.

Also the idea of Bhowmik et al. [2021] uses embedding tokens from text, but, like in this work, tries to detach from the semantic type of the entity, such as protein or gene, since this information is not always available on mentioned text. This is a more challenging task since less information will be passed to the model for understanding the difference between entities, but is a more concrete task based on actual information.

The entity linking problem is vastly studied in different applications also outside the scientific. In fact, the paper of Parravicini et al. [2019], started to investigate the graph embedding in the usage of NED, obtaining good results on benchmark datasets but not linked to the biomedical domain.

The model introduced in this thesis will be compared with a proposed model that is fully based on text embedding taken from the BERT model and with SciSpacy, which is a specialized library built on the spaCy framework, designed specifically for NLP tasks within the domain of scientific or biomedical text. Regarding entity linking, SciSpacy excels in linking or associating entity mentions in text in scientific literature with a knowledge base. It can identify entities like genes, proteins, diseases, and chemicals in text and link these entities to specific entries in databases. This linking process helps in disambiguating entities, ensuring the correct references and enabling further analysis or information retrieval.

SciSpacy's models and tools are fine-tuned to recognize and link entities within the scientific

domain, making it particularly useful for applications that require accurate identification and disambiguation of entities in biomedical or scientific texts.

1.5 Goals and contribution

As explained in the latter section, most of the available NED methods rely on textual information. They try to retrieve the information to disambiguate the identity using the context of the word in the document. Instead, this work tries to explore the usage of the knowledge graph embeddings and combine them with the text embeddings to achieve a high standard on the NED task. The goal of this approach is to utilise knowledge graph embedding to model entities and their connections within a structured knowledge base. By embedding entities and their relationships into continuous vector spaces, we can capture the semantic information associated with each entity, which can be highly valuable for disambiguation and go even more beyond the concept of the word trying to understand the geometry of the nodes in the graph.

In this framework, entity mentions within a given text are first embedded and then linked to potential entities in a knowledge graph. This results in a set of candidate entities for each mention. The candidates will be then evaluated to try to find the correct disambiguation. The key innovation of this approach lies in the representation of these candidates in a shared knowledge graph embedding space. By aligning the knowledge graph embedding with the context-based embeddings, we can compute the compatibility between the mentioned and candidate entities in a unified vector space.

In the biomedical domain it is really hard to obtain high accuracy on test sets but in this work I managed to achieve similar results compared to professional and advanced NLP libraries that have been out in the market for several years and are trained with huge machines. In fact, this framework can see substantial improvement in the next works since it is composed of different building blocks that can be easily replicated and each of them can be upgraded independently.

Chapter 2

Background

2.1 Knowledge Graph

A knowledge graph (KG) $\mathcal{G} = \{E, R, F\}$ is a collection of entities E, relations R, and facts F, which are a set of triples $(h, r, t) \in F$. The triple is the relationship $r \in R$ that connects the head $h \in E$ and the tail $t \in E$.

Basically a KG has the same physical structure of a graph, so with edges and nodes, but it can store much more information. Usually the link between two nodes in a graph can only be numeric. An example of numerical edges is shown on the left in figure 2.1. In that graph the number could correspond to the amount of litres of water that can flow from a site to another one. Instead, in a KG like the one on the right in figure 2.1, the edge is usually a verb that expresses a relationship between two entity nodes. This allows a more semantically rich representation of data compared to traditional relational databases, which usually use tables. An example of a relationship in the biomedical context could be the relationship "IT IS TREATED" between the nodes "Diabetes type 2" and "Insulin".

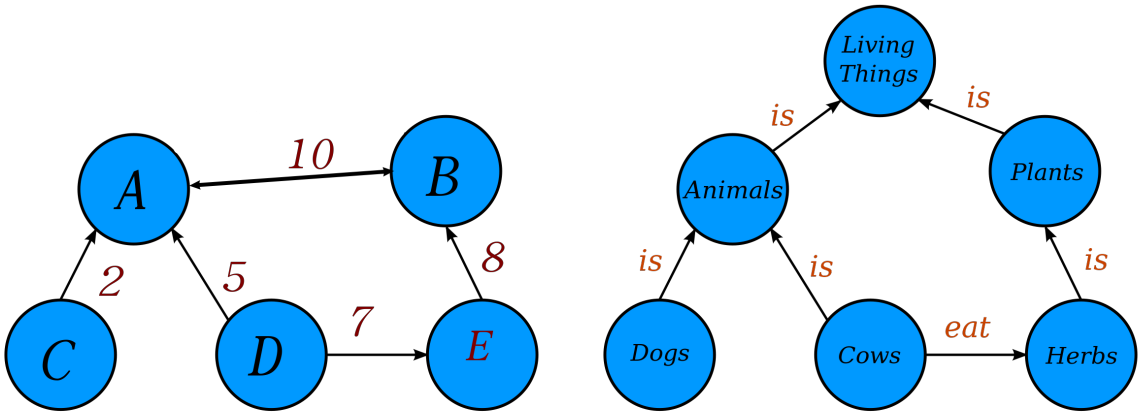

Figure 2.1. Conceptual difference between a normal graph and a Knowledge graph

In data science KG are really used and one of the most famous users is the Google search engine [1]. Google use a KG to have better results in a search and so the following example shows how to retrieve more pertinent information.

If we are looking on the web for *taj mahal*, the engine won't look for the words, but at the meaning behind them. This is because there are several concepts behind this search: it might be

[1] https://blog.google/products/search/introducing-knowledge-graph-things-not/

one of the world's most beautiful monuments, or a Grammy Award-winning musician, or possibly even a casino in Atlantic City. Or even depending on when you last ate, and where the nearest Indian restaurant is. Google KG can instantly get information that's relevant to your query, thanks also to public KB such as Freebase, Wikipedia and the CIA World Factbook. In 2012 the KG contained more than 500 million objects, as well as more than 3.5 billion facts/relationships between these different objects.

The KG enhances Google Search in three main ways:

1. the first step is finding the right thing. In the query before, did we mean Taj Mahal, the monument, or Taj Mahal, the musician? Now Google understands the different entities, and can narrow down your search results just to the one expected result as shown in figure 2.2;

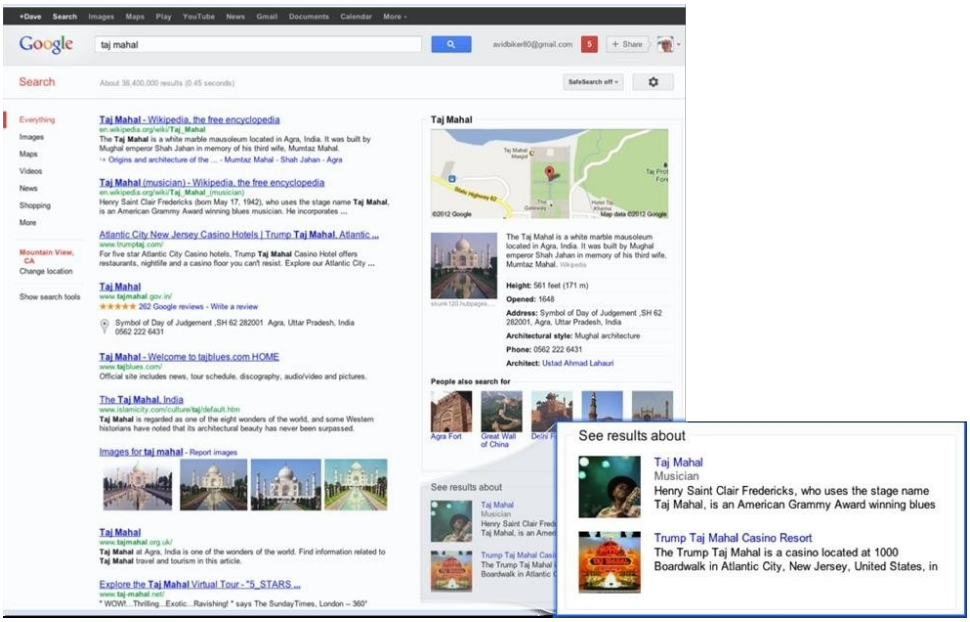

Figure 2.2. Most pertinent result for Taj Mahal in a Google search [1]

2. then it returns a complete summary with the relevant content, based on what the users looked for in the past for that specific query and the relationship between other similar interrogation such as in figure 2.3;

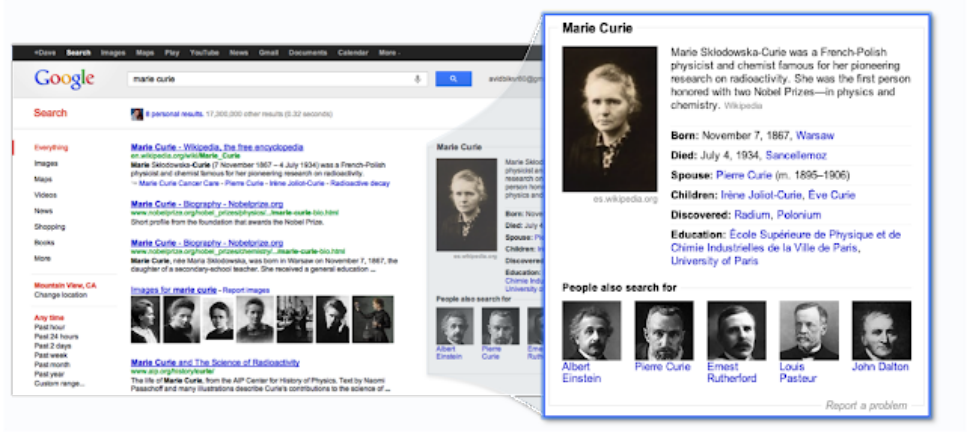

Figure 2.3. Summary and correlated search for Marie Curie [1]

3. finally, it discovers unexpected connections and looks for different facts, based on the relationship found in the KG. New suggestions from the first query can be proposed to find new interesting topics as in figure 2.4.

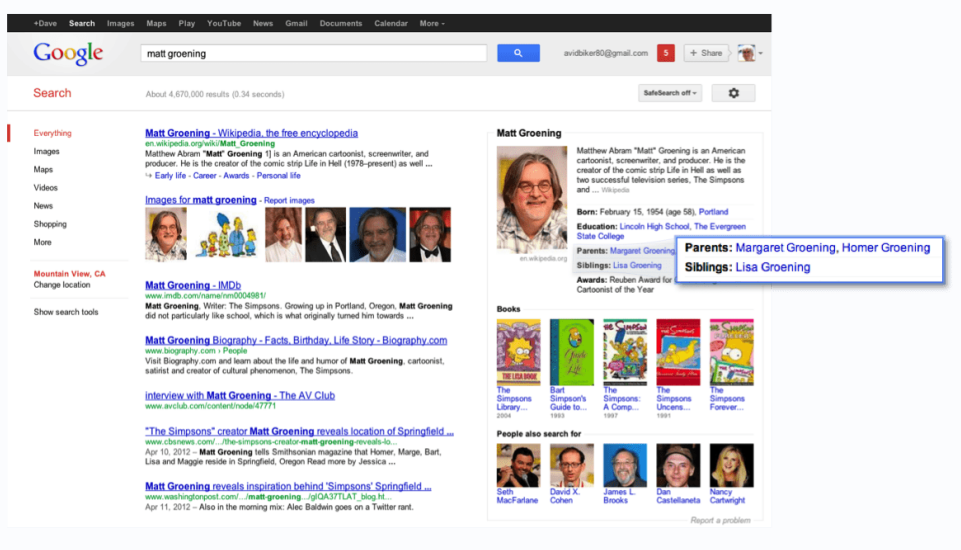

Figure 2.4. Highlight of relationships for Matt Groening search [1]

Coming back to the biomedical field, there are also other reasons why it is best to use a KG:

- for each node you can save *properties*. They are additional information useful, when retrieving a result from a query. For example, they can be different names that refer to the same node entity.
- from a KG you can create an embedding that preserves the shape of the graph, henceforth their location. It will be explained in more detail, later.
- KGs are also really flexible and can be enriched with new entities and relationships easily without needing to rebuild the whole database. For biomedical purposes, it could be really useful as soon as new entities are gradually discovered or if new connections between nodes are needed.

Database Management Systems are used to store KGs in the form of triplets, thus they are also called Triplestores. In this work, as it will be shown in later chapters, I have used the Neo4j, which is a labeled property graph (LPG). Neo4j is a really famous manager of knowledge graph which is used to implement the candidate selection, connecting the graph database with the Python code.

2.2 Embedding graph

Machine learning algorithms usually take vectors as input, so for the purpose of this work, a knowledge graph has to be transformed into a format that a neural network can read.

That is why this section will introduce knowledge graph embedding (KGE). This is an approach that transforms the representation nodes-entities and the edge-relationship into a vector space whilst maximising the preserved properties such as the graph structure and information. The goal of this embedding is to use them for prediction tasks such as entity classification, whether it be in this work or in link prediction and in recommender systems. The entities and the relationships can be translated in the same or in two vector spaces but with different dimensions.
Before starting the building up procedure of a KGE, it is needed to choose four aspects:

- a representation space, the lower-dimensional space where entities and relations will be represented;
- a scoring function, that quantifies how well a triple is represented within the embeddings;
- an encoding model, that will describe how embedded entities and relations interact with one another;
- any supplementary data from the knowledge graph that can enhance the embedded representation.

There are many different models to embed a KG, such as translational distance models, semantic matching models and deep learning models. Here are the details for each one:

- Translational distance models utilize a scoring function to assess the credibility of a triple by evaluating distances within a vector space, often following a translation operation. The one most mentioned distance model is TransE [Bordes et al. [2013]]. The relations and the entities are in the same space R^m. A triple (h, r, t) is translated in vectors $(\vec{v_h}, \vec{v_r}, \vec{v_t})$ where the head h and the tail t can be connected with the relationship r with a low error. The goal is to minimize the function

$$f_{TransE}(h, r, t) = ||\vec{v_h} + \vec{v_r} - \vec{v_t}||_{L_{1,2}}$$

  Other models introduced are TransH [Wang et al. [2014]] and TransR [Lin et al. [2015]]. The idea is to project the relations on a different surface: for TransH the projection is on a specific hyperplane, instead for TransH which is on a different vector space that can also have different dimensions.

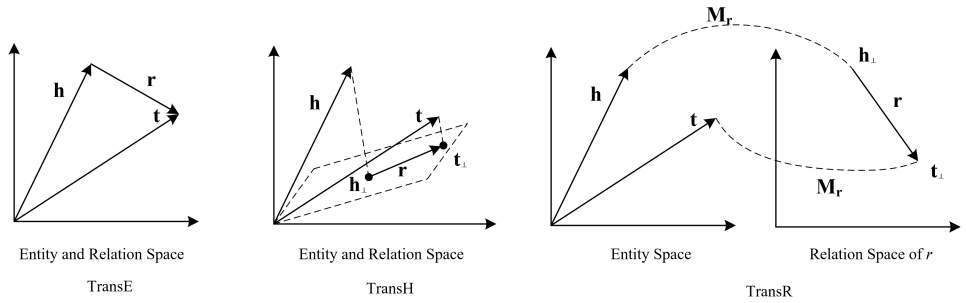

Figure 2.5. Easy geometrical explanation of the translational models

- Semantic matching models rely on a similarity-based scoring function to assess the truthfulness of a triple by aligning the semantics of latent representations of entities and relations. The first model introduced was RESCAL [Nickel et al. [2011]], which associates each entity with a vector to capture its latent semantics. Relations are represented as matrices that model pairwise interactions among latent factors. The score for a triple (h, r, t) is determined using a bilinear scoring function, which is minimised using tensor factorization. Other models have also been introduced such as DistMult [Yang et al. [2015]] which simplifies RESCAL by representing relations with diagonal matrices, thus reducing its complexity. Unlike the previous examples, ComplEX [Trouillon et al. [2016]], which extends DistMult, uses complex numbers instead of real numbers.
- Deep learning techniques employ deep neural networks to extract patterns and information from the input data of a KG. These models can distinguish the entity types and relations and the underlying structured data. In doing so, they overcome the distance-based and semantic-matching-based models, allowing a comprehensive representation of all KG features. These

kinds of approaches for KGE exhibit strong predictive capabilities, but it's worth noting that they tend to be more resource-intensive during training, requiring substantial amounts of data and also relying on pre-trained embeddings obtained from a different embedding model.

Even though this kind of model can differ a lot from one another, the goal and the main idea of the approach can be synthesised as follows.

Initially, the embedding vectors for entities and relations are set to random values. After the initial state, the algorithm iteratively optimises these embeddings from a training set until a stopping condition is met, often based on avoiding overfitting, because usually the relationships are drastically less numerous than the entities in a KG. In each iteration, a batch of training data of a fixed size is sampled, and for each triple within the batch, a random corrupted fact is sampled. This corruption replaces either the head, tail, or both entities in the triple to render the fact false. Both the original and corrupted triples are included in the training batch, and the embeddings are updated by optimising a scoring function. By the end of this process, the learned embeddings should have captured the semantic meaning of the triples and should be capable of accurately predicting unseen true and false facts within the KG.

In this thesis, graph embeddings will be retrieved from an algorithm called Node2Vec [Grover and Leskovec [2016]]. This algorithm, which is part of the semantic matching models, operates by employing a flexible notion of network exploration inspired by random walks. Starting from an initial target node and then navigating through the graph, Node2Vec strategically balances between breadth-first and depth-first search strategies. This enables it to capture both local and global neighborhood information for each node in the network. In fact, in the case of this algorithm, each node is treated like a word and the random walk is considered like a sentence. Node2Vec's key innovation lies in its ability to transform the topology of the graph into low-dimensional, continuous vector representations. By doing so, it ensures that nodes with similar network roles or connectivity patterns are positioned closer together in this embedding space.

2.3 Embedding text

For the goal of the thesis it is important to use a pre-trained language model. The concept of pre-training is closely connected to the idea of transfer learning. Transfer learning requires knowledge acquired from one or more tasks to apply it in new tasks. Traditional transfer learning relies on annotated data for supervised training, which has been the standard practice for many years. Nowadays, thanks to deep learning, pre-training with self-supervised learning on extensive unannotated data has emerged as the predominant approach to transfer learning. The key distinction lies in pre-training methods utilising unannotated data for self-supervised training and subsequently adapting to various downstream tasks through fine-tuning.

Language modelling is fundamental to start to create a pre-trained model. It aims to predict the next token based on a history of unannotated texts. The initial breakthrough in neural language modelling involved modelling n-gram probabilities using distributed word representations and feed-forward neural networks. Subsequently, deep learning techniques have gained prominence in the training of language models. Early neural language modelling used approaches predominantly relied on recurrent neural networks (RNNs), with long short-term memory (LSTM) networks.

The introduction of the transformer model sparked considerable efforts to construct more robust and efficient language models based on the transformer architecture. Some really well-known models are the trendy GPT (Generative Pretrained Transformer) [Brown et al. [2020]] and the Google model BERT (Bidirectional Encoder Representations from Transformers) [Devlin et al. [2019]], which gives a rough idea of how it works in figure 2.6. These models have demonstrated state-of-the-art performance on a wide range of NLP benchmarks and applications. They are really used nowadays in all kinds of commercial AI that replies to your answers. We can say that they are the new update of the old chat-bots.

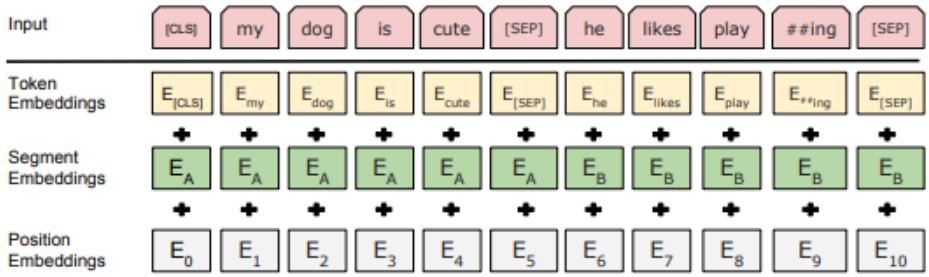

Figure 2.6. BERT input embeddings are the sum of the token embeddings, the segmentation embeddings and the position embeddings.

In the context of neural language modelling, distributed word representations, often referred as "word embeddings," play a crucial role. These representations, learned by models like Word2Vec [Mikolov et al. [2013]] and GloVe [Pennington et al. [2014]], are commonly used as initialisation for the word vectors of deep learning models. In more technical speaking subjects, they are improving many natural language processing tasks and applications, as they can leverage the vast amount of textual data on the internet. Such tasks are natural language inference and paraphrasing, which are more linked to the analysis and construction of a sentence. In this work instead, it is more needed a token-level tasks such as named entity recognition and question answering, where models are required to produce fine-grained output at the token level.

A big difference from the past models, is the abandonment of static word embeddings which may fall short in capturing diverse word meanings in context. To address this limitation, the new context-aware language models were introduced to incorporate complete contextual information into the training process. Thanks to this kind of model now it is possible to have great performance improvements in sentiment analysis, text classification, and object classification tasks Wang et al. [2023]. In the context of this work I needed an embedding for the entity that has been annotated from biomedical scientific papers. As said before, the embedding usually is composed of tokens for a single word. This means that usually long words are split into smaller "syllables" to compose the word. For example BERT can compose any word from the English dictionary using a 30,000 tokens vocabulary taken from WordPiece [Song et al. [2021]] .

The library that has been used is called Flair. It is a very simple framework for state-of-the-art NLP developed by Akbik et al. [2019]. The strength is in the possibility of having embedding of full word instead of only token embedding. This array will be then concatenated to the graph node embedding of the related entity to train the model and then test the result.

The embedding calculated by Flair are derived from a pre-trained language model. The model choose for this work is SapBERT [Liu et al. [2021]], which is a specific language model trained on biomedical entities that will permit to have a proper representation of the embeddings compared to model that are trained on a general knowledge base, as it will be shown in section 3.2.

Chapter 3

Model

In this chapter it will be presented the training model proposed for the NED task and the main idea of the new approach of the thesis in section 3.4.

The first section will describe the document dataset used for training the model. After it will be discussed the method choose to embed the annotated entities in the documents. Then the KG will be built with the help of a famous database management for graphs. Finally, the novelty of the thesis will be described and also how it affected the training of the neural network. At the end of the chapter there will be a first sight of the testing phase, presenting the candidate selection.

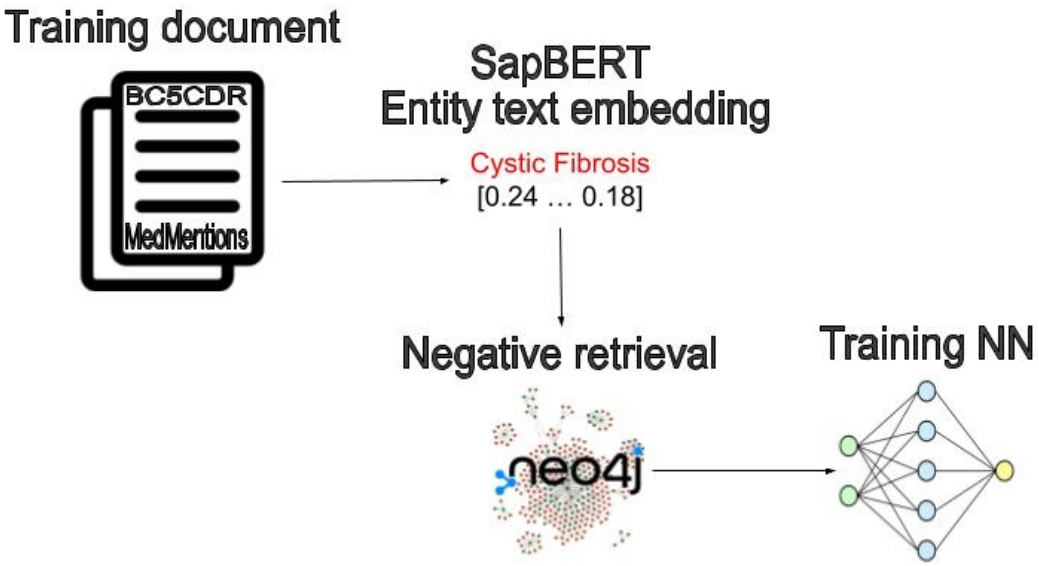

Figure 3.1. General pipeline for the model training

3.1 Dataset

For the purpose of this work different kinds of dataset were used. At the beginning of this section UMLS will be presented. It is the library that I took mostly for reference because it is the biggest biomedical storage that I used as a common base for all the other datasets I worked with. Then for the model training I mainly used the entity in SNOMED CT, which is, in a few words, a smaller portion of UMLS. The benchmark of the model presented in this thesis will be compared

with other NED models in chapter 5 and the datasets that were used to train and test the model, MedMentions and BC5CDR, are selected annotated text papers taken from PubMed®.

3.1.1 UMLS

One of the biggest libraries about biomedical and health related concepts is the Unified Medical Language System (UMLS) developed from the National Library of Medicine (NLM), part of the National Institutes of Health (NIH) in the United States. It is a comprehensive and extensive knowledge representation system and its purpose is to facilitate the development of computer systems in NLP tasks. For this reason NLM produces three databases in the UMLS Knowledge Sources: the Metathesaurus, the Semantic Network and the SPECIALIST Lexicon.

In this work, I used the Metathesaurus database. It is a very large, multi-purpose, and multilingual vocabulary database that contains information about biomedical and health-related concepts, their various names, and the relationships among them. The Metathesaurus is organised by concept or meaning. In essence, it links alternative names and views of the same concept and identifies useful relationships between different concepts. The Metathesaurus can be used for different reasons in computer programs but in this work I will focus on a few of them such as:

- synonymous biomedical terminology mapping
- natural language processing and automated indexing research
- linking between different clinical or biomedical vocabularies

The file with all the information and the entire concept structure is in a Rich Release Format, called MRCONSO1.RRF and MRCONSO2.RRF. An example result given from the search of *cystic fibrosis* term in the file, gives back different entities that contain the searched string.

```
fpcnexa@pop-os: $  grep "cystic fibrosis" MRCONSO1.RRF
C0010674|ENG|P|L0010674|VO|S0417308|N|A0477115|||AOD|DE|0000005782|cystic fibrosis
    |0|N|256|
C0055725|ENG|S|L0118189|PF|S0147652|Y|A0176219||M0050013|C007065|MSH|CE|C007065|cystic
    fibrosis  ciliary inhibitor|0|N|256|
C0016034|ENG|S|L0287645|PF|S1627441|Y|A25734023|||10016621|MDR|LLT|10006222|Breast
    cystic fibrosis |3|N|256|
```

As we can see there are many codes for each line, and multiple lines for the entity can be shown, but I omitted them from the box to be more easy to read. The code that I will work with is mainly the first one that starts with the letter *C*. It is the code from the UMLS entities that are in total more than 3 million. Since they were too many for the computational power in my possession, I decided to work with the entities in the SNOMED CT database.

3.1.2 SNOMED CT

SNOMED Clinical Terms (SNOMED CT) is an organised, machine-readable collection of medical terminology that includes codes, terms, synonyms, and explanations employed in clinical documentation and reporting. It is recognized as the most extensive and multilingual clinical healthcare terminology. SNOMED CT's primary objective is to encode the meaning employed in health-related information and to facilitate the efficient recording of clinical data to enhance patient care. Its extensive coverage contains various aspects of healthcare, including symptoms, diagnoses, procedures, body structures, organisms, substances, pharmaceuticals and devices.

This medical collection is managed and distributed by SNOMED International, a global non-profit organisation dedicated to developing healthcare standards. In July 2003, the NLM forged

an agreement with the College of American Pathologists to offer SNOMED CT to U.S. users without charge via the NLM's UMLS Metathesaurus.

The database plays a pivotal role in building a uniform method for indexing, storing, retrieving, and aggregating clinical data across various medical specialties and care settings. Thanks to the relationship available in the database, SNOMED CT it is also really good to build a KG, which is essential for the success of the work.

A good example taken from the official website of SNOMED international is the following shown in figure 3.2. The central entity in this case is the **viral pneumonia**. It is linked through a set of "*is a*" relationships that represent a poly-hierarchy of sub-types. As we can easily understand from the graph, viral pneumonia "*is a*" infective pneumonia. From there infective pneumonia "*is a*" infection, and similarly infective pneumonia "*is a*" respiratory disease. Another possibility in SNOMED CT is to link concepts to the part of the body, or a finding site. In this example, the viral pneumonia *finding site* is the lung. Finally SNOMED CT links concepts to a causative agent. So in the graph, the viral pneumonia *causative agent* is a virus.

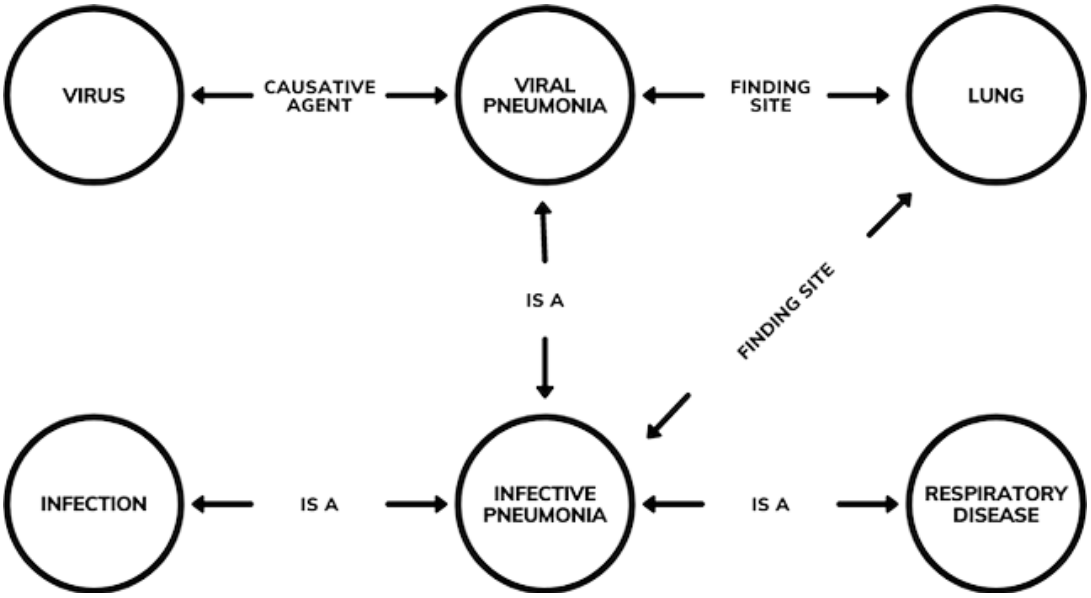

Figure 3.2. Sketch of viral pneumonia KG

So in the March 2023 edition of the SNOMED CT International database there were almost five hundred thousand entities that were also mapped in UMLS.

I also used the Medical Concept Embeddings for SNOMED-CT, but the last version found was of January 2019, produced from Pattisapu et al. [2020]. So combining the two databases in possession, in the end I was working with a total of 375180 entities.

3.1.3 PubMed® dataset

After collecting all the entities that will be used in the NED task, I needed to work with annotated PubMed® files. PubMed® is a free search engine accessing primarily the MEDLINE database of references and abstracts on life sciences and biomedical topics. It is part of the NLM. Two libraries that are challenging in the biomedical domain are MedMentions [Mohan and Li [2019]] and the BioCreative V Chemical Disease Relation (BC5CDR) [Li et al. [2016]].

MedMentions

MedMentions is a large-scale biomedical corpus with annotated UMLS concepts. There are a total of 4392 English language abstracts. The dataset has 352496 mentions, and each mention

is associated with a single UMLS Concept Unique Identifier (CUI). The concepts belong to 128 different semantic types. MedMentions also provides a 60% – 20% -20% random partitioning of the corpus into training, development, and test sets. For the purpose of this work I worked with a partition of 60% training and 40% test set. Considering the smaller database in my training set there are a total of 10895 different entities and in the test set there are 8579 distinct entities. Only half of the entities in the training set are also present in the test set, so that there are more than 40% of unknown entities in the test set.

BC5CDR

The BC5CDR corpus consists of 1500 English language articles with 4409 annotated chemicals, 5818 diseases and 3116 chemical-disease interactions, but they won't be used in this work. The articles equally partitioned into training, development, test set but I decided to have a unique test set joining the last two sets. Also in this case, selecting only the entity present in SNOMED CT the total number of unique entities dropped to 1037 for the train set and 1434 for the test set. In this case almost 50% of the entities in the test set are unknown.

A summary of all the numbers of annotated mentions taken into consideration are shown in a clear way in table 3.1

Dataset	Trainset	Testset	Common entities	Mentions trainset	Mentions testset
MedMentions	10895	8579	5038	116345	78079
BC5CDR	1037	1434	754	6593	13631

Table 3.1. Number of entities and mentions in the annotated documents

The annotated data in both datasets are published in a format called PubTator. Each paper or document it is divided with a final blank line and it is represented in the following way :

```
PMID | t | Title text
PMID | a | Abstract text
PMID TAB StartIndex TAB EndIndex TAB MentionTextSegment TAB SemanticTypeID
TAB EntityID
⋮
```

The first two lines show the title and the abstract of the paper. The following lines present the mentions, one per line. For each mentions there is a StartIndex and an EndIndex. The MentionTextSegment is the mention as shown in the document, between those index positions. The EntityID is the UMLS entity id, and the SemanticTypeID is the id of the type linked to in UMLS. If the UMLS entity is linked to different semantic types, then there's a comma-separated list of all these type IDs in this field.
An example of the first document in MedMentions is shown:

```
25763772| t | DCTN4 as a modifier of chronic Pseudomonas aeruginosa infection in cystic
fibrosis
25763772| a | Pseudomonas aeruginosa (Pa) infection in cystic fibrosis (CF) patients is ...
25763772   0   5   DCTN4   T116,T123   C4308010
25763772   23   63   chronic Pseudomonas aeruginosa infection   T047   C0854135
25763772   67   82   cystic fibrosis  T047   C0010674
25763772   83   120   Pseudomonas aeruginosa (Pa) infection   T047  C0854135
⋮
```

3.2 Flair and SapBERT

After collecting all the PubTator files, the first thing to do is to collect the embeddings of the mentioned entities in the documents. To do so, I decided to use the Flair framework and their TransformerWordEmbedding class. This object requires a pre-trained model as an input and I choose to use the one produced by Liu et al. [2021]. SAPBERT is a pretraining scheme that self-aligns the representation space of biomedical entities. It is a scalable metric learning framework based on the UMLS entities. Previous pipeline-based hybrid systems didn't offer a one-model-for-all solution like SAPBERT does. It is focused on the NED task in medical topics and achieved a new state-of-the-art on six biomedical benchmarking datasets.

After testing some examples, I preferred this pre-trained model over the bert-case-uncased, which is one of the most used transformers models pretrained on a large corpus of English data in a self-supervised fashion.
The test that I executed started with choosing two popular entities in the biomedical domain: *penicillin* and *cystic fibrosis*. I wrote four sentences for each entity. Three of them are taken from a basic internet search and the last one is a no-sense sentence to trick the model in a biomedical semantic way. The idea of this test is to check if the models can produce a really different embedding for the entity in the *fake sentence*, since it has no scientific meaning but has still a correct syntax. If the model can manage to do so, it means that it is working properly and can recognise the semantic of the sentences. The embedding produced from each model is compared between each other through a cosine similarity. The higher the cosine similarity, the closer the word embeddings and the incorporated meaning is.
For the *penicillin* the true sentences were:

1. Penicillins are a group of antibiotics used to treat a wide range of bacterial infections
2. The penicillins are chemically described as 4-thia-1-azabicyclo (3.2.0) heptanes.
3. Phenoxymethylpenicillin is a type of penicillin antibiotic. It is used to treat bacterial infections, including ear, chest, throat and skin infections

And the *fake sentence* was:

- The penicillin is a tumor that can evolve in the brain.

The result of the cosine similarity obtained are shown in the two tables 3.2 and 3.3.

	First	Second	Third	Fake
First	1.0	0.7223	0.8080	**0.5334**
Second	0.7223	1.0	0.7014	**0.4522**
Third	0.8080	0.7014	1.0	**0.5412**
Fake	0.5334	0.4522	0.5412	1.0

Table 3.2. SapBERT model penicillin

	First	Second	Third	Fake
First	1.0	0.7276	**0.5965**	0.7206
Second	0.7276	1.0	**0.6569**	0.7415
Third	**0.5965**	0.6569	1.0	0.7325
Fake	0.7206	0.7415	0.7325	1.0

Table 3.3. bert-base-uncased model penicillin

It is also interesting to consider the case of the *cystic fibrosis* entity, because it is composed of two words. The idea here is to do an arithmetic mean between the embedding of the word *cystic* and *fibrosis*.
Here the true sentences were:

1. Cystic fibrosis is a genetic condition. It is caused by a faulty gene that affects the movement of salt and water in and out of cells.
2. Cystic fibrosis is an inherited disorder that causes severe damage to the lungs, digestive system and other organs in the body.

3. The primary cause of morbidity and death in people with cystic fibrosis is progressive lung disease.

And the *fake sentence* was:

- The vaccine for the new COVID variant cystic fibrosis is going to be released soon.

The results of the cosine similarity obtained are shown in the two tables 3.4 and 3.5.

	First	Second	Third	Fake
First	1.0	0.8241	0.7425	**0.6609**
Second	0.8241	1.0	0.8429	**0.7034**
Third	0.7425	0.8429	1.0	**0.6843**
Fake	0.6609	0.7034	0.6843	1.0

Table 3.4. SapBERT model cystic fibrosis

	First	Second	Third	Fake
First	1.0000	0.9019	0.7777	**0.7315**
Second	0.9019	1.0000	0.8667	**0.7754**
Third	**0.7777**	0.8667	1.0000	0.8495
Fake	0.7315	0.7754	0.8495	1.0000

Table 3.5. bert-base-uncased model cystic fibrosis

In bold in the tables are highlighted the lowest similarity score for the first three sentences, the lowest the score the more difference there is between the word embedding. Of course on the diagonal there are always one, since it is calculated the similarity between the entity in the sentence and itself.
In the case of the SapBERT model, the lowest results are always against the entity in the *fake sentence*, which let us understand that the model can differentiate the entities for their biomedical meaning.
For bert-base-uncased model, the fake penicillin sentence never reports the lowest similarity score, which means that the pre-trained model can not properly differentiate the entity only from the embedding. Instead, for the cystic fibrosis example, the model can guess the lowest similarity score only in two out three sentences but in these cases it is however a 10% higher than the SapBERT model.

So after these results, I conclude that the bert-base-uncased model is a good sentence embedding when we are considering only the syntax of the sentences but SapBERT performs without any mistake in both biomedical entities taken in consideration for these examples. This is the reason why I chose to embed all the entities in the dataset with the SapBERT model.

3.3 SNOMED CT graph building

In this section it will be explained how the support KG has been built. Python api for Neo4j were used to communicate with the database to construct all the nodes and the relationships found in SNOMED CT.

The KG will be fundamental in the candidate selection phase, since it will retrieve the best result from a particular text search. It is in fact the KB that will be used in this work. I downloaded the SNOMED CT International edition of March 2023. Inside this repository there are two really important files over the others.

- the first one contains all the descriptions of the entities in SNOMED CT, so the most important columns are the one with the *conceptId* and the *term*.
- the second file has all the relationships between these entities. We can find the head of a relationship under the column *sourceId* and the tail under *destinationId*.

So for each entity in the description file a *SnomedEntity* node was built. Thanks to the information in the same file, to each node different property were added:

- *id*, it is the unique SNOMED CT ID to refer to a certain entity

- *name*, it is the first way the entity is presented in the database
- *aliases*, it is a list of the different string option of how the same entity can be found in a document
- *umls*, it is the corresponding UMLS code

The relationship built were of two kind: *SNOMED_RELATION* and *SNOMED_IS_A*
A small portion of the dataset can be seen in figure 3.3.

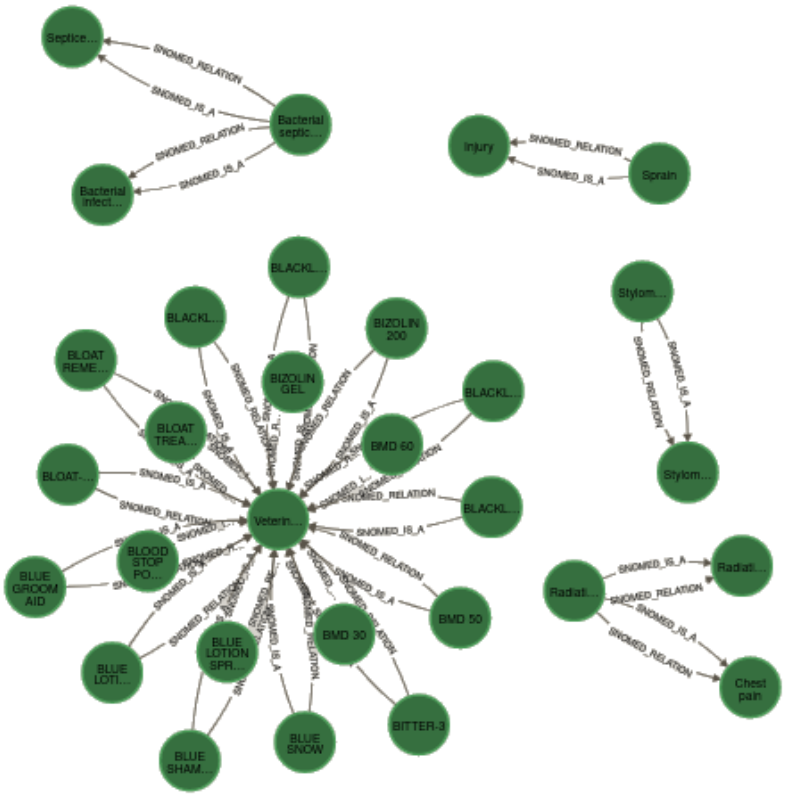

Figure 3.3. Example of a small portion of the KG

3.4 Negative retrieval

To train the model it is necessary to give in the training test some correct examples (positive) and some incorrect ones (negative).
The novelty of the thesis rely on the construction of these array that will be used in the training. The integration of a KG embedding in concatenation with a text embedding is a fusion of two different ingredient that can learn between each other the necessary information for the disambiguation. From the KG embedding, it can be taken all the topology of the KG and the relationship stored between the nodes and trying to find a pattern with the text embedding from the annotated documents through a robust NN.

In fact, the positive examples are tensor array of 896 values, as shown in figure 3.4:

1. the first 128 positions are occupied by the graph embedding of the Node2Vec model, taken from the work of Pattisapu et al. [2020].
2. the second part of the array is also 768 values but given from the correct embedding of the entity calculated from the SapBERT model.

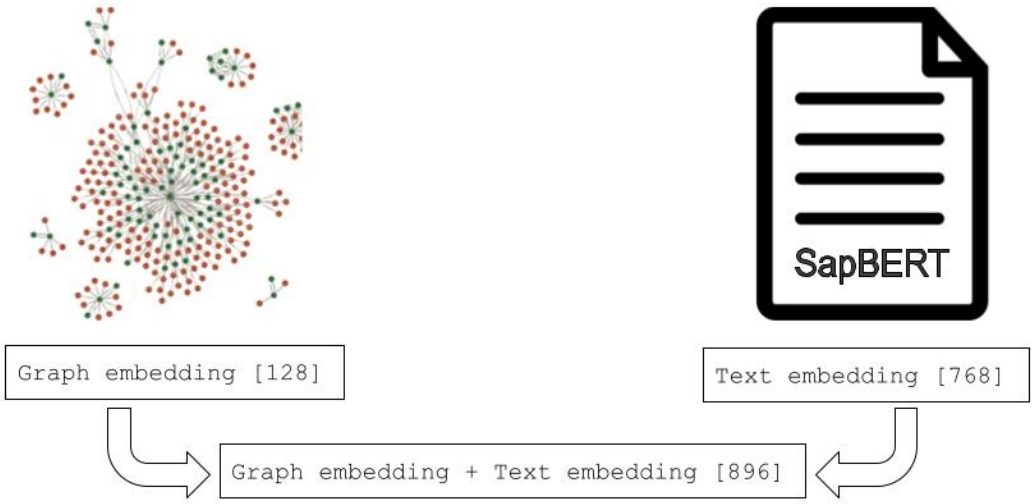

Figure 3.4. Idea of the vector concatenation used for testing

Instead, the negative examples have the same length but the first 128 positions are filled by a graph embedding of a wrong entity.

A first attempt to create the negative example was to use the Nearest Neighbours algorithm from the scikit-learn python library [Pedregosa et al. [2011]], to collect the 5 closest entities of the one I wanted to create the negative example. The sklearn.neighbors library offers both possibility to do supervised and unsupervised neighbour based learning techniques:

- Unsupervised nearest neighbours it is useful for different machine learning approaches, such as manifold learning and spectral clustering.
- Supervised neighbour-based learning comes in two forms: classification for data with discrete labels and regression for data with continuous labels.

The fundamental idea behind nearest neighbour methods is to identify a predefined number of training samples that are closest in distance to a new data point, and then predict its label based on these neighbouring samples. In this example I was not focusing on the label but only on the distance between the selected neighbours. So the class used was sklearn.neighbors.NearestNeighbors. The number of samples can either be a constant specified by the user (referred to as k-nearest neighbour learning) or can vary according to the local density of points (referred to as radius-based neighbour learning). In this case I chose to use a fixed k such as the 2.5% of the number of total entities, so $k = 9185$, since I only needed the first 5 closest neighbours. The distance metric used can be any suitable measurement, but I chose the standard Euclidean distance being the most commonly chosen option. These neighbour-based methods are considered non-generalizing machine learning techniques because they essentially "memorise" all the training data.

Beyond these five wisely selected entities, I choose to try two different approaches:

- select other five random entities embedding to have in total ten negative examples;
- not selecting any other negative examples and having to deal only with five.

Another approach that I tried is collecting some negative examples using a similar technique of the candidate selection that will be explained in section 3.6. For each entity I collected at least five negative examples, if available, from Neo4j's full-text search index. However, if the number of negatives collected with this method were less than five, I decided to add the negatives of the nearest neighbours with random entities to arrive at a maximum of 10 negatives per entity. The

Nearest Neighbours	
Atrial fibrillation non-rheumatic	C0340490
Permanent atrial fibrillation	C2586056
Arrhythmia, atrial	C0085611
Persistent atrial fibrillation	C2585653
Rapid atrial fibrillation	C1281999

Full-text search	
Ione atrial fibrillation	C0340489
Controlled atrial fibrillation	C0577699
Rapid atrial fibrillation	C1281999
ECG: atrial fibrillation	C0344434
Permanent atrial fibrillation	C2586056

Table 3.6. Negative entities comparison for the nearest neighbours and for the full-text search

negatives for the full-text search consider more the semantics of the word embedding found in the text because the search is pursued by looking for the text mention in the KG. Instead, the first idea is related to understanding the spatial location of the embedding in the graph but this method can't be used during the candidate selection since it needs as input the exact node.

In table 3.6 there is a comparison between the two methods of negative retrieval and in table 3.7 there is a summary of the three methods used to collect the negative examples. For each of them, a proper trainset will be built and used to train the NN.

Method used	Number of negatives collected
Nearest neighbours + Random entities	10
Nearest neighbours	5
Full-text search	5 or 10

Table 3.7. Summary of the different methods used for the negative retrieval

3.5 Model training on NN

In this section, I want to describe how I create the model that will be trained at first and then used for the test phase for the NED task.

One of the novelty of the thesis relies on the input of the NN, which is a vector that combines a KG embedding and a text embedding. Here the model wants to understand the relationship and the similarities between the two parts of each vector passed in the training phase to guess later in the testing phase if the text embedding of the word found in the annotated document corresponds to the proposed entity retrieved in the candidate selection phase.

To implement the NED task, the training phase will be performed using the positive and negative examples described in the previous section. The task at this moment will be a binary classification in which the model will have to predict 1 for the positive examples and 0 for the negative ones.

The model used for the classification is a Siamese Neural Network (SNN). It is a type of NN architecture that is commonly used for tasks involving similarity or dissimilarity measurement between pairs of input data. In this case such as the embedding from the KG and the embedding in the annotated text. The term *Siamese* comes from the concept that the network architecture consists of two identical subnetworks that share the same architecture and parameters. The only difference between these two subnetworks is the length of the input vector, since the graph node embedding is a vector of length 128 and instead, the text embedding from SapBERT is long 768, but it will be better explained in 3.6.

To create the NN I used Pytorch [Paszke et al. [2019]], an open-source deep learning framework that provides two high-level features:

- Tensor computation which provides a multidimensional array called "tensor" that is similar to a NumPy array but with the added ability to perform GPU acceleration for numerical

computations. Thanks to this PyTorch it is suitable for high-performance machine learning tasks.

- Deep neural networks that can be used for building and training neural networks. It is possible to create various types of neural network architectures, including feedforward networks, convolutional neural networks (CNNs), recurrent neural networks (RNNs) and more.

The library that I used mostly was torch.nn, with which you can easily build blocks for a neural network. The torch.nn.Linear function applies a linear transformation to the input data x into the y output vector:

$$y = xA^T + b$$

Then I chose the binary cross-entropy loss (torch.nn.BCELoss) as a criterion to train the network, which is a common choice for binary classification, so considering two classes labelled 0 and 1. It measures the dissimilarity between predicted probabilities and actual binary labels for each example in a dataset. Then a mean is computed to calculate the average loss in the batch.

In formula, BCE loss it is defined in the following way for a batch size of N samples:

$$L = \frac{1}{N}\sum_{i=1}^{N} l(y_i, p_i)$$

where:

$$l(y_n, p_n) = -[y_n log(p_n) + (1 - y_n) log(1 - p_n)]$$

y_n is the true binary label and p_n is the predicted probability that the example belongs to class 1. Since the goal during the training is to minimise the BCE loss, this formula penalises incorrect predictions. For instance, if $y = 1$, the first term encourages p to be close to 1, instead if $y = 0$ the second term encourages p to be close to 0. It encourages the model to produce probabilities that are close to the true class probabilities and it also penalises confidently wrong predictions heavily, making it robust to situations where the model's predictions are far from the true labels.

The minimization is typically achieved using optimization algorithms like gradient descent or its variants, such as Adam.

Adam optimizer [Kingma and Ba [2017]] from the library torch.optim. It is a popular choice to train NN thanks to his adaptive learning rate which often converges faster and requires less manual tuning of learning rates compared to traditional stochastic gradient descent, being its extension.
Adam combines elements of two other popular optimization algorithms: AdaGrad, which works well with sparse gradients, and RMSProp, which works well in on-line and non-stationary settings. It maintains two moving averages of the gradients: the first moment (mean) and the second moment (uncentered variance). These moving averages are used to adaptively adjust the learning rates for each parameter. In fact Adam stands for adaptive moment estimation. It offers several benefits: the first is that the size of parameter updates remains invariant to rescaling of the gradient. Additionally, its step sizes are approximately limited by the step size hyperparameter. Unlike other optimization methods, Adam doesn't demand a stationary objective, and it functions effectively with sparse gradients. Moreover, it naturally incorporates a form of step size annealing.

The activation functions used are the ReLU in the hidden layers and the Sigmoid for the output layers that will give the final result to choose the correct class. This two formula applies element-wise the following function, that it are shown in figure 3.5:

$$ReLU(x) = (x)^+ = max(0, x)$$

$$Sigmoid(x) = \sigma(x) = \frac{1}{1+exp(-x)}$$

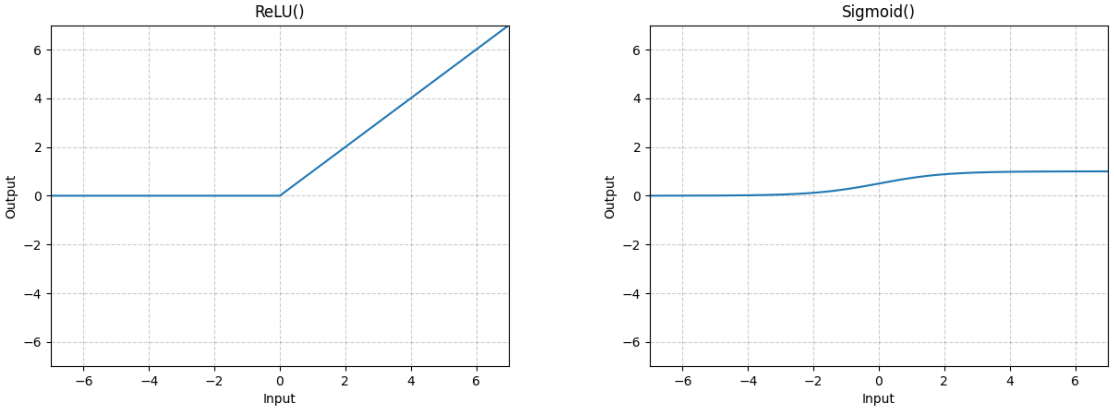

Figure 3.5. Graph of the activation function used

To upgrade the SNN I decided to include some dropout layers [Hinton et al. [2012]]. Between the input and output layers, in a feedforward neural network there are the so-called hidden layers. The concept behind dropout is to modify the weights on the incoming connections to these hidden units in order to learn feature detectors that can predict the correct output for a given input vector. If the number of hidden units is sufficiently high, there are often multiple weight configurations capable of almost perfectly modelling the training data, especially when the amount of labelled training data is limited. However, these different weight configurations tend to perform less effectively on unseen test data because the feature detectors have been tuned to work perfectly together specifically on the training data, not the test data. To address this, dropout is employed to mitigate overfitting. During each presentation of a training case, each hidden unit is randomly deactivated with a fixed probability, so that a hidden unit cannot rely on the presence of other hidden units.
Another feature of dropout is that it efficiently accomplishes model averaging with neural networks. To enhance performance on the test dataset, it is beneficial to average predictions generated by a large number of diverse networks. Traditionally, this involves training numerous separate networks and then applying each one to the test data. However, this approach is computationally demanding during both the training and testing phases. Random dropout offers a practical means to train a multitude of different networks in a reasonable amount of time. It is highly likely that there exists a distinct network for each presentation of each training case, yet all these networks share the same weights for the active hidden units.

A second way to upgrade the NN is using normalisation layers to speed up the training. The function torch.nn.BatchNorm1d applies batch normalisation over an input array as described in the paper of Ioffe and Szegedy [2015]. The idea behind it is to normalise each feature independently, making it have zero mean and variance of 1. For a layer with d-dimensional input $x = (x_1, \ldots, x_d)$), each dimension will be normalised in

$$\hat{x}_k = \frac{x_k - E[x_k]}{\sqrt{Var[x_k]}}$$

where the expectation and variance are computed over the training data set. This will speed up convergence, even when the features are correlated. But simply normalising every input of a layer could change the representation of the layers. To deal with this, a pair of parameters γ and β are added to scale and shift the normalised value:

$$y_k = \gamma_k \hat{x}_k + \beta_k$$

By default, the elements of γ are set to 1 and the elements of β are set to 0. At train time these parameters are learned along with the original model parameters, and restore the representation

power of the network. Another upgrade with the batch normalisation is that now too-high learning rate may not result in the gradients that explode or vanish, because it prevents small changes to the parameters from amplifying into larger and suboptimal changes in activations in gradients. Instead of getting stuck in poor local minima, batch normalisation can prevent the issue. In some experiments, it has been shown that with batch-normalised networks it is even possible to remove or reduce the dropout.

Finally we can give to the NN the input: it needs a TensorDataset class. It is used to collect the input features and the label. The train is performed with a batch size of 64 input data and a total of forty epochs. Every ten epochs the learning rate of the optimizer decreases: this is a common technique that offers several advantages, helping the network converge faster and reach better solutions. In fact, in the early stages of training, a high learning rate allows the network to make large weight updates, helping it quickly escape local minima. However, as training progresses, it's desirable to slow down the learning rate to make smaller and more precise weight adjustments. This can lead to faster convergence and quicker reduction of the loss function.

The SNN proposed will be trained on the three different train-set produced by the negative explained in section 3.4 and the one with the best global F1 score on each train set will be used then for the NED task. So a final number of three NN, for each PubTator dataset, will be used in the final testing phase.

The result of this training will be shown in chapter 4.

3.6 Candidate selection and NED testing

Candidate selection is a crucial step for the success of the NED task. In this section it will be presented why and how the candidates were selected. Later on the testing model will be explained since the candidate selection is its first phase.

After training the NN on the initial data, it is time to test the model on unseen annotated documents. To do this for each paper I collected the available mentioned entities and checked for each one of them a list of possible exact candidates using the full text index of Neo4j.

As explained in section 3.3, a KG with all the SNOMED entities and relationships was built with python using the available Neo4j api. The Neo4j full-text search index serves as a powerful tool for indexing nodes and relationships based on string properties. This type of index enables you to build queries that locate matches within the content of indexed string properties. Unlike some other functions in Neo4j, such as range and text indexes, which have limited capabilities for matching strings (exact, prefix, substring, or suffix matches), a full-text index takes a different approach. It tokenizes the values of indexed strings, allowing it to find terms anywhere within those strings.

The specific way in which the indexed strings are tokenized and segmented into terms depends on the configuration of the full-text index, particularly the choice of analyzer. For example, the Swedish language analyzer is equipped to tokenize and stem Swedish words, and it excludes common Swedish stop words from indexing. A complete list of stop words for each analyzer is available in the output of the *db.index.fulltext.listAvailableAnalyzers* procedure.

Building only the SNOMED KG was not enough to perform a proper full-text search for the candidate selection, since the papers were mainly annotating UMLS entities. The entities I was working with in the KG were 375180 in total. So to support the candidate selection more than 1.5 million nodes were added to the KG. These nodes contain different English names, with no repetition found in the file MRCONSO.RRF. These nodes are linked to the SNOMED main node with a relationship *HAS_UMLS_NAME*. In total 14705 out of the 15264 unique entities considered in the both PubTator datasets had a new relationship. Instead, considering the whole KG graph, 364184 nodes had a new same relationship. An example of the new relationship built is provided in figure 3.6.

Thanks to this relationship built, the full-text search index can perform better on the candidate selection during testing. This is because it performs the search also across the new added nodes.

Figure 3.6. New relationship built for cystic fibrosis

To show the improvement of the search, two different indexes in Neo4j will be shown. The difference code-wisely it is really small, as you can see in the figure code 3.7, but the function it is really different

```
CREATE FULLTEXT INDEX small
FOR (s:SnomedEntity)
ON EACH [s.name]
```

```
CREATE FULLTEXT INDEX candidate
FOR (s:SnomedEntity|UMLS_NAME)
ON EACH [s.name]
```

Figure 3.7. Cypher queries for the creation of the two indexes

- the code on the left is the one to create the index *small*. It is based only on the *SnomedEntity*, which is the one on the initial graph with the embeddings.
- instead, the code on the right is used to perform the candidate selection and the negative retreival by the index *candidate*. On the *FOR* line, also the nodes *UMLS_NAME* are added. This node will be explained shortly as they are essential to obtain the best candidate selection.

An example of the different results given from the two full-text search queries in figure 3.8 is provided in the tables 3.8 and 3.9.

In the tables different results are shown. First of all the queries are a bit different since the nodes *UMLS_NAME* have the property *node.code* instead of the *node.umls*. This is done on purpose so it is easy readable from the table 3.9 which result comes from a node *SnomedEntity*,

```
CALL db.index.fulltext.queryNodes("small","Pseudomonas aeruginosa")
YIELD node, score
RETURN node.name, node.umls, score
LIMIT 5
```

```
CALL db.index.fulltext.queryNodes("candidate","Pseudomonas aeruginosa")
YIELD node, score
RETURN node.name, node.umls, node.code, score
LIMIT 5
```

Figure 3.8. Queries for the full-text search with different indexes

node.name	node.umls	score
Pseudomonas aeruginosa	C0033809	10.144
Pseudomonas aeruginosa paronychia	C0406109	9.140
Mucoid Pseudomonas aeruginosa	C4518829	9.140
Pseudomonas aeruginosa antibody	C0443682	9.140
Pseudomonas aeruginosa mastitis	C0276076	9.140

Table 3.8. Full-text search result for *Pseudomonas aeruginosa* with the index *small*

property *node.umls* not null, and from a node *UMLS_NAME*, property *node.code* not null. In table 3.8 we have different *node.umls* cells because the search is done only on the *SnomedEntity* nodes, which are unique for each code. Instead, the result from the index *candidate*, it seems to say that the correct entity is the one with UMLS code C0033809. This search is done also on the *UMLS_NAME* node, which as we can see are really helpful also if the mention in the text could be a bit different, such as inverted words or in capital letters.

As we can see from the tables a score is calculated for each result. For the testing phase the mention in the text will be passed to the full-text index query. The first 10 results will be taken and the relative graph embedding will be obtained and concatenated to the text embedding of the word found in the text. Then this concatenated vector will be passed to the NN and the final score for each entity will be provide. The highest score will be taken as the top entity but the first 5 will count in the accuracy.
There is still a problem to consider. Let's bring back the initial example sentence:

The purpose of this study was to investigate the role of DCTN4 missense variants on Pa infection incidence

The problem in this sentence is that the entity we are trying to find, *Pseudomonas aeruginosa*, is written as an acronym, *Pa*. In fact if we perform the following code on Neo4j:

```
CALL db.index.fulltext.queryNodes("candidate","Pa")
YIELD node, score
RETURN node.name, node.umls, node.code, score
LIMIT 5
```

the correct entity won't be found (table 3.9). In this case it is not even in the first 20 candidates, so the NN for sure will fail to perform a correct disambiguation. The problem is that acronyms

node.name	node.umls	node.code	score
Pseudomonas aeruginosa	C0033809	*null*	10.588
aeruginosa pseudomonas	*null*	C0033809	10.588
pseudomonas aeruginosa	*null*	C0033809	10.588
PSEUDOMONAS AERUGINOSA	*null*	C0033809	10.588
Mucoid Pseudomonas aeruginosa	C4518829	*null*	9.534

Table 3.9. Full-text search result for *Pseudomonas aeruginosa* with the index *candidate*

are really used in a biomedical context and a lot of them can have completely different meanings as we can understand from the previous table.

node.name	node.umls	node.code	score
Pa	*null*	C0030532	6.794
PA	*null*	C0030853	6.794
u-pa	*null*	C0042071	6.034
hb PA	*null*	C0062424	6.034
SCU-PA	C0141582	*null*	6.034

Table 3.10. Full-text search result for *Pa*

That's why it is important first to detect the correct abbreviation and second pass the whole entity to the full-text search.

To perform the abbreviation detection it has been used the ScispaCy additional pipeline component [1]. The AbbreviationDetector is a Spacy component which implements the abbreviation detection algorithm described in the paper of Schwartz and Hearst [2003].

The algorithm initiates by first extracting abbreviations and their corresponding definitions from medical text, involving a two-step process. The initial step entails identifying <short-form, long-form> candidate pairs within the text. Subsequently, the second step focuses on pinpointing the correct long form within the sentence surrounding the short form. Many approaches, including the one discussed here, adopt a similar strategy for discovering candidate pairs. These abbreviation candidates are determined based on their proximity to parentheses. Two scenarios are considered:

1. The long form precedes the short form within parentheses.

2. The short form precedes the long form within parentheses.

In practice, the majority of <short form, long form> pairs adhere to the first pattern. When the content inside the parentheses encompasses more than two words, the second pattern is assumed, and the short form is sought immediately preceding the left parenthesis (word boundaries are indicated by spaces). Short forms are deemed valid candidates if they meet specific criteria: they consist of a maximum of two words, have a length ranging from two to ten characters, contain at least one alphabetic character, and begin with an alphanumeric character. For simplicity, the discussion below assumes the first pattern.

The subsequent phase involves identifying potential long form candidates. To qualify, a long form candidate must appear within the same sentence as the short form and should not exceed the minimum of either (|A| + 5) or (|A| * 2) words, where |A| represents the character count in the short form. The algorithm operates under the assumption that long forms are found only immediately adjacent to the short form. In the case of a given short form, a long form candidate

[1] https://github.com/allenai/ScispaCy#abbreviationdetector

comprises consecutive words extracted from the original text that encompass the word directly preceding the short form.

Following the initial identification process, a list of potential long-form words for the given short form is created, and the objective is to select the most appropriate subset of these words. The fundamental approach involves examining both the short and the long form from right to left in order to identify the shortest long form that corresponds to the short form. It mandates that every character in the short form aligns with a character in the long form, maintaining the same order. However, there is one exception: the initial character of the short form must correspond to the first character of the first word in the long form, even if it's a character that connects words through hyphens or other non-alphanumeric symbols.

In conclusion, to work properly, this algorithm needs as input the complete text of the document, since it won't perform well on small sentences where the abbreviation explanation is not present.

Chapter 4

Result

This section provides the results obtained throughout the study. It is organized into two key segments: it begins with the outcomes pertaining to the training phase and it follows with the results for the final model's performance in the NED task.

4.1 Training phase

The training phase involved the evaluation of various configurations and settings. As previously elaborated in Chapter 3.4, different negative retrieval methods were introduced. During this phase, several SNNs were constructed based on diverse training sets which were employed to train these networks effectively.

It is necessary to keep in mind that the testset used to verify the model's performance is unbalanced as much as the trainset:

- for the nearest neighbours (no random) trainset there is one positive example for five negative ones;
- for the nearest neighbours with random examples and for the full-text, the negative examples are up to ten.

So for this reason the F1 score was used to check the quality of the nets. It is a metric commonly used in machine learning and statistics to measure the quality of a binary classification model when the dataset is unbalanced. It takes into account both precision and recall, two metrics that are good for a balanced dataset, to provide a single score that balances these two factors. Before understanding these metrics, it is important to know the concepts of True Positives (TP), False Positives (FP), True Negatives (TN) and False Negatives (FN). TP are the instances that are correctly classified as positive, and FP are the instances that are incorrectly classified as positive. TN are the instances that are correctly classified as negative, and FN the instances that are incorrectly classified as negative. So the definitions are the following:

- Precision is a measure of how many of the positively classified instances were correctly classified. It is calculated as:

$$Precision = \frac{True\ Positives}{True\ Positives + False\ Positives}$$

- Recall is a measure of how many of the actual positive instances were correctly classified. It's calculated as:

$$Recall = \frac{True\ Positives}{True\ Positives + False\ Negatives}$$

The F1 score is defined as the harmonic mean of precision and recall, and it provides a balance between these two metrics. The formula for the F1 score is:

$$F1\ Score = 2 * \frac{(Precision * Recall)}{(Precision + Recall)}$$

The F1 score ranges between 0 and 1, with higher values indicating better model performance. A high F1 score indicates that the model has both high precision and high recall, which means it is good at correctly classifying positive instances while minimizing false positives and false negatives.

Then, all three different negative retrieval methods are evaluated for each dataset. The results of the best F1 score obtained during the 40 training epochs are shown in 4.1.

Negative retrieval method	**MedMentions**	**BC5CDR**
Full-text	0.6631	0.7732
Nearest Neighbours	0.7730	0.8853
Nearest Neighbours (no random)	0.7963	0.8840

Table 4.1. F1 score for the different testset built

In the training phase, the underlying concept of the SNN was to discern whether the node graph embedding, which is 128 units in length, exhibited similarity to positive text embeddings, which were 768 units in length, while distinguishing itself from negative examples.

The results for the network constructed for the Nearest Neighbours methods, both with and without the random entities, are notably promising, especially in the case of the MedMentions dataset. However, they are even more impressive for the BC5CDR dataset. This could be attributed to the BC5CDR dataset having only two entity types, making it easier for the network to comprehend the entities in use. Nevertheless, it's essential to acknowledge that nearly half of the entities in the testing set were absent from the training set.

Another observation is that the Nearest Neighbours method without random entities outperforms its counterpart with random entities, particularly in the context of the MedMentions dataset. Conversely, in the case of the BC5CDR dataset, the results are comparable. This aspect will gain further significance when we delve into the results of the testing phase.

Unfortunately, achieving a robust F1 score for the full-text negatives proved to be a challenging endeavor. This becomes particularly noteworthy as this network is likely to be utilized in the subsequent testing phase. Commencing with a lower F1 score may pose challenges when selecting the most appropriate entity between the candidates in the testing phase.

4.2 Testing phase

Finally, in this section, the NED model will be put all together and then tested.

For every entity within the document, a full-text search within the Neo4j database is performed. This search returns the first 50 results, if available. Subsequently, for each distinct UMLS code obtained, a concatenation is carried out with the corresponding UMLS code embedding and the embedding of the entity text found in the document. This concatenated vector is then fed into the Semantic Neural Network (SNN), which assigns a score to each UMLS entity. The highest score is considered for the gold accuracy, assessing the model's ability to precisely identify the entity on its initial attempt.

Alternatively, if the exact entity is found within the first five results, it is counted as a partial accuracy. This implies that the model still has four additional opportunities to identify the correct entity. An illustrative overview of the testing process is in figure 4.1.

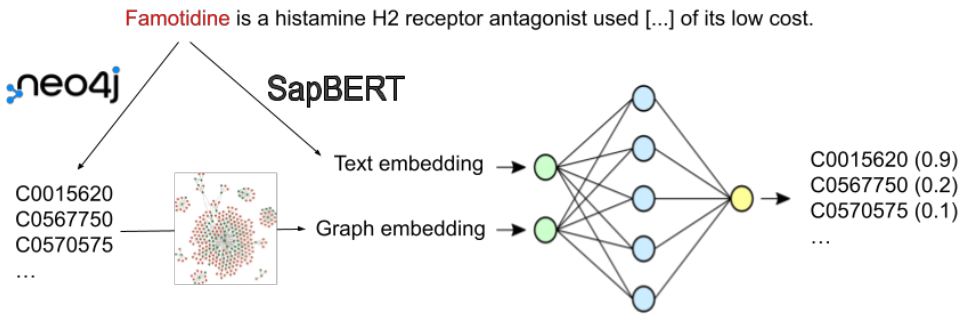

Figure 4.1. Idea of the pipeline followed in the testing phase

In tables 4.2 and 4.3, the accuracy gold and partial of models trained on the different negatives are presented for both test sets, MedMentions and BC5CDR.

The results for the MedMentions dataset are lower compared to the BC5CDR, but this result is as expected, since the net built were less accurate and in general the dataset is more challenging, due to the presence of many entity type. It is peculiar to notice that the SNN trained on the Nearest Neighbours without the random entity performed worse if the SNN had a better F1 score.

Negative retrieval method	Gold	Partial
Full-text	0.42 ±0.14	0.69 ±0.14
Nearest Neighbours	0.41±0.14	0.69±0.14
Nearest Neighbours (no random)	0.33±0.13	0.65±0.14

Table 4.2. Accuracy obtained for different testing models in MedMentions testset

The results obtained for the BC5CDR dataset are good and really similar when looking at the partial accuracy between all the methods tested. Of course the gold accuracy is always lower, since the task to pursue is more difficult, but we can notice a better result for the SNN trained with the full-text negative, also if in the previous section that was the network had the lowest F1 score. This is of course due to the similarity of the negatives retrieval and the candidate selection.

Negative retrieval method	Gold	Partial
Full-text	0.64 ±0.26	0.80±0.20
Nearest Neighbours	0.60 ±0.26	0.80 ±0.20
Nearest Neighbours (no random)	0.52 ±0.28	0.79 ±0.21

Table 4.3. Accuracy obtained for different testing models in BC5CDR testset

Chapter 5

Comparison result

In this section two different methods will be presented to compare the results of the idea proposed in this thesis.

- The first method it follows the same idea of the graph embedding method presented, with the difference that instead of training the neural network with a vector composed of a graph embedding and the text word embedding, I substitute the node graph embedding with the text BERT embeddings, also taken from Pattisapu et al. [2020].
- The second method presented is a possible way of doing the NED task with a really famous framework mainly used for NER. The ScispaCy method will be presented and will be adapted to be compared with the one in this work.

5.1 BERT embedding

This method is based on the similar idea of the one implemented in this thesis. The main change is the creation of the positive and negative examples. Here instead of using the graph embedding from Node2Vec, the text embedding from BERT of the nodes will be concatenated with the text embedding from SapBERT, as sketched in figure 5.1.

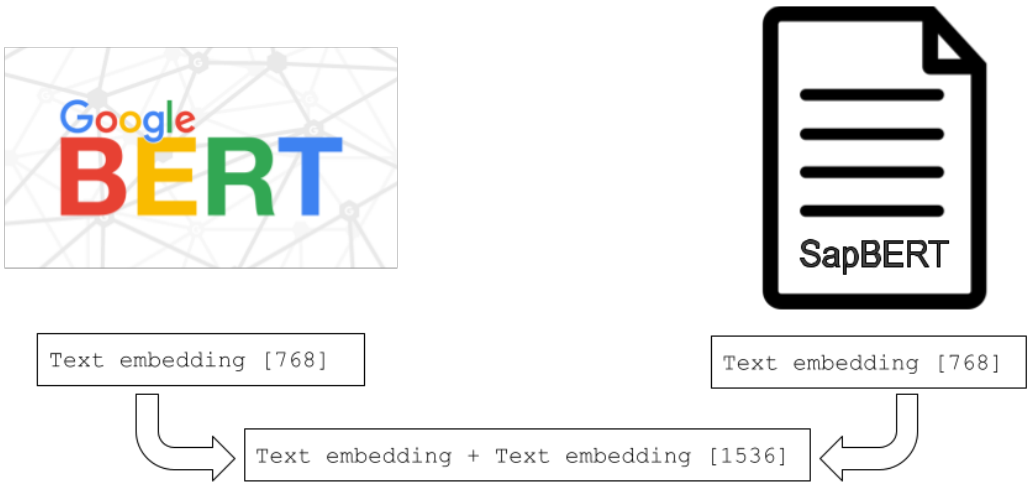

Figure 5.1. Idea of the vector concatenation used with the BERT embeddings

Thanks to the same length of the embeddings of these two pre-trained models, I was able to implement three different neural network, to find which one was the best one for the classification.

The three neural network implemented are:

- a sort of linear model that is the simplest form of neural network architecture. It consists of hidden nodes layer where they apply a linear transformation of the input data but adding some non linearity using as activation function the Relu and the Sigmoid.

- an upgrade version of the latter, with batch normalization and dropout layers.

- the same Siamese network presented in section 3.5, but now the input vector of the node embedding has length 768.

Then all the nine different settings for each dataset are being tested and their results are shown in 5.1. The score in the last two columns is the F1 score, and it is the best score obtained during the forty epochs of training. In bold it is highlighted the highest score for each negative retrieval method, per each dateset.

Model	**Negative retrieval method**	**MedMentions**	**BC5CDR**
Full-text	Net	0.7136	0.7772
	UpgradedNet	**0.7226**	0.7824
	SiameseNetwork	0.7156	**0.7941**
Nearest Neighbours	Net	**0.6681**	0.6710
	UpgradedNet	0.6382	0.6548
	SiameseNetwork	0.6342	**0.7270**
Nearest Neighbours (no random)	Net	0.6804	0.6724
	UpgradedNet	0.667	0.6524
	SiameseNetwork	**0.7053**	**0.7506**

Table 5.1. F1 score for the different testset built

Generally speaking, it can be seen that the score on the BC5CDR dataset is usually higher. This could be due to a lower complexity of the dataset, since fewer entities are present and the types of the entity is also way smaller compared to MedMentions. In fact, in BC5CDR we have only chemicals and diseases.

Another interesting thing is that it is not sure that the more complex the NN is, the better the results are. In fact as we can see from the table, only in the BC5CDR dataset the Siamese Network is a "safe choice". For the MedMentions dataset, different NN models were preferred. In fact, this could also be linked to the bigger size of the MedMentions trainset given to train. The Siamese network suffered a bit the over-fitting, since the maximum score was reached before the ten epochs and then decreased in the following. Also, the UpgradedNet is not always better than the same Net without any dropout or normalization layers.

Moreover the negative retrieval method Full-text is always performing better than the other negatives, in any NN configuration. This could be useful later for the testing phase, since the candidate retrieval is done with the same logic.

For the sake of completeness, I still wanted to try the best NN trained on each negative retrieval, also if I'm expecting that the Full-text negatives will perform better than the others. So the 3 testing models in bold per each dataset will be evaluated for the testing phase. This part it follows the same logic of the method presented before, but here I am using instead the text embedding from BERT to link them with the text embedding from SapBERT.

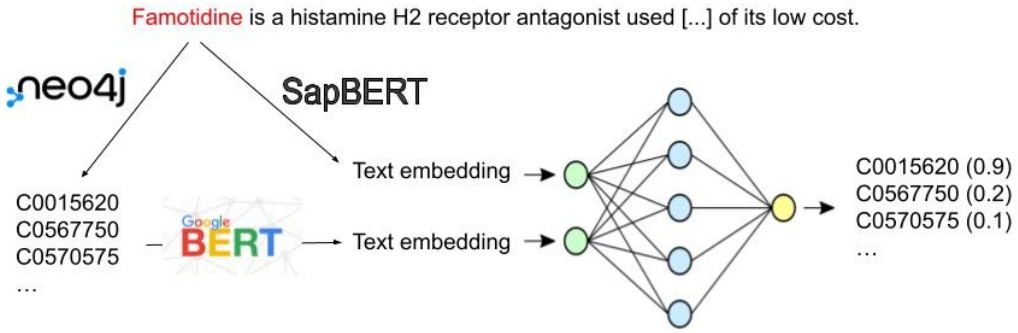

Figure 5.2. Testing phase schema with BERT embeddings

In the tables 5.2 and 5.3, the accuracy gold and partial are calculated for both testset.

Testing model	Gold	Partial
UpgradeNet / Full-text	0.55±0.14	0.74 ±0.13
Net / Nearest Neighbours	0.53 ±0.14	0.73 ±0.13
SiameseNetwork / Nearest Neighbours (no random)	0.22 ±0.11	0.56 ±0.15

Table 5.2. Accuracy obtained for different testing models in MedMentions testset

Testing model	Gold	Partial
SiameseNetwork / Full-text	0.48 ±0.22	0.76 ±0.23
SiameseNetwork / Nearest Neighbours	0.37 ±0.26	0.73 ±0.23
SiameseNetwork / Nearest Neighbours (no random)	0.26 ±0.23	0.64 ±0.24

Table 5.3. Accuracy obtained for different testing models in BC5CDR testset

As we could easily predict the models that were trained on the full-text negative retrieval are generally better on both the gold and the partial result. This is of course given the fact that these kinds of negatives were trained in the same way as the candidate selection in the testing phase is performed. But also the Nearest Neighbours are well performing in the partial accuracy, being close to the values of the full-text model. The usage of random entities in the Nearest Neighbours model seemed that gave to the neural network a better knowledge in prediction. In fact, the model without random entities usually performs worse, in both gold and partial accuracy.

Let's now compare these results with the previous one from the testing model with the graph embeddings. To help the visualization of the scores, I took in consideration only the best result obtained for both testing methods and reported them in tables 5.4 and 5.5.

Testing model	Gold	Partial
SiameseNetwork / Full-text / Node2Vec embedding	0.42 ±0.14	0.69 ±0.14
UpgradedNet / Full-text / BERT embedding	0.55±0.14	0.74 ±0.13

Table 5.4. Accuracy comparison between different testing models in MedMentions testset

Testing model	Gold	Partial
SiameseNetwork / Full-text / Node2Vec embedding	0.64 ±0.26	0.80 ±0.20
SiameseNetwork / Full-text / BERT embedding	0.48 ±0.22	0.76 ±0.23

Table 5.5. Accuracy comparison between different testing models in BC5CDR testset

Before commenting on the result, I wanted to remember that the BERT embeddings have length 768, instead the Node2Vec 128. This means that if we take two identical net with the same hidden layers structure, the net with the BERT embedding will still have more parameters since the input vector is longer. For the MedMentions dataset there is an advantage for the testing model with BERT embeddings, but in the partial accuracy setting the gap between the two models shrinks. The problem here could be due to the poor quality of the SNN that I could train for the Node2Vec model, compared to the best net found for the BERT model, the UpgradedNet. Analysing the result from the BC5CDR testset, it can be noticed an exchange of the best model. Here, also if the best F1 score was obtained by the SNN trained with BERT embeddings, the result of the accuracy are better for the model with Node2Vec embeddings.

In conclusion for this first comparison results, I understood that the quality of the NN used can influence a lot the final result on the accuracy of the testset. But, when we are taking into account a similar quality NNs, or even a less quality ones for the Node2Vec model, the graph embeddings can give a boost to the framework and gave a better accuracy score.

5.2 ScispaCy

The result obtained in the previous chapter will be also compared with ScispaCy [Neumann et al. [2019]]. It is a NLP library built on top of the popular spaCy [Honnibal et al. [2020]] library, but it extends the capabilities to handle and analyze text in the biomedical and scientific domains.

ScispaCy includes pre-trained models and resources tailored for tasks such as NER, part-of-speech tagging, dependency parsing and this makes it particularly useful for applications like extracting key information from scientific articles, medical records, or any text related to the life sciences.

The library provides tools for tokenization, language modeling, and other NLP tasks that can be essential for researchers, healthcare professionals, and organizations working with scientific and biomedical text data. It has become a valuable resource for the development of NLP applications in these domains.

To perform a sort of NED task in ScispaCy it is important to introduce a fundamental component in the library. The *EntityLinker* is a SpaCy component which performs linking to a knowledge base, in this case the UMLS KB was chosen for testing purposes. The linker simply performs a string overlap based search (char-3grams) on named entities, comparing them with the concepts in a knowledge base using an approximate nearest neighbours search.

During the testing phase, ScispaCy has to perform a NER and a NED task, since it cannot receive in input an annotated document such as the one I worked with. In fact, the first step is to recognize all the entities that are in a document. For a correct recognize entity, the first and the last character of the result have to match with the one in the annotated document. Let's say that when this happens we have a *perfect match*. Instead, if ScispaCy can recognize correctly the first char of the entity but not the last one, we will have a *partial match*. After the recognition phase, thanks to the *EntityLinker*, five possibilities of disambiguation are proposed with a decreasing probability. To use the *EntityLinker*, it is needed to choose a pre-trained model. There are several models available on ScispaCy, but to compare with the results obtained through this work, I decided to use three models:

- *en_core_sci_sm*[1], it is a spaCy pipeline for biomedical data with more or less 100 thousand vocabulary;

- *en_core_sci_lg*[2], it is a full spaCy pipeline for biomedical data with almost 785 thousands vocabulary and 600k word vectors;

- *en_core_sci_scibert*[3], it is full a spaCy pipeline for biomedical data with almost 785 thousands vocabulary and allenai/scibert-base as the transformer model.

Since the models in ScispaCy are more focused on the NER task, to compare the disambiguation results of this paper, only the *perfect match* entities are taken into account. Of course, not all the annotated entities in the documents are recognised as *perfect match*. In fact the number of total entities recognised changed from each model chosen, as we can see form table 5.6.

ScispaCy model	MedMentions		BC5CDR	
	Perfect match	*Partial match*	*Perfect match*	*Partial match*
en_core_sci_sm	28	32	10	11
en_core_sci_scibert	28	32	10	11
en_core_sci_lg	29	33	10	12

Table 5.6. Average number of entities per document in the testset

Initially, it is essential to highlight that the primary filter for selecting entities necessitates their association with SNOMED CT embeddings. The *en_core_sci_lg* model exhibits the highest average number of recognized entities, underlining its robust performance. Instead, the *en_core_sci_sm* model achieves comparable results to the *en_core_sci_scibert*, despite being trained on a substantially smaller dataset of 100,000 entities, as opposed to the latter's 785,000 entities.

Nevertheless, it is worth noting that these models cannot achieve flawless entity recognition for all annotated entities within the test sets. MedMentions, on average, contains over 44 entities per document, while BC5CDR boasts an average of 14 entities per document. This observation underscores the significance of having annotated documents for the NED task if we aspire to disambiguate all entities without any omissions.

For an analysis of the accuracy of ScispaCy models, a comprehensive comparison is presented in tables 5.7 and 5.8.

Testing model	**Gold**	**Partial**
sci_sm	0.70 ±0.16	0.82 ±0.14
sci_scibert	0.70 ±0.16	0.82 ±0.14
sci_lg	0.70 ±0.16	0.82±0.14

Table 5.7. Accuracy score for the MedMentions dataset using ScispaCy models

[1]https://s3-us-west-2.amazonaws.com/ai2-s2-ScispaCy/releases/v0.5.3/en_core_sci_sm-0.5.3.tar.gz

[2]https://s3-us-west-2.amazonaws.com/ai2-s2-ScispaCy/releases/v0.5.3/en_core_sci_lg-0.5.3.tar.gz

[3]https://s3-us-west-2.amazonaws.com/ai2-s2-ScispaCy/releases/v0.5.3/en_core_sci_scibert-0.5.3.tar.gz

Testing model	Gold	Partial
sci_sm	0.71 ±0.27	0.76 ±0.24
sci_scibert	0.70 ±0.27	0.76 ±0.24
sci_lg	0.71 ±0.26	0.76 ±0.24

Table 5.8. Accuracy score for the BC5CDR dataset using ScispaCy models

Upon a careful examination of the results, it becomes evident that the disparities among the testing models are subtle, even though there is a substantial contrast in the sizes of their respective training sets. It is plausible to infer that the entirety of entities recognized in the MedMentions and BC5CDR datasets are likely well-represented in all training datasets utilized by the ScispaCy models.

To ensure a fair and meaningful comparison between these results and the previously introduced models, which rely on Node2Vec and BERT embeddings, I opted to select the ScispaCy *sci_lg* model. This choice is driven by the model's capacity to identify the largest number of entities, as highlighted in the preceding table 5.6. Consequently, this approach levels the playing field by subjecting all models to the same set of entities, thereby eliminating any potential advantage stemming from the annotated documents.

Furthermore, for models employing candidate selection by the Neo4j full-text search, I conducted a slight variation in the candidate retrieval process. Instead of solely relying on the first 50 results obtained from the full-text query, I explored the effects of selecting the first 10 and 100 returned entities.

The comprehensive comparison of all models presented in this thesis is encapsulated in tables 5.9 and 5.10. The most noteworthy results for both gold and partial accuracy are highlighted in bold, offering a clear and concise summary of the models' performances in a consistent and equitable manner.

Testing model	Gold	Partial
ScispaCy *sci_lg*	**0.70 ±0.16**	**0.82±0.14**
BERT embeddings 10 results	0.69 ±0.17	**0.82 ±0.14**
BERT embeddings 50 results	0.61 ±0.17	0.79 ±0.14
BERT embeddings 100 results	0.58 ±0.17	0.76±0.14
Node2Vec embeddings 10 results	0.62 ±0.17	0.81 ±0.14
Node2Vec embeddings 50 results	0.48 ±0.17	0.75 ±0.15
Node2Vec embeddings 100 results	0.42 ±0.16	0.72 ±0.16

Table 5.9. Accuracy score comparison for the MedMentions dataset

Testing model	Gold	Partial
ScispaCy *sci_lg*	0.71 ±0.26	0.76 ±0.24
BERT embeddings 10 results	0.65±0.27	0.79 ±0.21
BERT embeddings 50 results	0.52 ±0.28	0.78 ±0.22
BERT embeddings 100 results	0.46 ±0.28	0.75 ±0.24
Node2Vec embeddings 10 results	**0.73 ±0.26**	0.81 ±0.19
Node2Vec embeddings 50 results	0.68 ±0.28	**0.82 ±0.18**
Node2Vec embeddings 100 results	0.61 ±0.29	0.81 ±0.19

Table 5.10. Accuracy score comparison for the BC5CDR dataset

To begin, it is evident that despite the significantly larger training set utilized in implementing the pre-trained ScispaCy model, its performance does not exhibit a significant advantage over the

two models developed within this study. Additionally, it is noteworthy that models leveraging embeddings generally exhibit superior performance when the candidate selection process focuses on the top ten results. This observation highlights the effective performance of candidate selection, even when limited to a relatively small subset of results, and underscores that the neural network's complexity is less pronounced when provided with a smaller set of entities.

Delving into the numerical specifics, we can observe that the highest achieved partial accuracy results across different models are notably similar for the MedMentions testset. It's worth noting that the Node2Vec-based neural network is somewhat less reliable than the BERT-based neural network. However, the disparity in partial accuracy is only marginal, with Node2Vec trailing behind the best result by a mere one hundredth.

Conversely, for the BC5CDR testset, the model with the Node2Vec embeddings yield the most impressive results, both in partial and gold accuracy. While the ScispaCy model approaches the Node2Vec's gold accuracy performance, it exhibits limited improvement when considering a more relaxed scoring metric. In fact, the second-best performing models are those employing BERT embeddings, nearly matching the Node2Vec model's performance. These findings underscore the nuanced performance characteristics of different models across distinct test sets, emphasizing the versatility and potential of the Node2Vec approach in certain contexts.

Chapter 6

Conclusion and future work

In conclusion, this thesis has explored the new interplay between KG embedding and NED, aiming to enhance the understanding of how these two components can complement each other in the context of information retrieval and knowledge representation. Through a systematic investigation and a comprehensive comparison against various models, I have identified the strengths of the proposed approach.

The results have demonstrated that the integration of KG embeddings and NED techniques can yield highly competitive performance when compared to famous ready-to-use models, such as ScispaCy, or to gold standard text embedding models, such as the framework proposed with BERT. The result are comparable also if the initial condition of the model were not the best for the disambiguation, such as the not well performing NN on the MedMentions dataset. Instead, when coming by a well structured NN, such as for the BC5CDR dataset, the result are even better than the comparison models. In fact, an outstanding gold accuracy of 0.73 is reached on this standard benchmark dataset and the best performance on the partial accuracy is reached with a score of 0.82. By fusing the structured knowledge from KG with the context and semantics of a annotated entities, the model has shown the ability to disambiguate entities effectively, improving the overall quality of information retrieval and knowledge representation.

Furthermore, this work has opened up new avenues for research and development, emphasizing the significance of the relationship between these two areas. This exploration of the knowledge graph embedding and named entity disambiguation synergy not only contributes to the ongoing discourse in the field of natural language processing but also holds promise for practical applications in various domains, such as recommendation systems, question-answering systems, and semantic search engines.

The next steps for the work will be to upgrade all the individual phase and section of the model to create an integrated pipeline that communicate from the begging to the end, using data that are related to each other. In fact, in this work different dataset from different developing system were used. A future implementation will be to build a more complete KG, mainly with more relationships between the nodes. From this one, also their embedding can be then extract, using more modern techniques which implements deep learning models. This should improve the quality of the NN since the candidate selection and the graph embedding are related to the same exact KG.

In summary, this thesis has shown that the integration of KG embedding and NED can significantly enhance the state of the art in information retrieval and knowledge representation. The results are not only comparable to other models but also promising, in a future vision of an integrated framework which can even more leverage the structured knowledge and context for more effective information retrieval. This research opens up exciting prospects for future developments and applications in the ever-evolving landscape of information management and natural language understanding.

Appendix A

Python code

All the python code used for this thesis can be found in this appendix or the last updated version on the GitHub repository at the following website:

$$https://github.com/ploppo/KG\_NED/$$

A.1 Import dataset pubtator

```
from pubtatortool import PubTatorCorpus # Library to import PubTator file
import pickle
train_corpus = PubTatorCorpus(['/home/felicepaolocolliani/OneDrive/Tesi/Dataset/'
                               'PubTator_format/BC5CDR/CDR.Corpus.v010516/CDR_Full.txt'])
f = open('BC5CDR_full.obj', 'wb')
pickle.dump(train_corpus, f)
f.close()
```

A.2 Accuracy Scispacy

```
#
# Scispacy NED quality check: understanding the accuracy of their model
#
import spacy
import pickle
from scispacy.linking import EntityLinker

# Loading NER model scispacy choosing between:
# en_core_sci_scibert /en_core_sci_lg
spacy.require_cpu()
nlp = spacy.load("en_core_sci_lg")
print("add pipe start")  # wait, downloading data 1.1 GB
nlp.add_pipe("scispacy_linker", config={"resolve_abbreviations": False,
                                        "linker_name": "umls"})
print("add pipe end")

f = open('Dict_embed_full_.obj', 'rb')
diz_embed = pickle.load(f)
```

```
f.close()
snomed_entities = set(diz_embed.keys())

# Import pubtator format dataset (MM or BC5CDR)
dataset_pubtator = 'MM_test_with_SAPBERT_emb.obj'
f = open('/home/felicepaolocolliani/OneDrive/PycharmProjects/KG_NED/'+dataset_pubtator,'rb')
data = pickle.load(f)
f.close()

f = open('list_index_lg_MM.obj','rb')
list_index = pickle.load(f)
f.close()

# Lists to store gold and partial accuracy
accuracy_gold_list = []
accuracy_partial_gold_list = []
n_entities_file = []

n_document = 0
list_dict_save = []
# Looping on each pubtator document
for document in data:
    print(n_document)
    text = document.raw_text  # Document full text
    doc = nlp(text)  # Text embedding analysis with scispacy model

    dict_mention_scispasy = {}
    for entity in doc.ents:
        if len(entity._.umls_ents) > 0:
            # sono tute le entità trovate con le relative probabilità
            dict_mention_scispasy[(entity.start_char, entity.end_char)] = (
                list(x[0] for x in entity._.umls_ents))

    # Creating dictionary with values correct UMLS code and keys start/end indexes
    # of the mention enity text
    count_gold = 0
    count_partial = 0

    entity_found = 0
    for entity in document.umls_entities:
        if (entity.start_idx, entity.stop_idx) in list_index[n_document]:
            if entity.cui in snomed_entities:
                # Selection of the one with a text embedding
                if hasattr(entity, 'bert_embedding'):
                    entity_found += 1
                    if (dict_mention_scispasy[(entity.start_idx, entity.stop_idx)][0]
                            == entity.cui):
                        count_gold += 1
                    if entity.cui in dict_mention_scispasy[(entity.start_idx,
                                                            entity.stop_idx)]:
                        count_partial += 1

```

```
    if  entity_found > 0:
        accuracy_gold_mentions = count_gold / entity_found
        accuracy_gold_list.append(accuracy_gold_mentions)
        n_entities_file.append(entity_found)
        accuracy_partial_mentions = count_partial / entity_found
        accuracy_partial_gold_list.append(accuracy_partial_mentions)

    n_document += 1
# Storing the values in a file
faccuracy_gold = open('Accuracy_gold_MM_testset_sci_lg.obj', 'wb')
pickle.dump(accuracy_gold_list, faccuracy_gold)
faccuracy_gold.close()

faccuracy_partial_gold = open('Accuracy_partial_gold_MM_testset_sci_lg.obj', 'wb')
pickle.dump(accuracy_partial_gold_list, faccuracy_partial_gold)
faccuracy_partial_gold.close()
```

A.3 Flair pre-trained model testing

```
#
# Test of the best pre-trained model: choosing between SapBERT or BERT
#
from flair.embeddings import TransformerWordEmbeddings
from flair.data import Sentence
from torch import nn

# Input embedding model
model = 'cambridgeltl/SapBERT-from-PubMedBERT-fulltext'  #bert-base-uncased
embedding = TransformerWordEmbeddings(model)

# Cystic fibrosis real examples
sentence1 = Sentence('Cystic fibrosis is a genetic condition. It is caused by a faulty gene'
                     ' that affects the movement of salt and water in and out of cells.')
sentence2 = Sentence('Cystic fibrosis is an inherited disorder that causes severe damage to '
                     'the lungs, digestive system and other organs in the body.')
sentence3 = Sentence('The primary cause of morbidity and death in people with cystic fibrosis'
                     ' is progressive lung disease.')
# Fake sentence
sentence4 = Sentence('The vaccine for the new COVID variant cystic fibrosis is going to be '
                     'released soon.')

sentenceCF = [sentence1, sentence2, sentence3, sentence4]
# Embed words in sentence
for sentence in sentenceCF:
    embedding.embed(sentence)

# Collecting cystic fibrosis embeddings and doing the mean between the two word embedding
cf_embeddings = [(sentence1.tokens[0].embedding + sentence1.tokens[1].embedding)/2,
                 (sentence2.tokens[0].embedding + sentence2.tokens[1].embedding)/2,
                 (sentence3.tokens[10].embedding + sentence3.tokens[11].embedding)/2,
```

```
                (sentence4.tokens[7].embedding + sentence4.tokens[8].embedding)/2]
# Check cosine similarity
print("Cosine similarity for CF, model: "+model)
for el1 in cf_embeddings:
    for el2 in cf_embeddings:
        cos = nn.CosineSimilarity(dim=0)
        print(cos(el1, el2))

# Penicillin real examples
sentence5 = Sentence('Penicillins are a group of antibiotics used to treat a wide range '
                     'of bacterial infections.')  # 0
sentence6 = Sentence('The penicillins are chemically described as 4-thia-1-azabicyclo '
                     '(3.2.0) heptanes.')  # 1
sentence7 = Sentence('Phenoxymethylpenicillin is a type of penicillin antibiotic. It is used'
                     ' to treat bacterial infections, including ear, chest, throat and '
                     'skin infections.')  # 5
# Fake sentence
sentence8 = Sentence('The penicillin is a tumor that can evolve in the brain.')  # 6

sentencepenicellin = [sentence5, sentence6, sentence7, sentence8]

# Embed words in sentence
for sentence in sentencepenicellin:
    embedding.embed(sentence)

# Collecting penicillin word embedding
penicillin_embeddings = [sentence5.tokens[0].embedding,
                         sentence6.tokens[1].embedding,
                         sentence7.tokens[5].embedding,
                         sentence8.tokens[1].embedding]
# Check cosine similarity
print("Cosine similarity for Penicillin, model:"+model)
for el1 in penicillin_embeddings:
    for el2 in penicillin_embeddings:
        cos = nn.CosineSimilarity(dim=0)
        print(cos(el1, el2))
```

A.4 Import SNOMED embedding

```
###
### Dictionary creation with keys UMLS code and value SNOMED CT graph embedding
###

import pickle
import os
import re
from ast import literal_eval
import numpy as np

# Create lists to store files with keys and values from SNOMED embedding directories
```

```
keys_file = []
for root, dirs, files in os.walk("/home/felicepaolocolliani/OneDrive/Tesi/Dataset/"
                                 "snomed_embeddings/Keys"):
    for file in files:
        keys_file.append(file)
    keys_file.sort()

embedding_file = []
for root, dirs, files in os.walk("/home/felicepaolocolliani/OneDrive/Tesi/Dataset/"
                                 "snomed_embeddings/Embedding"):
    for file in files:
        embedding_file.append(file)
    embedding_file.sort()

# Get the number of files in the directories
n_file = len(keys_file)

# Create an empty dictionary to store data
big_dict = {}

# Load pre-existing dictionary with only SNOMED entities presents in the embedding
f_snomed = open('SNOMEDtoUMLS_dict_clean_for_embed.obj', 'rb')
dict_snomed = pickle.load(f_snomed)
f_snomed.close()

# Loop over each file
for i in range(n_file):
    # Open and read files containing keys and embeddings
    f_keys = open('/home/felicepaolocolliani/OneDrive/Tesi/Dataset/snomed_embeddings/'
                  'Keys/' + keys_file[i], 'r')
    f_embedding = open('/home/felicepaolocolliani/OneDrive/Tesi/Dataset/'
                       'snomed_embeddings/Embedding/' + embedding_file[i], 'r')
    keys = f_keys.readlines()
    n_keys = len(keys)
    embedding = f_embedding.readlines()
    n_embedding = len(embedding)

    # Convert string representations of arrays into actual numpy arrays
    for k2 in range(n_embedding):
        embedding[k2] = np.array(literal_eval(embedding[k2]))

    # Check if the number of keys and embeddings match
    if n_embedding == n_keys:
        for k1 in range(n_keys):
            # Extract integers keys from the dict_snomed and map them to embedding values
            keys[k1] = int(re.sub(r'[^0-9]', '', keys[k1]))
            if keys[k1] in dict_snomed.keys():
                keys[k1] = dict_snomed[keys[k1]]
                big_dict[keys[k1][0]] = embedding[k1]

# Save the resulting dictionary to a file
f = open("Dict_embed_full.obj", 'wb')
```

```
pickle.dump(big_dict, f)
f.close()

```

A.5 Embedding PubTator file with SapBERT

```
#
# Using the pre-trained model SapBERT to add the embedding to the entities in documents
#
import pickle
import torch
from flair.embeddings import TransformerWordEmbeddings
from flair.data import Sentence

# Load a pubtator dataset (MM or BC5CDR) from file
dataset_pubtator = 'BC5CDR_Trainset.obj'
f = open(dataset_pubtator, 'rb')
data = pickle.load(f)
f.close()
# Check if the dataset name starts with 'BC5CDR'
if dataset_pubtator[0:6] == 'BC5CDR':
    # Load a dictionary to convert BC5CDR entities to UMLS code
    f = open('BC5CDRtoUMLS_dict.obj', 'rb')
    diz_bc5 = pickle.load(f)
    f.close()

# Load dictionary with SNOMED embeddings
f = open('Dict_embed_full.obj', 'rb')
diz = pickle.load(f)
f.close()

# Initialize a Transformer-based word embedding model with SapBERT
embedding = TransformerWordEmbeddings('cambridgeltl/SapBERT-from-PubMedBERT-fulltext')

# Get a list of SNOMED entities
snomed_ent = list(diz.keys())
start = 0

# Loop over each document in the dataset
for i in range(data.n_documents):
    print(i)
    # Preprocess the document by replacing '-' with spaces, to avoid errors
    data.document_list[i].raw_text = data.document_list[i].raw_text.replace('-', ' ')
    # Get the number of sentences in the document
    n_sent = len(data.document_list[i].sentences)

    sentences = [] # Initialize an empty list to store sentences
    # Create Sentence objects for each sentence in the document and embed them
    for sent_indices in data.document_list[i].sent_start_end_indices:
        sentences.append(Sentence(data.document_list[i].raw_text
```

```
                                [sent_indices[0]:sent_indices[1]]))
        embedding.embed(sentences[:])

    j = 0 # Initialize an index 'j' to keep track of sentences

    # Loop over each UMLS entity in the document
    for el in data.document_list[i].umls_entities:
        if dataset_pubtator[0:6] == 'BC5CDR':
            if el.cui in diz_bc5.keys():
                if any(x in snomed_ent for x in diz_bc5[el.cui]):
                    while el.stop_idx not in range(
                            data.document_list[i].sent_start_end_indices[j][1] + 1):
                        j += 1
                        start = 0
        else:
            while (el.stop_idx not in
                   range(data.document_list[i].sent_start_end_indices[j][1] + 1)):
                j += 1
                start = 0
                for k in range(start, len(sentences[j].tokens)):
                    if j == 0:
                        if sentences[j].tokens[k].start_position == el.start_idx:
                            mean = 1
                            while (sentences[j].tokens[k].end_position!=el.stop_idx and
                                   sentences[j].tokens[k].end_position+1 != el.stop_idx):
                                mean += 1
                                k = k + 1
                            if sentences[j].tokens[k].end_position + 1 == el.stop_idx:
                                print(el.mention_text, sentences[j].tokens[k].form)
                            if mean == 1:
                                el.bert_embedding =(
                                sentences[j].tokens[k].embedding[:, None].t())
                                start = k
                                break
                            elif mean > 1:
                                mean_embedding = (
                                sentences[j].tokens[k].embedding[:,None].t())
                                for z in range(1, mean):
                                    mean_embedding = torch.add(sentences[j].tokens[k-z].
                                                               embedding, mean_embedding)
                                el.bert_embedding = torch.div(mean_embedding, mean)
                                start = k
                                break
                    if j > 0:
                        if (sentences[j].tokens[k].start_position +
                            data.document_list[i].sent_start_end_indices[j][0]
                            == el.start_idx):
                            mean = 1
                            while (sentences[j].tokens[k].end_position +
                                   data.document_list[i].sent_start_end_indices[j][0]
                                   !=el.stop_idx and sentences[j].tokens[k].end_position
                                   +data.document_list[i].sent_start_end_indices[j][0]+1
```

```
                                    != el.stop_idx):
                                mean += 1
                                k = k + 1
                            if (sentences[j].tokens[k].end_position +
                                data.document_list[i].sent_start_end_indices[j][0]+1
                                == el.stop_idx):
                                print(el.mention_text, sentences[j].tokens[k].form)
                            if mean == 1:
                                el.bert_embedding = (
                                sentences[j].tokens[k].embedding[:, None].t())
                                start = k
                                break
                            elif mean > 1:
                                mean_embedding = (
                                sentences[j].tokens[k].embedding[:, None].t())
                                for z in range(1, mean):
                                    mean_embedding = torch.add(sentences[j]
                                                .tokens[k-z].embedding, mean_embedding)
                                el.bert_embedding = torch.div(mean_embedding, mean)
                                start = k
                                break

# Save the modified dataset with now SapBERT embeddings to a pickle file
f = open('BC5CDR_train_with_SAPBERT_emb.obj', 'wb')
pickle.dump(data, f)
f.close()
```

A.6 Graph Creation

```
import sys
from tqdm import tqdm

from db_connection import Neo4jConnection
from db_import import BaseImporter

SNOMED_RELS_FILE = 'ontologies/snomed/sct2_Relationship_Full_INT_20230331.txt'
SNOMED_NAMES_FILE = 'ontologies/snomed/sct2_Description_Full-en_INT_20230331.txt'

class SnomedRelationshipsImporter(BaseImporter):

    def __init__(self, argv):
        super().__init__(file=SNOMED_RELS_FILE, argv=argv)

    def set_constraints(self):
        queries = ["CREATE CONSTRAINT IF NOT EXISTS FOR (n:SnomedEntity) "
                   "REQUIRE n.id IS UNIQUE",
                   "CREATE INDEX snomedNodeName "
                   "IF NOT EXISTS FOR (n:SnomedEntity) ON (n.name)",
                   "CREATE INDEX snomedRelationId "
```

```
                    "IF NOT EXISTS FOR ()-[r:SNOMED_RELATION]-() ON (r.id)",
                    "CREATE INDEX snomedRelationType "
                    "IF NOT EXISTS FOR ()-[r:SNOMED_RELATION]-() ON (r.type)",
                    "CREATE INDEX snomedRelationUmls "
                    "IF NOT EXISTS FOR ()-[r:SNOMED_RELATION]-() ON (r.umls)"]

        for q in queries:
            self.connection.query(q, db=self.db)

    def import_SNOMED_RELS(self):
        query = """
        UNWIND $rows as item
        MERGE (e1:SnomedEntity {id: item.sourceId})
        MERGE (e2:SnomedEntity {id: item.destinationId})
        MERGE (e1)-[:SNOMED_RELATION {id: item.typeId}]->(e2)
        FOREACH(ignoreMe IN CASE WHEN item.typeId = '116680003' THEN [true] ELSE [] END|
            MERGE (e1)-[:SNOMED_IS_A]->(e2))
        """
        size = self.get_file_size()
        data = self.get_rows()
        self.load_in_batch(query, data, size, chunk_size=1000)

class SnomedNamesImporter(BaseImporter):

    def __init__(self, argv):
        super().__init__(file=SNOMED_NAMES_FILE, argv=argv)

    def importNodeNames(self):
        query = """
        UNWIND $rows as item
        MATCH (e:SnomedEntity {id: item.conceptId})
        FOREACH(x in CASE WHEN e.name IS NULL THEN [1] ELSE [] END |
            SET e.name = item.term
        )
        FOREACH(x in CASE WHEN item.term in e.aliases THEN [] ELSE [1] END |
            SET e.aliases = coalesce(e.aliases, []) + item.term
        )
        """

        size = self.get_file_size()
        data = self.get_rows()
        self.load_in_batch(query, data, size, chunk_size=1000)

    def importRelNames(self):
        query = """
        UNWIND $rows as item
        MATCH (:SnomedEntity)-[r:SNOMED_RELATION {id: item.conceptId}]->(:SnomedEntity)
        FOREACH(x in CASE WHEN r.type IS NULL THEN [1] ELSE [] END |
            SET r.type = toUpper(replace(item.term, ' ', '_'))
        )
        FOREACH(x in CASE WHEN toUpper(replace(item.term, ' ', '_')) in r.aliases
        THEN [] ELSE [1] END |
```

```
            SET r.aliases = coalesce(r.aliases, []) + toUpper(replace(item.term, ' ', '_')))
        """

        size = self.get_file_size()
        data = self.get_rows()
        self.load_in_batch(query, data, size, chunk_size=1000)


class SnomedLabelPropagator():
    def __init__(self, argv):
        self.db = argv[3]
        self.connection = Neo4jConnection(uri=argv[0], user=argv[1], pwd=argv[2])

    def perform_label_propagation(self):
        root_nodes = self.getSnomedRootNodes()
        self.annotateRootNodes(root_nodes)
        self.propagateLabels(root_nodes)

    def getSnomedRootNodes(self):
        query = """
                MATCH p=(n:SnomedEntity)<-[:SNOMED_IS_A]-(m:SnomedEntity)
                WHERE n.id= "138875005" // Root node
                RETURN DISTINCT id(m) as id, m.name as label
                """

        return self.connection.query(query, db=self.db)

    def annotateRootNodes(self, root_nodes):
        query= """
               MATCH (first_node) WHERE id(first_node)= $id
               WITH first_node, first_node.name as name
               SET first_node:ToBeProcessed
               RETURN first_node
               """
        for i in root_nodes:
            node_id = i['id']
            self.connection.query(query, parameters={'id': node_id}, db=self.db)

    def propagateLabels(self, root_nodes):
        query = """
                MATCH (first_node)
                WHERE id(first_node) = $id
                WITH first_node

                CALL apoc.path.expandConfig(first_node, {
                    relationshipFilter: '<SNOMED_IS_A',
                    minLevel: 1,
                    maxLevel: -1,
                    uniqueness: 'RELATIONSHIP_GLOBAL'
                }) yield path
                UNWIND nodes(path) as other_level
                WITH collect(DISTINCT other_level) as uniques
```

```
                UNWIND uniques as unique_other_level
                FOREACH(x in CASE WHEN $label in unique_other_level.type THEN [] ELSE [1]
                END | SET unique_other_level.type = coalesce(unique_other_level.type, [])
                + $label )
                RETURN unique_other_level
                """

        for i in tqdm(root_nodes):
            node_id = i['id']
            node_label = i['label']
            self.connection.query(query, parameters={'id': node_id, 'label': node_label},
                                  db=self.db)

if __name__ == '__main__':


    rel_importing = SnomedRelationshipsImporter(argv=sys.argv[1:])
    rel_importing.set_constraints()
    rel_importing.import_SNOMED_RELS()

    names_importing = SnomedNamesImporter(argv=sys.argv[1:])
    names_importing.importNodeNames()
    names_importing.importRelNames()

    type_propagator = SnomedLabelPropagator(argv=sys.argv[1:])
    type_propagator.perform_label_propagation()
```

A.7 Hard negative retrieval by Nearest Neighbours

```
#
# Creation of the hard negative dictionary using the KG as ideas
#
import random
from sklearn.neighbors import NearestNeighbors
import pickle

# Define the percentage of the dataset to use
dataset_percentage = 1

# Load dictionary of SNOMED embeddings
f = open('Dict_embed_node2vec.obj', 'rb')
diz = pickle.load(f)
f.close()

# Extract embeddings and corresponding SNOMED entities
embedding = list(diz.values())
snomed_ent = list(diz.keys())
n_entities = len(snomed_ent)

```

```python
# Initialize a Nearest Neighbors model with a specific number of neighbors
neigh = NearestNeighbors(n_neighbors=int(n_entities*0.10*0.25))
print(int(n_entities*0.10*0.25))  # Print the number of neighbors (e.g., 9379)
neigh.fit(embedding[0:int(n_entities*dataset_percentage)])
print(int(n_entities*dataset_percentage))  # Print the number of entities (e.g., 37518)

# Initialize an empty dictionary with SNOMED entities as keys and their neighbors as values
hard_negative = {}
x = 0

# Loop over each SNOMED entity
for i in range(n_entities):
    # Initialize an empty set for the current entity's hard negatives
    hard_negative[snomed_ent[i]] = set()

    # Find the nearest neighbors of the current entity and add them to the set
    for el in neigh.kneighbors([embedding[i]], 6, return_distance=False)[0][1:]:
        hard_negative[snomed_ent[i]].add(snomed_ent[el])
        # Adding random entity as negative
    while len(hard_negative[snomed_ent[i]]) < 10:
        random_ent = random.choice(snomed_ent)
        if random_ent != snomed_ent[i]:
            hard_negative[snomed_ent[i]].add(random_ent)
    # Convert the set to a list
    hard_negative[snomed_ent[i]] = list(hard_negative[snomed_ent[i]])

# Save the hard negatives dictionary to a file
f = open('hard_negative_dict_full_node2vec.obj', 'wb')  # Full and small
pickle.dump(hard_negative, f)
f.close()
```

A.8 Hard negative retrieval by full-text search

```python
import pickle
from full_text_search import full_text

# Specify Neo4j database connection details
argv = ['bolt://localhost:7687', 'neo4j', 'honey-judo-tahiti-moses-fossil-8127', 'snomed']

# Initialize a full-text search object with the provided arguments
full_search = full_text(argv=argv)

# Load dictionary of SNOMED embeddings
f = open('Dict_embed_node2vec.obj', 'rb')
diz = pickle.load(f)
f.close()

# Load dictionary with entities which need the negatives examples
f = open('Dict_name_MM_test.obj', 'rb')
diz_entity = pickle.load(f)
```

```
f.close()

# Extract SNOMED entities from the loaded dictionary
snomed_ent = set(diz.keys())

# Extract entity names and their UMLS codes from the loaded dictionary
entities = list(diz_entity.keys())
n_entities = len(entities)

# Initialize a dictionary with UMLS codes as keys and similar UMLS codes as values
hard_negative = {}

# Loop over each entity in the training set
for i in range(n_entities):
    print(i)
    hard_negative[entities[i]] = set()

    # Perform a full-text search using the entity's name and retrieve similar results
    res = full_search.search(diz_entity[entities[i]])

    # Process the search results
    if res is not None:
        for el in res[:min(50,len(res))]:
            if el[1] in snomed_ent:
                if el[1] != entities[i]:
                    hard_negative[entities[i]].add(el[1])
            if el[2] in snomed_ent:
                if el[2] != entities[i]:
                    hard_negative[entities[i]].add(el[2])
    hard_negative[entities[i]] = list(hard_negative[entities[i]])
    hard_negative[entities[i]] = (
                hard_negative)[entities[i]][:min(5,len(hard_negative[entities[i]]))]

# Save the hard negatives dictionary
f = open('hard_negative_dict_test_name_MM_node2vec.obj', 'wb')
pickle.dump(hard_negative, f)
f.close()
```

A.9 Candidate selection / full-text index search

```
#
# Candidate selection through neo4j
#
from db_import import BaseImporter2

class full_text(BaseImporter2):
    def __init__(self, argv):
        # Connection to the neo4j database
        super().__init__(argv=argv)
    def search(self, text):
```

```
        # Query to pursue the full-text index search
        q = 'CALL db.index.fulltext.queryNodes("rel","'+str(text)+('") '
            'YIELD node, score RETURN node.name, node.umls, node.code, score')
        a = self.connection.query(q, db=self.db)

        return a
```

A.10 Dataset building

```
#
# Creation of the dataset that will be used in model_train.py
#
import pickle
import torch

# Import of the text embedding in the MedMention/BC5CDR datasets
dataset_pubtator = 'MM_test_with_SAPBERT_emb.obj'
f = open(dataset_pubtator, 'rb') # full, train, test_dev
data = pickle.load(f)
f.close()

# Import of the dictionary with keys Snomed code and values graph embedding (positive)
f = open('Dict_embed_node2vec.obj','rb')
diz_emb = pickle.load(f)
f.close()

# Import of the dictionary with the negative examples
f = open('hard_negative_dict_test_name_MM_node2vec.obj', 'rb')
diz_neg = pickle.load(f)
f.close()

snomed_ent = set(diz_emb.keys()) # Selecting only the snomed entities
neg_keys = set(diz_neg.keys())
x = []  # It will contain the only positive example and the negatives examples
y = []  # It will contain the label [0,1] if the example is negative or positive

for i in range(len(data)): # Cycle for document to process
    print(i)
    for el in data[i].umls_entities:  # Cycle over all the UMLS entity in the file
        if el.cui in snomed_ent and el.cui in neg_keys: # Selecting only the snomed entities
            if hasattr(el, 'bert_embedding'): # Selection of the one with a text embedding
                # appending positive example and label 1
                x.append(torch.cat((torch.from_numpy(diz_emb[el.cui])[:, None].t(),
                                    el.bert_embedding), 1))
                y.append(1)
                # appending negative examples and label 0
                for neg in diz_neg[el.cui][:5]:
                    x.append(torch.cat((torch.from_numpy(diz_emb[neg])[:, None].t(),
                                        el.bert_embedding), 1))
                    y.append(0)
```

```

# Saving the data tensors and the labels
f_x = open('x_y/MM/x_tensors_full_test_name_MM_node2vec_name.obj','wb')
pickle.dump(x,f_x) # small, train, test_dev, _NO_RANDOM
f_x.close()

f_y = open('x_y/MM/y_label_full_test_name_MM_node2vec_name.obj','wb')
pickle.dump(y, f_y) # small, train, test_dev, _NO_RANDOM
f_y.close()
```

```
import pickle
import torch
from torch.utils.data import TensorDataset

f_x_train = open('x_y/BC5CDR/x_tensors_full_train_name_BC5CDR_node2vec_name.obj', 'rb')
x_train = pickle.load(f_x_train)
f_x_train.close()

f_y_train = open('x_y/BC5CDR/y_label_full_train_name_BC5CDR_node2vec_name.obj', 'rb')
y_train = pickle.load(f_y_train)
f_y_train.close()

x_train = torch.cat(x_train)
y_train = torch.tensor(y_train, dtype=torch.float64).reshape(-1, 1)

trainset = (TensorDataset(x_train, y_train))

f = open('testset_trainset/trainset_name_BC5CDR_node2vec_name.obj','wb')
pickle.dump(trainset,f)
f.close()

f_x_test = open('x_y/BC5CDR/x_tensors_full_test_name_BC5CDR_node2vec_name.obj', 'rb')
x_test = pickle.load(f_x_test)
f_x_test.close()

f_y_test = open('x_y/BC5CDR/y_label_full_test_name_BC5CDR_node2vec_name.obj', 'rb')
y_test = pickle.load(f_y_test)
f_y_test.close()

x_test = torch.cat(x_test)
y_test = torch.tensor(y_test, dtype=torch.float64).reshape(-1, 1)

testset =  TensorDataset(x_test, y_test)

f = open('testset_trainset/testset_name_BC5CDR_node2vec_name.obj','wb')
pickle.dump(testset,f)
f.close()
```

A.11 Model train

```
#
# Defining of a neural network and training it, in the end checking the f-1 score
#
import torch.nn as nn
import torch.utils.data
import torch.optim as optim
from torcheval.metrics.functional import binary_f1_score
import pickle
from nn import Net, SiameseNetwork, UpgradedNet

# Create an instance of the neural network
net = SiameseNetwork() #UpgradedNet() #SiameseNetwork()  #Net()

# Set batch size for data loading
batch_size = 256

# Load training and test datasets
trainset = pickle.load(open('testset_trainset/trainset_name_BC5CDR_node2vec_name.obj', 'rb'))
trainloader = torch.utils.data.DataLoader(trainset, batch_size=batch_size, shuffle=True,
                                          num_workers=2, drop_last=True)
testset = pickle.load(open('testset_trainset/testset_name_BC5CDR_node2vec_name.obj', 'rb'))

# Define loss function
criterion = nn.BCELoss()

# Initialize the optimizer with the initial learning rate
lr = [0.005, 0.001, 0.0005, 0.0001]
k = -1
optimizer = optim.Adam(net.parameters(), lr=lr[0])
massimo = 0

# Training loop
for epoch in range(40):
    if epoch % 10 == 0:
       k += 1
       optimizer = optim.Adam(net.parameters(), lr=lr[k])
    running_loss = 0.0

    for i, data in enumerate(trainloader, 0):
        # Get inputs and labels
        inputs, labels = data
        inputs1, inputs2 = torch.split(inputs, [128,768], dim=1)

        # Zero the parameter gradients
        optimizer.zero_grad()

        # Forward pass and backward pass
        # outputs = net(inputs)
        outputs = net(inputs1, inputs2)
        loss = criterion(outputs, labels)
```

```
        loss.backward()
        optimizer.step()

        # Print statistics
        running_loss += loss.item()

        if i % 2000 == 1999:
            print(f'[{epoch + 1}, {i + 1:5d}] loss: {running_loss / 2000:.3f}')
            running_loss = 0.0

    # Calculate and print F1 scores on the test set
    a, b = torch.split(testset.tensors[0],  [128,768], dim=1)
    outputs = net(a,b)
    # outputs = net(testset.tensors[0])
    f1_score_running_test_05 = binary_f1_score(outputs.t()[0], testset.tensors[1].t()[0]
                                               .type(torch.int64), threshold=0.5)
    print(f'[{epoch + 1} f_1 score test set 0.5: {f1_score_running_test_05:.3f}')
    a, b = torch.split(trainset.tensors[0], [128, 768], dim=1)
    outputs = net(a, b)
    f1_score_running_train_05 = binary_f1_score(outputs.t()[0], trainset.tensors[1].t()[0]
                                              .type(torch.int64), threshold=0.5)
    print(f'[{epoch + 1} f_1 score train set: {f1_score_running_train_05:.3f}')

    # Save the model with the highest F1 score on the test set
    if f1_score_running_test_05 > massimo:
        massimo = f1_score_running_test_05
        print('massimo f1_score: ', f1_score_running_test_05)
        torch.save(net.state_dict(), 'models/model_name_siamese_BC5CDR_node2vec_name.pth')
print('Finished Training')
```

A.12 Neural Networks

```
#
# Neural network used in model_train
#
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.utils.data


# Define neural network class 'Net' for binary classification
class Net(nn.Module):
    def __init__(self):
        super().__init__()
        # Define layers for the neural network
        self.fc1 = nn.Linear(1536, 400, dtype=torch.float64)
        self.fc2 = nn.Linear(400, 150, dtype=torch.float64)
        self.fc3 = nn.Linear(150, 64, dtype=torch.float64)
        self.fc4 = nn.Linear(64, 1, dtype=torch.float64)
```

```
19
20      def forward(self, x):
21          x = F.relu(self.fc1(x))
22          x = F.relu(self.fc2(x))
23          x = F.relu(self.fc3(x))
24          x = F.sigmoid(self.fc4(x))
25          return x
26
27  # Define an upgraded neural network class 'UpgradedNet' with BatchNormalization, dropout
28  class UpgradedNet(nn.Module):
29      def __init__(self):
30          super(UpgradedNet, self).__init__()
31          # Define layers, BatchNormalization, and dropout
32          self.fc1 = nn.Linear(1536, 400, dtype=torch.float64)
33          self.bn1 = nn.BatchNorm1d(400, dtype=torch.float64)
34          self.fc2 = nn.Linear(400, 150, dtype=torch.float64)
35          self.bn2 = nn.BatchNorm1d(150, dtype=torch.float64)
36          self.fc3 = nn.Linear(150, 64, dtype=torch.float64)
37          self.bn3 = nn.BatchNorm1d(64, dtype=torch.float64)
38          self.fc4 = nn.Linear(64, 1, dtype=torch.float64)
39          self.dropout_less = nn.Dropout(0.2)
40          self.dropout = nn.Dropout(0.5)
41
42      def forward(self, x):
43          # Apply dropout and BatchNormalization in the forward pass
44          x = self.dropout_less(x)
45          x = F.relu(self.bn1(self.fc1(x)))
46          x = self.dropout(x)
47          x = F.relu(self.bn2(self.fc2(x)))
48          x = self.dropout(x)
49          x = F.relu(self.bn3(self.fc3(x)))
50          x = self.fc4(x)
51          x = torch.sigmoid(x)
52          return x
53
54  # Define a Siamese neural network class 'SiameseNetwork' for binary classification
55  class SiameseNetwork(nn.Module):
56      def __init__(self):
57          super(SiameseNetwork, self).__init__()
58          # Define layers, BatchNormalization, dropout, and sigmoid activation
59          self.brothers_short = nn.Sequential(
60              nn.Linear(128, 300, dtype=torch.float64),
61              nn.ReLU(),
62              nn.BatchNorm1d(300, dtype=torch.float64),
63              nn.Linear(300, 250, dtype=torch.float64),
64              nn.ReLU(),
65              nn.BatchNorm1d(250, dtype=torch.float64),
66              nn.Linear(250, 200, dtype=torch.float64),
67              nn.ReLU(),
68              nn.BatchNorm1d(200, dtype=torch.float64)
69          )
70          self.brothers = nn.Sequential(
71              nn.Linear(768, 300, dtype=torch.float64),
```

```
            nn.ReLU(),
            nn.BatchNorm1d(300, dtype=torch.float64),
            nn.Linear(300, 250, dtype=torch.float64),
            nn.ReLU(),
            nn.BatchNorm1d(250, dtype=torch.float64),
            nn.Linear(250, 200, dtype=torch.float64),
            nn.ReLU(),
            nn.BatchNorm1d(200, dtype=torch.float64)
        )
        self.fc1 = nn.Linear(400, 128, dtype=torch.float64)
        self.fc2 = nn.Linear(128, 64, dtype=torch.float64)
        self.fc3 = nn.Linear(64, 1, dtype=torch.float64)
        self.sigmoid = nn.Sigmoid()
        self.dropout_less = nn.Dropout(0.2)
        self.dropout = nn.Dropout(p=0.5)

    def forward_once(self, output):
        output = self.dropout_less(output)
        output = F.relu(self.fc1(output))
        output = self.dropout(output)
        output = F.relu(self.fc2(output))
        output = self.sigmoid(self.fc3(output))
        return output

    def forward(self, input1, input2):
        # Forward common pass for Siamese network
        output1 = self.brothers_short(input1)
        output2 = self.brothers(input2)
        output = torch.cat((output1, output2), 1)
        output = self.forward_once(output)
        return output
```

A.13 Model test

```
import pickle
from full_text_search import full_text
import torch
from nn import SiameseNetwork, Net, UpgradedNet
import spacy
from scispacy.linking import EntityLinker
from scispacy.abbreviation import AbbreviationDetector

spacy.require_cpu()
# Add the abbreviation pipe to the spacy pipeline.
nlp = spacy.load("en_core_sci_lg")
nlp.add_pipe("abbreviation_detector")

# Specify Neo4j database connection details
argv = ['bolt://localhost:7687', 'neo4j', 'honey-judo-tahiti-moses-fossil-8127', 'snomed']

```

```
# Initialize a full-text search object with the provided arguments
full_search = full_text(argv=argv)

# Loading the neural network already trained
net = SiameseNetwork()
net.load_state_dict(torch.load('models/model_name_siamese_BC5CDR_node2vec_7732.pth'))
net.eval()

f = open('Dict_embed_node2vec.obj', 'rb')
diz_embed = pickle.load(f)
f.close()
snomed_entities = set(diz_embed.keys())

documents_test = pickle.load(open('BC5CDR_test_with_SAPBERT_emb.obj', 'rb'))

f = open('list_index_lg_BC5CDR.obj','rb')
list_index = pickle.load(f)
f.close()

# Lists to store gold and partial accuracy
accuracy_gold_list = []
accuracy_partial_gold_list = []
n_entities_file = []
n_document = 0
# Looping on each pubtator document
for doc in documents_test:
    print(n_document)
    doc_abbreviation = nlp(doc.raw_text)
    long_dict = {}
    for short in doc_abbreviation._.abbreviations:
        long_dict[short.lemma_] = short._.long_form.lemma_
    abbreviation = list(long_dict.keys())
    count_gold = 0
    count_partial = 0
    entity_found = 0
    for entity in doc.umls_entities:
        x = []
        if (entity.start_idx, entity.stop_idx) in list_index[n_document]:
            if entity.cui in snomed_entities:
                # Selection of the one with a text embedding
                if hasattr(entity, 'bert_embedding'):
                    if entity.mention_text.lower() in abbreviation:
                        entity.mention_text = long_dict[entity.mention_text.lower()]
                    result = full_search.search(entity.mention_text)
                    entity_found += 1 # Count how many entity are counted for the accuracy
                    candidates = set()
                    if result is not None and len(result)>0:
                        for el in result[:min(50, len(result))]:
                            if el[1] in snomed_entities:
                                candidates.add(el[1])
                            if el[2] in snomed_entities:
                                candidates.add(el[2])
```

```
                        candidates = list(candidates)
                        if len(candidates) > 0:
                            for candidate in candidates:
                                x.append(torch.cat((torch.from_numpy(diz_embed[candidate])
                                                    [:, None].t(),entity.bert_embedding), 1))
                            x = torch.cat(x)
                            inputs1, inputs2 = torch.split(x, [128,768], dim=1)
                            outputs = net(inputs1, inputs2)
                            #outputs = net(x)

                            index_max = torch.topk(outputs,min(5,outputs.shape[0]),
                                                   dim=0).indices[:,0].tolist()
                            if candidates[index_max[0]] == entity.cui:
                                count_gold += 1
                            for index in index_max:
                                if candidates[index] == entity.cui:
                                    count_partial += 1
    if entity_found > 0:
        n_entities_file.append(entity_found)
        accuracy_partial_gold_list.append(count_partial / entity_found)
        accuracy_gold_list.append(count_gold/entity_found)
    n_document += 1

faccuracy_gold = open('Accuracy_gold_BC5CDR_testset_node2vec_index.obj', 'wb')
pickle.dump(accuracy_gold_list, faccuracy_gold)
faccuracy_gold.close()

faccuracy_partial_gold = open('Accuracy_partial_gold_BC5CDR_testset_node2vec_index.obj','wb')
pickle.dump(accuracy_partial_gold_list, faccuracy_partial_gold)
faccuracy_partial_gold.close()
```

Bibliography

Alan Akbik, Tanja Bergmann, Duncan Blythe, Kashif Rasul, Stefan Schweter, and Roland Vollgraf. FLAIR: An easy-to-use framework for state-of-the-art NLP. In *NAACL 2019, 2019 Annual Conference of the North American Chapter of the Association for Computational Linguistics (Demonstrations)*, pages 54–59, 2019.

Rajarshi Bhowmik, Karl Stratos, and Gerard de Melo. Fast and effective biomedical entity linking using a dual encoder, 2021.

Antoine Bordes, Nicolas Usunier, Alberto Garcia-Duran, Jason Weston, and Oksana Yakhnenko. Translating embeddings for modeling multi-relational data. 26, 2013. URL `https://proceedings.neurips.cc/paper_files/paper/2013/file/1cecc7a77928ca8133fa24680a88d2f9-Paper.pdf`.

Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners, 2020.

Yuanfei Dai, Shiping Wang, Neal N. Xiong, and Wenzhong Guo. A survey on knowledge graph embedding: Approaches, applications and benchmarks. *Electronics*, 9(5), 2020. ISSN 2079-9292. URL `https://www.mdpi.com/2079-9292/9/5/750`.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding, 2019.

Aditya Grover and Jure Leskovec. node2vec: Scalable feature learning for networks, 2016.

Geoffrey E. Hinton, Nitish Srivastava, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Improving neural networks by preventing co-adaptation of feature detectors. *CoRR*, abs/1207.0580, 2012. URL `http://arxiv.org/abs/1207.0580`.

Matthew Honnibal, Ines Montani, Sofie Van Landeghem, and Adriane Boyd. spaCy: Industrial-strength Natural Language Processing in Python, 2020.

Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift, 2015.

Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2017.

Jiao Li, Yueping Sun, Robin J. Johnson, Daniela Sciaky, Chih-Hsuan Wei, Robert Leaman, Allan Peter Davis, Carolyn J. Mattingly, Thomas C. Wiegers, and Zhiyong Lu. Biocreative v cdr task corpus: a resource for chemical disease relation extraction. *Database: The Journal of Biological Databases and Curation*, 2016, 2016. URL `https://api.semanticscholar.org/CorpusID:88817`.

Yankai Lin, Zhiyuan Liu, Maosong Sun, Yang Liu, and Xuan Zhu. Learning entity and relation embeddings for knowledge graph completion. page 2181–2187, 2015.

Fangyu Liu, Ehsan Shareghi, Zaiqiao Meng, Marco Basaldella, and Nigel Collier. Self-alignment pretraining for biomedical entity representations, June 2021.

Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space, 2013.

Sunil Mohan and Donghui Li. Medmentions: A large biomedical corpus annotated with UMLS concepts. *CoRR*, abs/1902.09476, 2019. URL `http://arxiv.org/abs/1902.09476`.

Mark Neumann, Daniel King, Iz Beltagy, and Waleed Ammar. ScispaCy: Fast and Robust Models for Biomedical Natural Language Processing. In *Proceedings of the 18th BioNLP Workshop and Shared Task*, pages 319–327, Florence, Italy, August 2019. Association for Computational Linguistics. doi: 10.18653/v1/W19-5034. URL `https://www.aclweb.org/anthology/W19-5034`.

Maximilian Nickel, Volker Tresp, and Hans-Peter Kriegel. A three-way model for collective learning on multi-relational data. In *Proceedings of the 28th International Conference on International Conference on Machine Learning*, ICML'11, page 809–816, Madison, WI, USA, 2011. Omnipress. ISBN 9781450306195.

Alberto Parravicini, Rhicheek Patra, Davide B. Bartolini, and Marco D. Santambrogio. Fast and accurate entity linking via graph embedding, 2019. URL `https://doi.org/10.1145/3327964.3328499`.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019. URL `http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf`.

Nikhil Pattisapu, Sangameshwar Patil, Girish Palshikar, and Varma Vasudeva. Medical Concept Embeddings for SNOMED-CT (Jan 2019 version), May 2020. URL `https://doi.org/10.5281/zenodo.3842143`.

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

Jeffrey Pennington, Richard Socher, and Christopher Manning. GloVe: Global vectors for word representation. pages 1532–1543, October 2014. doi: 10.3115/v1/D14-1162. URL `https://aclanthology.org/D14-1162`.

Ariel Schwartz and Marti Hearst. A simple algorithm for identifying abbreviation definitions in biomedical text. *Pacific Symposium on Biocomputing. Pacific Symposium on Biocomputing*, 4: 451–62, 02 2003. doi: 10.1142/9789812776303_0042.

Xinying Song, Alex Salcianu, Yang Song, Dave Dopson, and Denny Zhou. Fast wordpiece tokenization, 2021.

Théo Trouillon, Johannes Welbl, Sebastian Riedel, Éric Gaussier, and Guillaume Bouchard. Complex embeddings for simple link prediction, 2016.

Maya Varma, Laurel Orr, Sen Wu, Megan Leszczynski, Xiao Ling, and Christopher Ré. Cross-domain data integration for named entity disambiguation in biomedical text, 2021.

Haifeng Wang, Jiwei Li, Hua Wu, Eduard Hovy, and Yu Sun. Pre-trained language models and their applications. *Engineering*, 25:51–65, 2023. ISSN 2095-8099. doi: https://doi.org/10.1016/j.eng.2022.04.024. URL https://www.sciencedirect.com/science/article/pii/S2095809922006324.

Zhen Wang, Jianwen Zhang, Jianlin Feng, and Zheng Chen. Knowledge graph embedding by translating on hyperplanes. *Proceedings of the AAAI Conference on Artificial Intelligence*, 28 (1), Jun. 2014. doi: 10.1609/aaai.v28i1.8870. URL https://ojs.aaai.org/index.php/AAAI/article/view/8870.

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