ALMA MATER STUDIORUM – UNIVERSITY OF BOLOGNA
CAMPUS OF CESENA

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
Computer science and engineering

SUBGRAPH RETRIEVAL FOR BIOMEDICAL OPEN-DOMAIN QUESTION ANSWERING: PREDICTING RELATIONAL PATHS WITH LANGUAGE MODELS

Elaborato in
Web Semantico

Relatore
Prof. Antonella Carbonaro

Presentata da
Mattia Achilli

Co-relatori
Prof. Gianluca Moro
Dr. Giacomo Frisoni

Quarta Sessione di Laurea
Anno Accademico 2021 – 2022

PAROLE CHIAVE

Open-domain Question Answering

Subgraph Retrieval

Dense Information Retrieval

Knowledge Graph Embedding

Language Models

The people who are crazy enough to think they can change the world are the ones who do.
- Steve Jobs

Sommario

Negli ultimi anni, l'open book question answering (QA) è diventato un campo di ricerca di grande interesse, che mira a rispondere a domande complesse utilizzando conoscenze esterne. La maggior parte delle soluzioni esistenti si basa sull'utilizzo di grafi per rappresentare la conoscenza semantica, strutturata e non ambigua, che può essere utilizzata per rispondere alle domande.

Tuttavia, la costruzione di grafi su larga scala è una sfida complessa, che richiede l'utilizzo di tecniche di reasoning avanzate. Inoltre, molte delle soluzioni esistenti si basano su approcci discreti o simbolici, che sono inefficienti e portano a molto rumore.

In questo lavoro di tesi, è stata progettata e implementata una soluzione di subgraph retrieval approssimata, che lavora all'interno di spazi densi ed è plug and play. Inoltre, la soluzione proposta si focalizza sul dominio biomedico, dove la conoscenza è molto vasta e spesso altamente specializzata.

In particolare, la soluzione proposta si basa sul predire, partendo dalle entità della domanda, le relazioni da visitare in modo autoregressivo grazie al fine-tuning di un language model biomedico encoder only e cross document allo stato dell'arte, cosi da sfruttare al massimo la conoscenza e il significato del testo e dei concetti.

La soluzione proposta è stata testata e confrontata con diverse tecniche di KG embedding come TransE e DistMult, e query SPARQL attraverso GraphDB. I risultati empirici mostrano l'efficacia e l'efficienza della soluzione proposta all'interno della tesi, comparata a quelle disponibili in letteratura.

Abstract

In recent years, open-book question answering (QA) has become a great, interesting research field that aims to answer complex questions using outside knowledge. Most existing solutions are based on graphs representing semantic, structured knowledge and are unambiguous, which can be used to answer questions.

However, constructing large-scale graphs is a complex challenge requiring advanced reasoning techniques. Also, many existing solutions are based on discrete or symbolic approaches, which are inefficient and lead to much noise.

In this thesis work, a solution has been designed and implemented for approximate subgraph retrieval, which works within dense spaces and is plug-and-play. Furthermore, the proposed solution focuses on the biomedical domain, where knowledge is very broad and often highly specialized.

In particular, the proposed solution is based on predicting, starting from entity of the question, the relations to visit in an autoregressive way thanks to the fine-tuning of a biomedical language model encoder only and cross-document state of the art, to make the most of the knowledge and meaning of the text and concepts.

The proposed solution has been tested and compared with different techniques of KG embedding, such as TransE and DistMult, and SPARQL queries via GraphDB.

The empirical results show the effectiveness and efficiency of the proposed solution within the thesis compared to those available in the literature.

Introduzione

Il question answering è un task complesso che si occupa della costruzione di sistemi che rispondono automaticamente alle domande poste dagli esseri umani in un linguaggio naturale che richiede un reasoning complesso sia sul contesto testuale della domanda, nonché conoscenze non dichiarate e pertinenti sul mondo (cioè, conoscenza del dominio di interesse). Solitamente i sistemi di risposta alle domande estraggono risposte da una raccolta non strutturata di documenti in linguaggio naturale.

Si possono avere due principali tipi di question answering:

- **Closed-domain**: si occupa di domande relative a un dominio specifico (come la medicina) e può sfruttare la conoscenza specifica del dominio spesso formalizzata in ontologie.
- **Open-domain**: si occupa di domande relative a qualsiasi contesto e può fare affidamento solo su ontologie generali e conoscenza generale. D'altra parte, questi sistemi di solito hanno a disposizione molti più dati da cui estrarre la risposta. Un esempio di domanda a dominio aperto è "Per cosa Albert Einstein ha vinto il premio Nobel?" mentre nessun articolo su questo argomento viene fornito al sistema. A loro volta il question answering open domain si divide in:
  - **Closed-book**: il sistema ha una conoscenza interna appresa durante l'addestramento per rispondere, allo stesso modo di uno studente che si prepara per un esame senza avere la possibilità di visionare informazioni esterne durante l'esame.
  - **Open-book**: il sistema ha accesso alle informazioni esterne per rispondere, come uno studente che durante l'esame può visionare libri o documenti per rispondere.

L'informazione può essere sotto forma testuale o a grafo in cui le entità fondamentali sono collegate tra loro tramite relazioni. Nell'elaborato di tesi il contenuto informativo tratta di dati biomedici relativi al dataset **MedQA-USMLE** contenente domande e risposte multiple di esame di dominio biomedico presenti negli Stati Uniti.

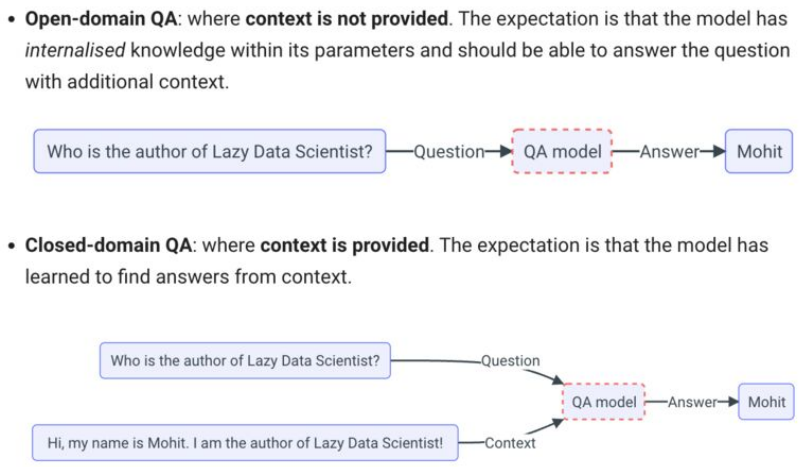


Figure 1: Differenze tra open-domain e closed-domain.

Organizzazione della tesi

La tesi è organizzata come segue:

- **Capitolo 1** - Viene presentato un quadro generale sui concetti di natural language processing, language models, knowledge graphs, ed embeddings.
- **Capitolo 2** - Viene chiarito il contesto e si riassumono lavori pre-esistenti.
- **Capitolo 3** - Viene descritta il metodo proposto, soffermandosi sugli aspetti che lo compongono.
- **Capitolo 4** - Viene descritto l'ambiente e la soluzione più nello specifico e tecnicamente.
- **Capitolo 5** - Vengono descritti i risultati ottenuti.

Introduction

Question answering is a complex task that deals with the construction of systems that automatically answer questions posed by humans in natural language that require complex reasoning both on the textual context of the question, as well as unstated and relevant knowledge about the world (i.e., knowledge of the domain of interest). Question answering systems typically extract answers from an unstructured collection of natural language documents.

There are two main types of question answering:

- **Closed-domain**: deals with domain-specific questions (such as medicine) and can leverage domain-specific knowledge often formalized in ontologies.
- **Open-domain**: deals with questions related to any context and can only rely on general ontologies and general knowledge. On the other hand, these systems usually have much more data available to extract the answer from. An example of an open domain question is "What did Albert Einstein win the Nobel Prize for?" while no articles on this topic are provided to the system. In turn, the open domain question answering is divided into:
  - **Closed-book**: the system has internal knowledge learned during training to answer. In the same way, a student prepares for an exam without having the opportunity to view external information during the exam.
  - **Open-book**: the system has access to external information to answer, such as a student who can view books or documents during the exam to answer.

The information can be in textual or graph form in which the fundamental entities are connected through relationships. In the thesis work, the information content deals with biomedical data relating to the **MedQA-USMLE** dataset containing questions and multiple answers of biomedical domain exams in the United States.

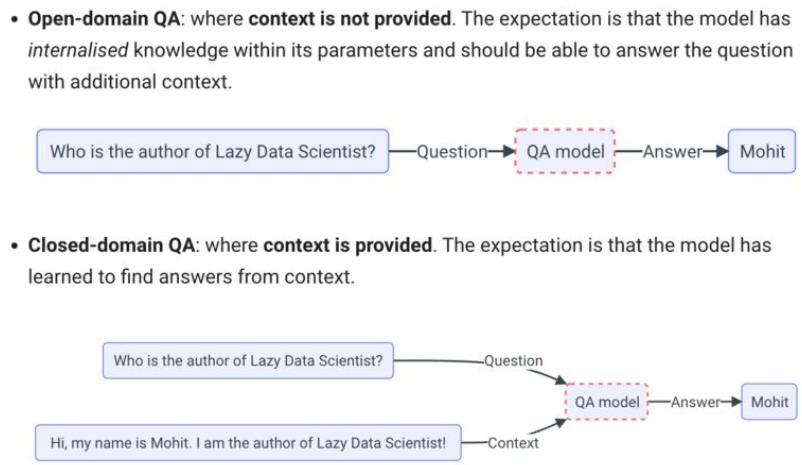


Figure 2: Difference between open-domain and closed-domain.

Thesis Organization

The thesis is organized as follows:

- **Chapter 1** - A general framework is presented on natural language processing, language models, knowledge graphs, and embeddings.
- **Chapter 2** - The context is clarified, and pre-existing work is summarized.
- **Chapter 3** - The method is described, focusing on the aspects that make it up.
- **Chapter 4** - The environment and the solution are described more specifically and technically.
- **Chapter 5** - The results obtained are described.

Contents

List of Figures

Chapter 1

Theoretical Framework

This chapter will provide some basic preliminary studies about Natural Language Processing, Transformers, and Knowledge Graphs.

1.1 Natural Language Processing

Natural Language Processing (NLP) is a branch of computer science that bridges the gap between human and computer language [3]. The main issues faced in the NLP field concern the aspects required by machines to understand natural language text. Among these, we find:

- Language modeling that focuses on quantifying the associations between the set of words.
- The morphological processing, which consists of the segmentation of the significant components of words and the identification of the parts of speech.
- Syntactic processing, the process of constructing sentences according to the constraints dictated by the language.
- Semantic processing that deals with extracting the meaning of words, phrases, or text components.

1.1.1 Transformers

The concept of Transformers in NLP was proposed in the article **"Attention Is All You Need"** [4].

It is a model that focuses on tracing dependencies between inputs and outputs. It is very efficient in translating texts; in fact, it can reach the state

of the art after a few hours of training. The model consists mainly of two components, an **encoder** that maps a sequence of input symbols

$$(x_1, ..., x_n)$$

into a sequence of continuous representations $z = (z_1...z_n)$ and a **decoder** that generates a sequence of output $(y1...yn)$ one element at a time. The generation occurs token after token, where the prediction of the token t depends only on those generated up to $t - 1$. As shown in Figure 1.1, both the encoder and the decoder are formed by a stack of N layers, all identical, each layer has two other sublayers. The former refers to the multi-head Attention mechanism, while the latter is simply a Feed-Forward network. The decoder has an extra sublayer of multi-head attention that operates on the encoder output.

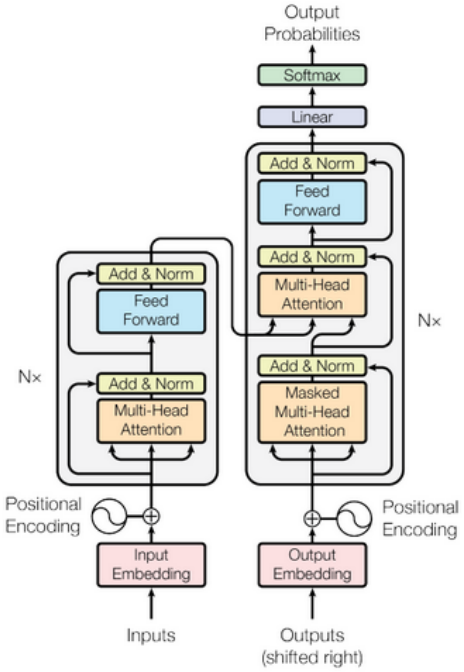


Figure 1.1: The Transformer - model architecture.
From [5]

An Attention function is defined like this:

$$Attention(Q, K, V) = softmax(\frac{QK^T}{\sqrt{d_K}})$$

where Q is the matrix that contains the queries, i.e. the vectors representing the words in the sequence K are the keys and V the values. The function outputs a weighted sum of the values, where the weight assigned to each value

is calculated through Q and K. The multi-head lets you pay attention to information from different representations in different positions. Multi-head Attention is used in three distinct ways in the model:

1. *Encoder-decoder Attention*: queries come from the decoder's previous layer; instead, memory keys come from the decoder's output. This allows each decoder position to occupy each position in the input sequence.

2. The encoder contains layers of *self-Attention*: all keys, values, and queries come from the output of the previous layer of the encoder. Each position can occupy all positions in the previous level.

3. Similarly, the self-attention layers in the decoder allow any position in the decoder to occupy all positions up to the current one.

The transformer has swiftly overtaken other neural models as the industry standard for natural language processing, exceeding them in comprehension and text production. The architecture is especially well-suited for pretraining on massive datasets, which significantly improves performance on tasks like *classification*, *machine translation*, and *text summarization*. As a result of these advancements, using these models at scale poses new obstacles, necessitating the need for systems to train, assess, and scale the model on the domain. The transformer skeleton is used to create extensions and highly complex models [6]. Still, as technology advances, new obstacles arise when applying them on a broad scale, necessitating the refinement of the task's scope.

1.2 Encoder only language models

Encoder language models use only the encoder of a Transformer model. At each stage, the attention layers can access all the words in the initial sentence. These models are often characterized as having "bi-directional" attention and are often called auto-encoding models.

The pre-training of these models usually revolves around somehow corrupting a given sentence (for instance, by masking random words in it) and tasking the model with finding or reconstructing the initial sentence. Encoder models are best suited for tasks requiring an understanding of the full sentence, such as sentence classification, named entity recognition (and more general word classification), and in our case, extractive question answering.

Two of the most representative models of this family are BERT (Bidirectional Encoder Representations from Transformers) and RoBERTa (Robustly Optimized BERT Approach).

1.2.1 BERT

BERT [1] is a representation model developed by Google AI. It uses pre-training and fine-tuning phases to create state-of-the-art models for various tasks. These tasks include question-answering systems, sentiment analysis, and language inference. The main goal is to help the information retrieval systems better understand the context around your searches. Figure 1.2 the sentiment classification task, which represents the sentiments of the user reviews, whether positive or negative.

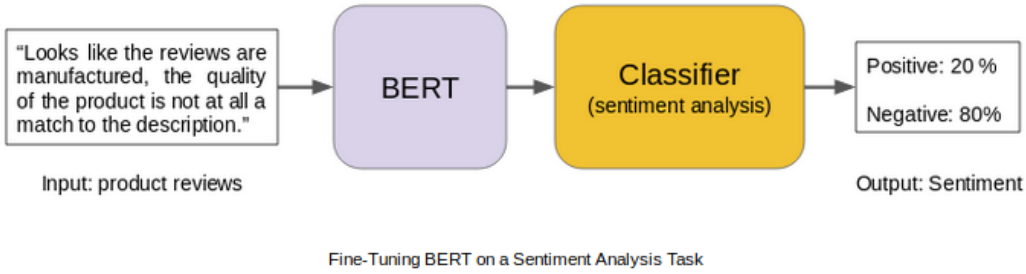


Figure 1.2: Classical Bert Sentiment Analysis Task [1].

The architecture of BERT is a multi-layer bidirectional Transformer encoder, and the self-attention layer performs self-attention in both directions, forward and backward. The variants of Google BERT are shown below, with the number of transformer layers and the number of parameters used for tuning the model.

- BERT Base:
  - Number of Transformers layers: 12
  - Total Parameters: 110M
- BERT Large:
  - Number of Transformers layers: 24
  - Total Parameters: 340M

Apart from output layers, the same architectures are used in pre-training and fine-tuning. In most of the classification problems, we use one part of the model, like the encoder, for classification purposes. The same pre-trained model parameters are used to initialize models for different downstream tasks.

During fine-tuning, all parameters are fine-tuned. The special token `[CLS]` is added in front of every input example, and the `[SEP]` is a special separator token (e.g. separating questions/answers). Pre-trained Language Models (PLMs) are large neural network models that are first pre-trained on a large number of documents and used in a wide variety of NLP tasks. For the fine-tuning of the model, we need to use the pre-trained or unfreeze model, add a new layer on top of the existing one, perform tasks related to classification, etc.

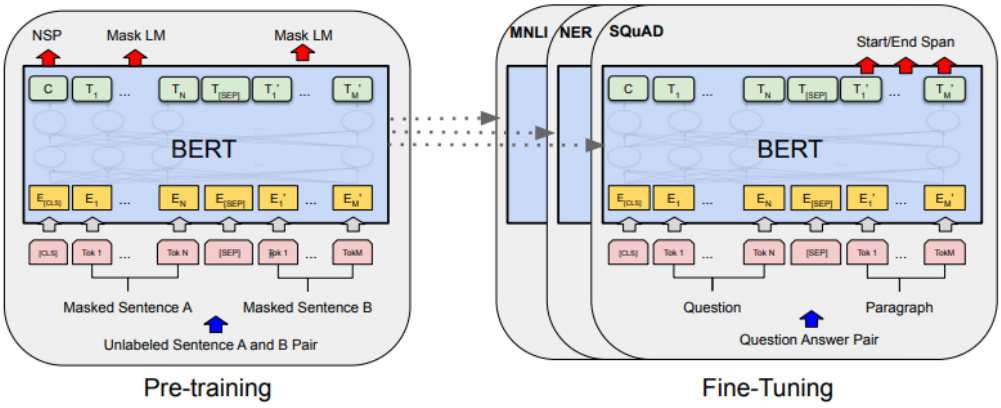


Figure 1.3: Google BERT Architecture

1.2.2 RoBERTa

RoBERTa [7] is a variant of the BERT model, which was developed by researchers at Facebook AI. Like BERT, RoBERTa is a transformer-based language model that uses self-attention to process input sequences and generate contextualized representations of words in a sentence. One key difference between RoBERTa and BERT is that RoBERTa was trained on a much larger dataset and using a more effective training procedure. In particular, RoBERTa was trained on a dataset of 160GB of text, which is more than 10 times larger than the dataset used to train BERT. RoBERTa uses a dynamic masking technique during training that helps the model learn more robust and generalizable representations of words.

RoBERTa has been shown to outperform BERT and other state-of-the-art models on a variety of natural language processing tasks, including language translation, text classification, and question answering. It has also been used as a base model for many other successful NLP models and has become a popular choice for research and industry applications.

RoBERTa has almost similar architecture as compared to BERT, but—to improve results—the authors made some simple design changes in BERT architecture and training procedure.

These changes are:

- **Removing the Next Sentence Prediction (NSP) objective**: in the next sentence prediction, the model is trained to predict whether the observed document segments come from the same or distinct documents via an auxiliary Next Sentence Prediction (NSP) loss, removing the NSP loss matches or slightly improves downstream task performance.

- **Training with bigger batch sizes & longer sequences**: originally, BERT is trained for 1M steps with a batch size of 256 sequences. RoBERTa is trained with 125 steps of 2K sequences and 31K steps with 8k sequences of batch size. This has two advantages: the large batches improve perplexity on masked language modeling objectives and end-task accuracy. Large batches are also easier to parallelize via distributed parallel training.
- **Dynamically changing the masking pattern**: in BERT architecture, the masking is performed once during data preprocessing, resulting in a single static mask. To avoid using the single static mask, training data is duplicated and masked 10 times, each time with a different mask strategy over 40 epochs, thus having 4 epochs with the same mask. This strategy is compared with dynamic masking in which different masking is generated every time we pass data into the model.

RoBERTa has significantly improved dataset performance.

1.2.3 Notable examples of language models on biomedical domain

BERT can be adapted for our thesis's biomedical domain. BioBERT (Bidirectional Encoder Representations from Transformers for Biomedical Text Mining) [2], which is a domain-specific language representation model pre-trained on large-scale biomedical corpora. With almost the same architecture across tasks, BioBERT largely outperforms BERT in a variety of biomedical text mining tasks.

The pre-trained model is used to fine-tune on various biomedical text mining tasks like named entity recognition, question & answer, and relation extraction.

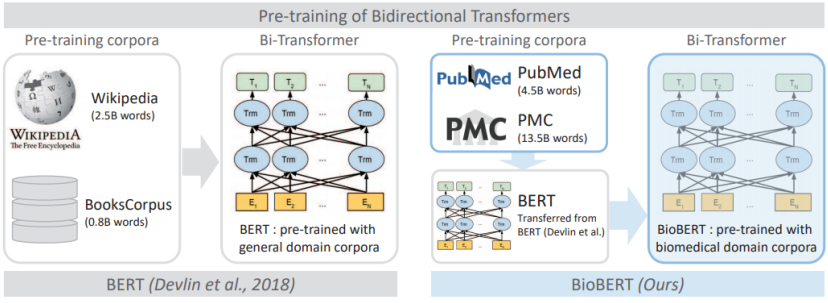


Figure 1.4: BioBERT is pre-trained on PubMed Abstracts 4.5 billion words & PMC Full-text articles 13.5 billion words [2].

The interesting part was that the pre-training was not just on biomedical corpora but rather took different combinations of general and biomedical

corpora since their research objective was to figure out the performance of BERT with different combinations of corpora and the amount of data needed to pre-train the model.

1.3 Knowledge Graphs

1.3.1 Definitions

The heart of the knowledge graph is a knowledge model: a collection of interlinked descriptions of concepts, entities, relationships and events. Knowledge graphs put data in context via linking and semantic metadata and this way provide a framework for data integration, unification, analytics and sharing.

Knowledge graphs combine characteristics of several data management paradigms:

- **Database**: because the data can be explored via structured queries;
- **Graph**: because they can be analyzed as any other network data structure;
- **Knowledge base**: because they bear formal semantics, which can be used to interpret the data and infer new facts.

Knowledge graphs can be represented in **RDF**, provide the best framework for data integration, unification, linking and reuse, because they combine:

- **Expressivity**: the standards in the Semantic Web stack – RDF and OWL – allow for a fluent representation of various types of data and content: data schema, taxonomies and vocabularies, all sorts of metadata, reference and master data. The RDF extension makes it easy to model provenance and other structured metadata.
- **Performance**: all the specifications have been thought out, and proven in practice, to allow for efficient management of graphs of billions of facts and properties.
- **Interoperability**: there is a range of specifications for data serialization, access (SPARQL Protocol for end-points), management (SPARQL Graph Store) and federation. The use of globally unique identifiers facilitates data integration and publishing.
- **Standardization**: all the above is standardized through the W3C community process, to make sure that the requirements of different actors are satisfied – all the way from logicians to enterprise data management professionals and system operations teams.

Figure 1.5 shows an example of a simple Knowledge Graph.

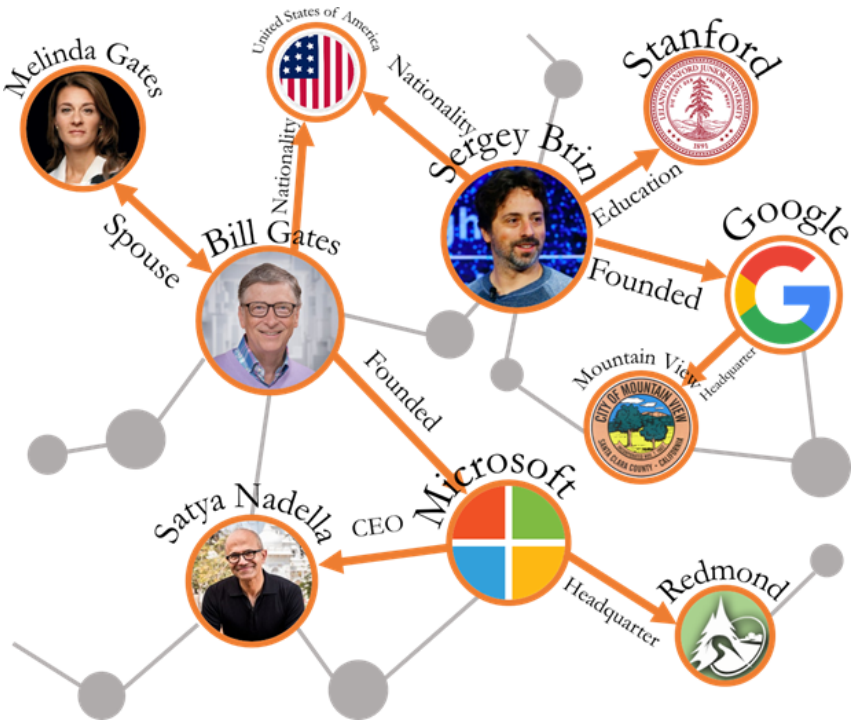


Figure 1.5: Simple example of Knowledge Graph

In a Knowledge base, the information is usually stored as a triple: subject, predicate, and object.

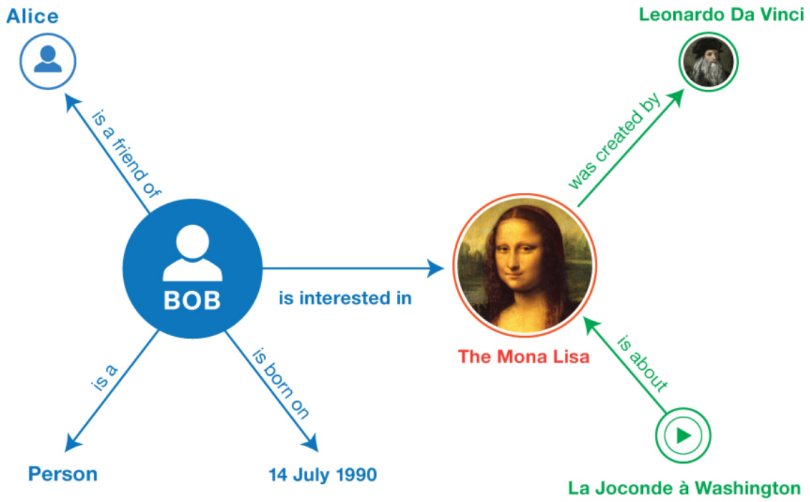


Figure 1.6: Example of Knowledge Base as triplets

For example, data is available in articles or general documents such as Wikipedia, Commonsense, and USMLE directories. Through the use of NLP techniques such as information extraction systems that can extract meaningful information and encode them in triplets.

1.3.2 Examples of big Knowledge Graphs

There are several examples of large knowledge graphs including:

- **DBPedia**: this project leverages the structure inherent in the infoboxes of Wikipedia to create an enormous dataset and an ontology that has encyclopedic coverage of entities such as people, places, films, books, organizations, species, diseases, etc. This dataset is at the heart of the Open Linked Data movement. It has been invaluable for organizations to bootstrap their internal knowledge graphs with millions of crowdsourced entities.
- **Geonames**: under a creative commons, users of Geonames dataset have access to 25 million geographical entities and features.
- **Wordnet**: one of the most well-known lexical databases for the English language, providing definitions and synonyms. Often used to enhance the performance of NLP and search applications.

And so many others Knowledge Graphs exists.
Later we will see in detail how to build and use a Knowledge graph using the GraphDB triplestore.

1.3.3 How can Knowledge Graphs help text analysis?

Modern text analysis technology makes considerable use of knowledge graphs:

- Big graphs provide background knowledge, human-like concepts and entity awareness, to enable a more accurate interpretation of the text;
- The results of the analysis are semantic tags (annotations) that link references in the text to specific concepts in the graph. These tags represent structured metadata that enables better search and further analytics;
- Facts extracted from the text can be added to enrich the knowledge graph, which makes it is much more valuable for analysis, visualization and reporting.

1.4 Embedding

In this section, we will understand embedding and how algorithms deal with textual information. In Natural Language Processing, the main goal is to extract useful information and features from the given input. The input is in the human language, consisting of words, sentences, and documents in different languages like Italian, English, Hindi, etc.

1.4.1 Basic concepts about the embedding

The basic task of information extraction such as:

- Text summarization: extractive or abstractive text summarization.
- Sentiment Analysis [Positive, Negative].
- Translating from one language to another: Neural Machine Translation.
- Chatbots.

For processing the input information we need to transform in such a representation that is useful for algorithms because machine and deep learning algorithms only process the numeric input. So we need to find a mechanism that transforms the input into numerical representations.

Bag of words (BOW) was one of the first feature extraction methods that processes the information in order to count the frequencies of each word that appear in the text corpora or sentences. By using pre-processing steps like tokenization of sentences and removing stop words and other punctuation symbols from the given text for building the vocabulary of the known words. These counts can be in binary counts which means the text is present in the documents or not, leading to problems:

- If most of the elements are zero then the BOW will be a sparse matrix and the sparse representations are difficult for model computation because of the high number of input vector weights.
- Another issue is that BOW does not preserve the semantics of words and the order of word appearance in the text is not considered.

1.4.2 Semantics embeddings

Word embedding was the solution for these limitations of the high dimension of the vectors because embedding transforms the large sparse vectors into the lower dimensional space. It also preserves the semantic relationship

between words. Considering the order of words means the vectors of similar words are placed near each other in the space as shown in the figure below.

For example: **"king is to queen as man is to woman"**.

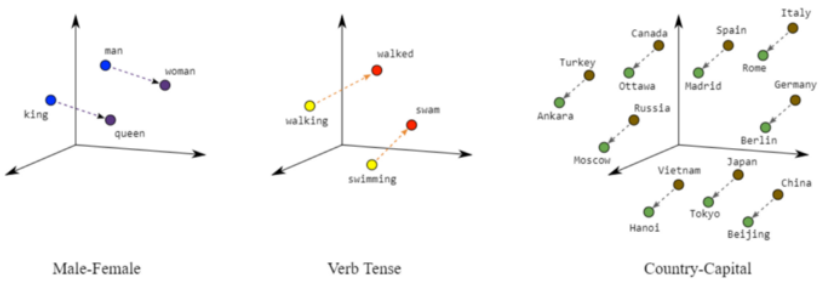


Figure 1.7: Similar words placed near to each other in the space.

Word2Vec [8] A Google-invented algorithm used for training word embeddings. It relies on the distributional hypothesis, meaning that the words that are semantically similar can be placed near each other in the space as shown. Word2Vec method helps to map semantically similar words to geometrically close embedding vectors. For the distributional hypothesis, it uses the continuous bag of words or the skip grams. A continuous bag of words uses the continuous distribution of the context; it also considers the order of the word as well the previous history of the words like collaborative filtering.

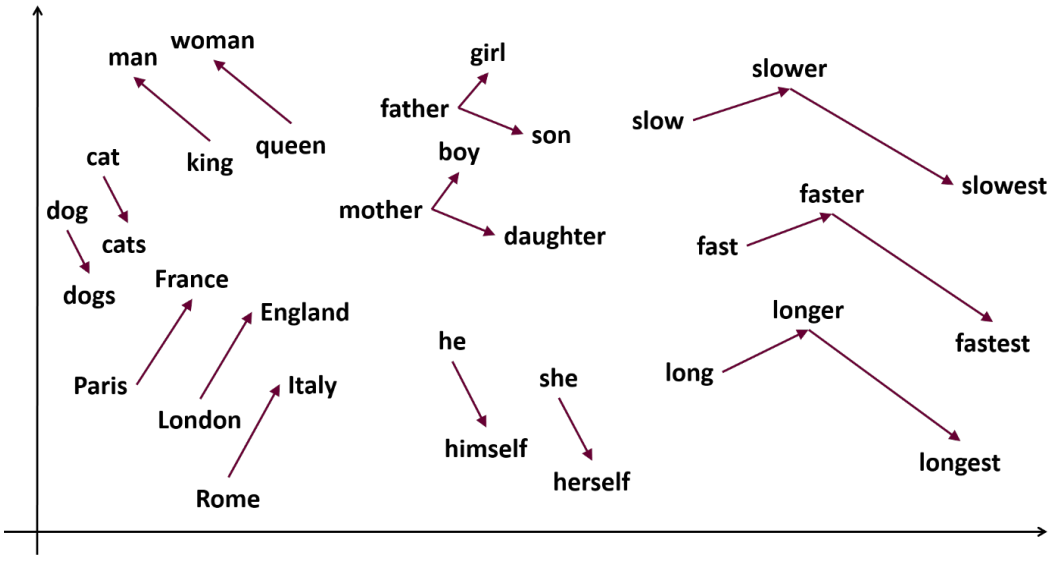


Figure 1.8: Example of Word2Vec

1.4.3 Graph embedding

Graph embedding is a technique that allows mapping the elements of a graph (nodes or arcs) in a low-dimensional representation space, called embedding space. Graph embedding is used for various graph machine-learning tasks, such as node classification, arc prediction, and visualization.

Translational Embeddings. Translational methods for graph embedding are a category of techniques that focus on representing the relationships between nodes in the graph. These translational methods create an embedding based on properties of the graph structure, such as the distance between nodes or the presence of certain relationships. The main concept behind these methods is to translate the properties of a node into another node, which is considered to be nearby. The best-known translational embedding technique is **TransE** (Translation-based Embedding) [9]. TransE uses a linear model to translate the representation of a node in its relation to another node. In this model, each node is represented by a vector, and a translation vector represents each relation. The dot product of these vectors allows us to calculate the similarity between the relationships, which is an indicator of their affinity. This method is very efficient in semantic graph embedding and link prediction. In general, translational methods offer a simple and intuitive representation of the nodes and relationships in the graph, but they can be limited in representing more complex graphs and more complex relationships. However, they are very popular due to their simplicity and ability to work with large graphs.

Multiplicative Embeddings Multiplicative methods are another family of graph embedding techniques that are based on the concept of matrix multiplication. These methods focus on the product of matrices to represent the relationships between nodes in the graph.

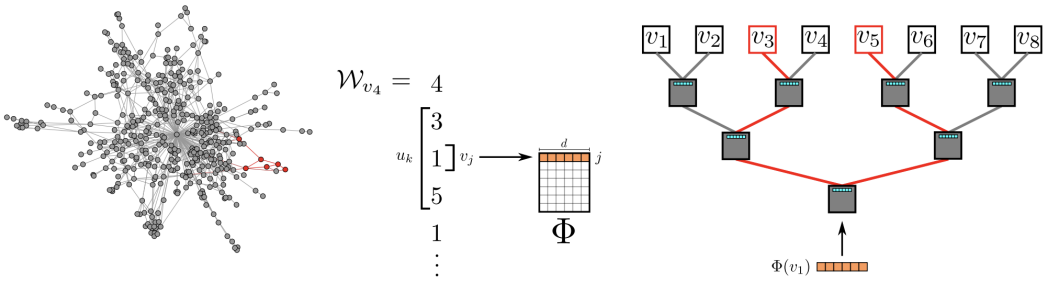
One of the best-known multiplicative methods is **HolE** (Holographic Embeddings) [10], which uses a projection matrix that encodes the relationship between two nodes. The projection matrix is used to multiply the node embedding vectors, generating a new vector representing the relationship between the nodes.

In general, multiplicative methods are more complex than translational methods but offer greater flexibility in representing the relationships between nodes. However, they also require more data to train the models and can be slower during the inference phase than translational methods.

Random paths neural methods Random path neural methods are a type of graph embedding that uses information about the topological properties of

the graph to generate node embeddings. These methods base their idea on the assumption that the semantic properties of a node are related to nearby nodes in the graph. These methods use random walks on the graph as input to neural models and train the models to predict the next node in the random walk.

An example of this type of method is DeepWalk [11], first proposed in 2014. Its approach extracts random paths from the graph and uses them as input to a natural language model. The technique is based on the assumption that the semantic property of a node is related to its neighbours in the graph, which is why these random paths are used as input to the natural language model. The natural language model then trains a model to predict the next node in the random walk to learn a dense representation of the node. This dense representation is used as an embedding for the node and can be used in various machine learning task taxonomies, such as classification, clustering, and link prediction. Overall, DeepWalk has been proven to be an effective method for learning graphics embedding and continues to be one of the foundational techniques in graphics embedding research.



(a) Random walk generation. (b) Representation mapping. (c) Hierarchical Softmax.

Figure 1.9: Deep Walk

Graph Neural Networks Graph Neural Networks (GNN) are a class of machine learning models that focus on the relationships between elements within a graph. In the context of embedding, GNNs can produce graph node embeddings that capture the node's properties and its location within the graph.

This is done by defining an aggregate function that combines information from a node's neighbours to produce a new node representation. This new representation is then used as input to a neural network which produces the desired embedding. GNNs have often been used in graph matching, link prediction and node classification problems.

Several variants of GNN, such as Graph Convolutional Networks (GCN) and Graph Attention Networks (GAT), have different ways of defining the

aggregation function and using information about a node's neighbours. In general, GNNs are a very powerful class of graph embedding methods and are used in many graph analysis problems.

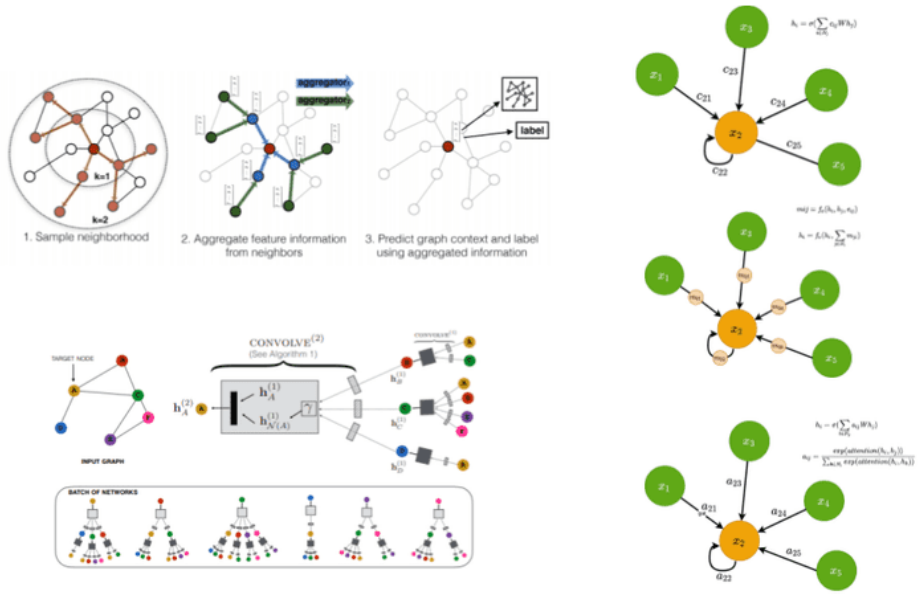


Figure 1.10: Workflow of Graph Neural Network

Chapter 2

Background

This chapter clarifies the context and presents the related works; also, the last project presented is the one used and extended for the solution.

2.1 Overview

Open-domain question answering has attracted much attention, as the logically organized entities and their relations are beneficial for inferring the answer. Semantic parsing-based (SP-based) and embedding-based methods are mainstream methods. The former heavily relies on the expensive annotation of the intermediate logic form, such as SPARQL.

Instead of parsing the questions, the later ones directly represent and rank entities based on their relevance to input questions. Among them are the models which first retrieve a question-relevant subgraph and then perform reasoning on it, reducing the reasoning space and showing superiority compared with reasoning on the whole KB.

Subgraph retrieval is crucial to the overall QA performance, as a small subgraph is highly likely to exclude the answer. Still, a large one might introduce noises that affect the QA performance.

This thesis proposes a subgraph retrieval enhanced model for open-domain question answering, which devises a trainable subgraph retriever (SR) decoupled from the subsequent reasoner. SR is devised as an efficient dual-encoder that can automatically expand paths to induce the subgraph and stop the expansion. Such separable retrieval and reasoning ensure the reasoning only on the final entire instead of the intermediate partial subgraphs.

The data in thesis deals with biomedical data relating to the MedQA-USMLE dataset, which contains multiple biomedical domain exam questions and answers featured in the United States.

2.2 Related Works

The following sections will illustrate the existing solutions to the problem, describing their approaches, results and limitations.

2.2.1 Reasoning with Language Models and Knowledge Graphs for Question Answering (QA-GNN)

QA-GNN [12] is one of the first projects that address the above challenges through two key innovations:

- **Relevance scoring**: since the KG subgraph consists of all few-hop neighbors of the topic entities, some entity nodes are more relevant than others concerning the given QA context. They propose KG node relevance scoring: they score each entity on the KG subgraph by concatenating the entity with the QA context and calculating the likelihood using a pre-trained LM. This presents a general framework to weigh information on the KG;

- **Joint reasoning**: they design a joint graph representation of the QA context and KG, where they explicitly view the QA context as an additional node (QA context node) and connect it to the topic entities in the KG subgraph. This joint graph, which they termed the working graph, unifies the two modalities into one graph. They then augment the feature of each node with the relevance score and design a new attention-based GNN module for reasoning.

They evaluate the model on QA benchmarks in the commonsense (CommonsenseQA, OpenBookQA) and biomedical (MedQA-USMLE) domains.

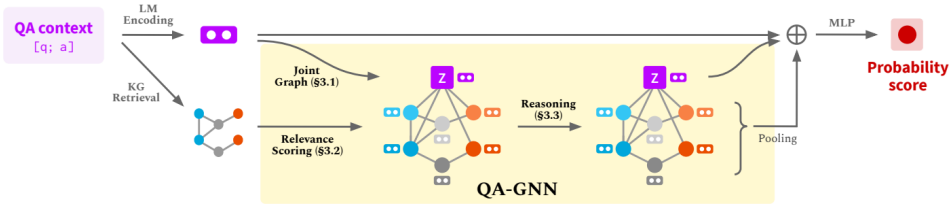


Figure 2.1: Overview of their approach. Given a QA context (z), they connect it with the retrieved KG to form a joint graph, compute the relevance of each KG node conditioned, and perform reasoning on the working graph

Approach As shown in the figure 2.1, given a question and an answer choice **a**, they concatenate them to get the QA context **[q; a]**. To reason over a given QA context using knowledge from both the LM and the KG, QA-GNN works as follows:

- They use the LM to obtain a representation for the QA context, and retrieve the subgraph from the KG.

- Then they introduce a QA context node **z** that represents the QA context, and connect z to the topic entities, so that they have a joint graph over the two sources of knowledge, which they term the working graph. To adaptively capture the relationship between the QA context node and each of the other nodes, they calculate a relevance score for each pair using the LM and use this score as an additional feature for each node.

- Finally, they make the final prediction using the LM representation, QA context node representation and a pooled working graph representation.

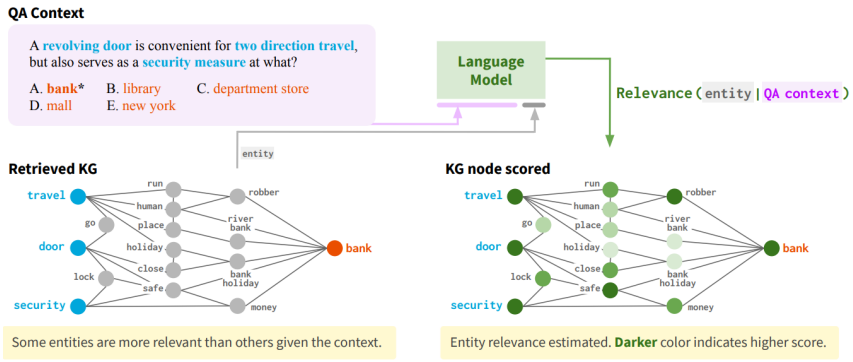


Figure 2.2: Relevance scoring of the retrieved KG: they use a pre-trained LM to calculate the relevance of each KG entity node conditioned on the QA context

Results They compared with existing LM+KG methods, which share the same high-level framework as their but use different modules to reason on the KG in place of QA-GNN. The key differences between QA-GNN and these are that they do not perform relevance scoring or joint updates with the QA.

The image below shows the result of QA-GNN on MedQA-USMLE. QA-GNN outperformed fine-tuned LMs (e.g., SapBERT [13]).

Methods	Test
BERT-base (Devlin et al., 2019)	34.3
BioBERT-base (Lee et al., 2020)	34.1
RoBERTa-large (Liu et al., 2019)	35.0
BioBERT-large (Lee et al., 2020)	36.7
SapBERT (Liu et al., 2020a)	37.2
SapBERT + QA-GNN (**Ours**)	**38.0**

Figure 2.3: Test accuracy on MedQA-USMLE

Limitations QA-GNN presents different limitations:

- First, in structured Knowledge Graphs, the size of the retrieved graph is very challenging because if the retrieved graph is too small so there are chances of missing the answer because of the small size, and if it is retrieved large, that can cause to introduce the noise that may effect on the reasoning phase. Earlier methods use different strategies like the retrieved the top-K entities from the knowledge base triple store (head, relation, tail). This approach can also retrieve some off-topic entities that do not belong to the question.

- Second, the issue about the weak internal intermediate node or entities means making or establishing a connection with the intermediate node. Suppose we look into space when we want to explore the space and start from topic entities and explore other paths until we find the desired answer in the space; if there is no answer in the space, which direction do we follow to go through in between them answer and the question. So, the fact that there is no correct answer for the intermediate process is called the state of Weak Internal Media Supervision, which is also faced in the existing studies. This challenge can be seen in the dataset, only the question and its correct answer are given, and there is no answer to the intermediate nodes of which multi-hop process to find the correct answer. (Intermediate node answers missing). Therefore, the extracted KG subgraph ignores some intermediate concept (entity) nodes and edges. In such cases, the subgraph does not contain a complete chain of reasoning.

- Third, an incomplete subgraph is retrieved without complete information about the given entities, relations, neighbour nodes, etc.

These are significant challenges mentioned above. However, these challenges are the issues that the various methodologies currently being researched are trying to solve.

2.2.2 Graph Reasoning Enhanced Language Models For Question Answering (GreaseLM)

GreaseLM [14] is the next new model that fuses encoded representations from pre-trained LMs and graphs neural networks over multiple layers of modality interaction operations. Information from both modalities propagates to the other, allowing language context representations to be grounded by structured world knowledge and linguistic nuances (e.g., negation, hedging) in the context to inform the graph representations of knowledge. They use the same dataset of QA-GNN: CommonsenseQA, OpenbookQA and medical question answering MedQA-USMLE.

GreaseLM consists of two stacked components:

- a set of unimodal LM layers which learn an initial representation of the input tokens
- a set of upper cross-modal GREASELM layers which learn to represent the language sequence jointly and linked knowledge graph, allowing textual representations formed from the underlying LM layers and a graph representation of the KG to mix.

They denote the number of LM layers as N, and the number of GREASELM layers as M. The total number of layers in their model is N + M.

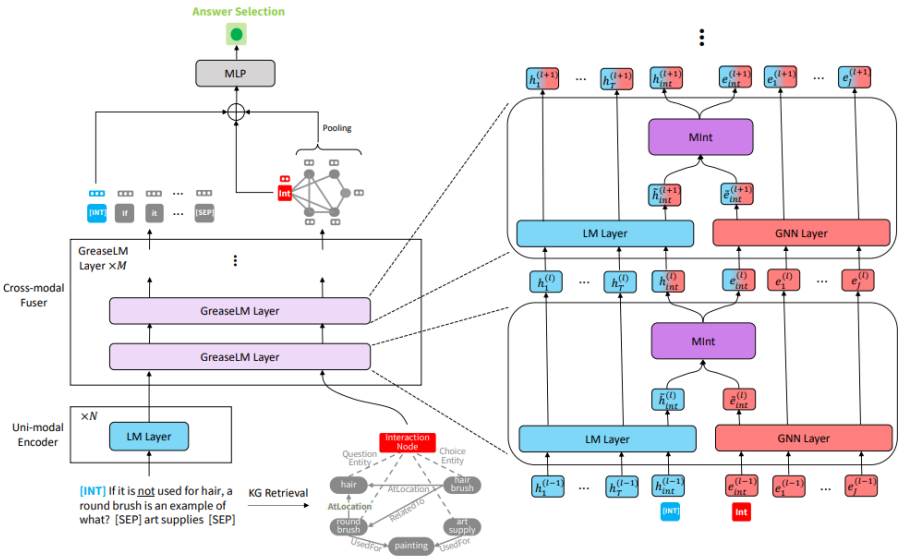


Figure 2.4: Architecture of GreaseLM

Approach First, GreaseLM takes input representations of both modalities, expressive large language models, and structured KGs. The KG retrieval retrieves the subgraph from the KG (similarly mentioned in the QA-GNN). For the second source of the input, pass the input question through the multiple language model encoder layers called a uni-model encoder, take its representations, and use it in the other phase.

Then the cross-modal fuser takes both sources of the inputs and mixes the information from KG and language representations for bidirectional integrations. GreaseLM has three components; the first transformer LM encoder block continuously encodes the language context, and the second GNN layer reasons over the KG entities and relations. The third one, called the mint modality operator, takes the uni-model representation of interaction tokens and nodes and exchanges the information through them.

In each layer in the network, the representation of each modality is updated, and the representations of the interaction token and node are pulled, concatenated, and passed through a modality interaction **(Mint)** unit to mix their representations. In the following layers, the mixed information from the interaction elements combines with their respective modalities, allowing knowledge from the KG to affect the representations of individual tokens and context from language to affect fine-grained entity knowledge representations in the GNN.

GreaseLM focuses on the reasoning phase with multi-layer attention model architecture that performs better reasoning compared to the existing QA-GNN in some cases, like dealing with the negation, hedging, or flip of the entities, etc. It is considered the extended version of QA-GNN, especially in the reasoning phases. GreaseLM also uses the same subgraph retrieval phase used in the QA-GNN; at the end.

For example, the model correctly predicts that the answer to the given question is "airplane," while on the other hand, QA-GNN predicts the incorrect one as "motor vehicle." For both models, GreaseLM and QAGNN, they perform the Best First Search (BFS) on the retrieved KG subgraph to trace high attention weights from the interaction node.

Results As in QA-GNN, they compared GreaseLM with existing LM+KG methods, including QA-GNN.

The image below shows the result of GreaseLM on MedQA-USMLE.

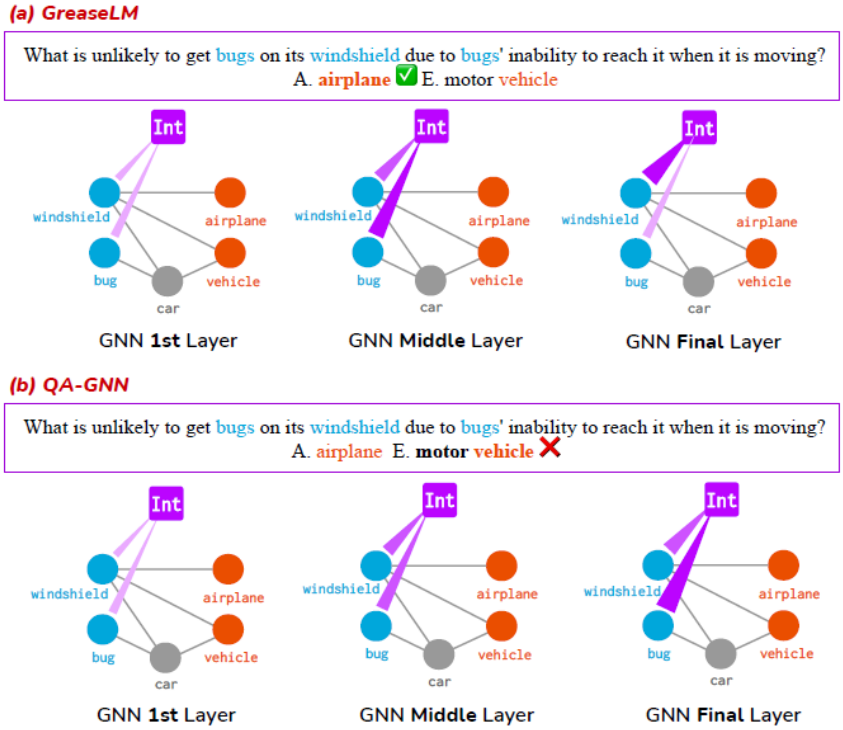


Figure 2.5: Qualitative analysis of GreaseLM's graph attention weight changes across multiple messages passing layers compared with QA-GNN. GreaseLM demonstrates attention change patterns that more closely resemble the expected change in focus on the "bug" entity.

Methods	**Acc.** (%)
Baselines (Jin et al., 2021)	
CHANCE	25.0
PMI	31.1
IR-ES	35.5
IR-CUSTOM	36.1
CLINICALBERT-BASE	32.4
BIOROBERTA-BASE	36.1
BIOBERT-BASE	34.1
BIOBERT-LARGE	36.7
Baselines (Our implementation)	
SapBERT-Base (w/o KG)	37.2
QA-GNN	38.0
GREASELM (**Ours**)	**38.5**

Figure 2.6: Test accuracy of GreaseLM on MedQA-USMLE

So a not insignificant amount of work is introduced for a very small 0.5 improvement over QA-GNN.

Limitations Existing methods typically combine two modalities (LM and KG) in a shallow and non-interactive mode, encoding both sources of knowledge separately and combining them at the output phase for a prediction or using one to augment the input of the other. The existing method fails to deal with multiple constraints, such as negation and hedges conditions that require practical reasoning over both language context and KG. The subgraph retrievals are similar to the work of QA-GNN, which is inefficient concerning the size and the retrieved off-topic entities.

2.2.3 Deep Bidirectional Language-Knowledge Graph Pre-training (DRAGON)

A self-supervised method called DRAGON (Deep Bidirectional Language-Knowledge Graph Pre-training) [15] has greatly improved QA-GNN and GreaseLM; DRAGON pre-trains a deeply joint language-knowledge foundation model from both sources like textual and KG at any scale. DRAGON model takes the input pairs of text segments, and the relevant KG sub-graph (the KG sub-graph retrievals phase is the same as used in QAGNN and GreaselM) as input and bi-directionally joins information from both modalities of inputs. DRAGON pre-train on two self-supervised reasoning tasks: **Masked Language modeling and the Knowledge Graph link predictions.**

Language models (LMs) are pre-trained on large amounts of text data, such as BERT and GPTs, and have shown strong performance on many natural language processing (NLP) tasks. However, effectively combining text and KGs for pre-training is an open problem and presents challenges.

The core components of DRAGON:

- Deeply bi-directional model for the two modalities (LM and KG) to interact.

- Self-supervised task to learn joint reasoning over text and KG.

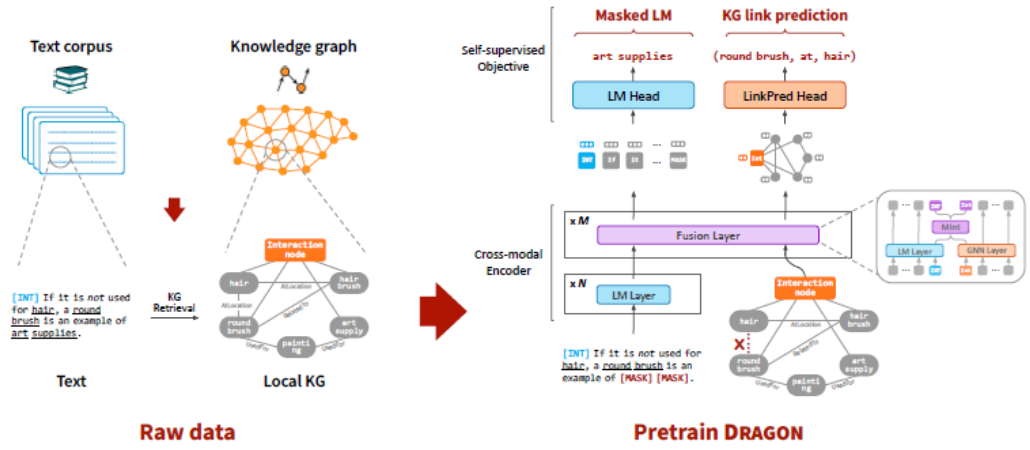


Figure 2.7: General architecture of Dragon.

Approach On the left of the image, the given input, raw data of a text corpus, and an extensive knowledge graph construct the aligned (text, local KG) pairs by sampling a text segment from the text corpus and extracting a relevant subgraph from the large KG. As in structured knowledge, KG can ground the text, and the text can provide the KG with a rich context for reasoning. The Dragon Models aims to pre-train a language-knowledge model on the right side of the image for a common reason on the text-KG pairs (DRAGON).

To model the interactions over text and KG, DRAGON and GreaseLM work similarly to using a cross-modal encoder that bi-directionally exchanges information between them to produce fused text token and KG node representations.

To pre-train DRAGON jointly on text and KG, they unify two self-supervised reasoning tasks

- Masked Language Modeling (MLM).
- Link prediction or Knowledge graph link prediction.

Knowledge graph link prediction holds out some edges from the input KG and then predicts them. This joint objective encourages text and KG to mutually inform each other, facilitating the model to learn joint reasoning over text and KG. Masked Language Modeling MLM took some tokens as input and masked some of them and tried to predict those masked tokens again from the sequence using a language model encoder like BERT.

Results Dragon's results are quite better than previous solutions.

Method	MedQA
BioBERT [74]	36.7
PubmedBERT [75]	38.1
BioLinkBERT [19]	44.6
+ QAGNN	45.0
+ GreaseLM	45.1
DRAGON (**Ours**)	**47.5**

Figure 2.8: Test accuracy of Dragon on MedQA-USMLE

2.2.4 Subgraph Retrieval Enhanced Model for Multi-hop Knowledge Base Question Answering

This paper [16] is used as work for the thesis and proposes a trainable subgraph retriever (SR) decoupled from the subsequent reasoning process that enables a plug-and-play framework to enhance any subgraph-oriented KBQA model.

Approach This work proposes a subgraph retrieval-enhanced model for KBQA that separates subgraph retrieval from the subsequent reasoning process and allows reasoning on the final entire subgraph instead of intermediate partial subgraphs. The paper investigates different training strategies for the subgraph retriever, including weakly supervised/unsupervised pre-training and end-to-end fine-tuning with the reasoner. They conduct experiments on WebQSP and CWQ. The results show that the proposed model achieves better performance, creates new state-of-the-art results for embedding-based KBQA models, and improves the overall efficiency of KBQA.

The information in the KG is organized in the form of triples which contains the head, tail, and relations. $G = (e, r, e0)$ where the E and R represents the set of entities and set relations, respectively. Given the question q, KG-based retrieval aims to figure out the answer A_q to the question from the set of entities E of Knowledge Graph G. The entities E_q mentioned in the given question $\mathbf{q}$ are called topic entities, which are assumed to be given.

In the probabilistic formulation, we aim to maximize the probability of answers a to the question by measuring the probability of answer a given by

graph G and question q, respectively.

$$p(a/G, q) = \sum_{G} P\phi(a/q, G_s)\ P\theta(G_s/q) \tag{2.1}$$

Instead of performing direct reasoning on the entire G, they need to retrieve the subgraph G_s from the Knowledge graph G and then infer the answer a. According to the given Equation 2.1, they have two modalities, the first is to retrieve the portion of subgraph G_s as per given the question q, and the second one is to infer the answer given by the retrieved subgraph G_s and the question. The goal is to find the optimal parameters θ (subgraph retriever G_s conditioned by question q) and ϕ (inferring the answer condition by question q and the G_s) that maximize the log-likelihood of training data as shown in Equation 2.2.

$$L(\theta, \phi) = max_{\theta}, \phi \sum_{(q,a,G)\epsilon D} \log \sum_{G_s} P\phi(a/q, G_s)\ P\theta(G_s/q) \tag{2.2}$$

D contains the whole training dataset, and here they present the concept of decoupled phase for the retrieval $p\theta$ and the reasoner $p\phi$. Both parts (retriever $p\theta$ and reasoner $p\phi$) can be optimized respectively, as shown in Equation 2.3.

$$L(\theta, \phi) = max_{\theta,\phi} \sum_{(q,a,G)\epsilon D} \log\ P\phi(a/q, G_s)\ P\theta(G_s/q) \tag{2.3}$$

Topic entities **Eq** are our starting points, and they expand routes from the topic entities and induce a corresponding tree. When they have generated trees from each topic entity, they merge those induced trees into the standard unified graph using the union techniques. After all the work, as a result, they have a standard graph. It is used for the next reasoner phase to perform reasoning to infer the answer to the given question **q** instead of in the initial stage on the KG without filtering anything.

For finding paths, the method involves expanding top-K paths relevant to the question from the topic entities and then inducing the subgraph following these paths. The expansion process starts from a topic entity and follows a sequential decision process to select the next relation to expand. The relevance of each relation to the question is measured by the dot product between their embeddings, and the probability of a relation being expanded can be formalized as a function of its relevance score. The method also involves performing a top-K beam search each time to get K paths and taking the union of top-K trees from one topic entity into a single subgraph. Finally, the method merges the same entities from different subgraphs to induce the final subgraph. This

can reduce the subgraph size, i.e., the answer reasoning space, as the subgraphs from different topic entities can be viewed as the constraints of each other.

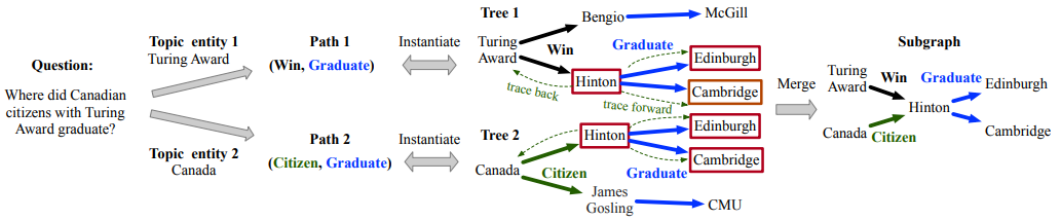


Figure 2.9: They expand a path from each topic entity as well as induce a corresponding tree, then merge the trees from different topic entities to form a unified subgraph.

For example, in the figure above, given the question "Where did Canadian citizens with Turing Award graduate?" with two topic entities "Turing Award" and "Canada", they can explain it by the two expanded paths (Win, Graduate) and (Citizen, Graduate) and merge the trees induced by them to form a unified subgraph. Only the top-1 path is presented in the figure for a clear illustration.

Results The authors conduct experiments to evaluate the proposed subgraph retrieval (SR) enhanced model for knowledge-based question answering (KBQA). The experiments are designed to answer four main questions:

- Does SR improve the QA performance?
- Can SR obtain smaller but higher-quality subgraphs?
- How do weakly supervised and unsupervised pre-training affect SR's performance?
- Can end-to-end fine-tuning enhance the performance of the retriever and the reasoner?

The authors evaluate the proposed model using WebQuestionSP (WebQSP) and Complex WebQuestion 1.1 (CWQ) benchmarks. They evaluate the retriever using the answer coverage rate, denoted as Hits@K, and the QA performance using Hits@1 and F1 score. They compare the proposed model with embedding-based KBQA models, such as EmbedKGQA, KV-Mem, BAMNet, GRAFTNet, NSM, PullNet, and SP-based models as QGG, SPARQA, and CBR-KBQA.

The authors show that the proposed model improves the QA performance on both datasets compared to the baseline models. For example, SR+NSM improves 0.4% Hits@1 and 1.3% F1 on WebQSP and 3.9% Hits@1 and 4.7%

F1 on CWQ compared with the original NSM. They also demonstrate that SR can be adapted to different subgraph-oriented reasoners, such as GRAFT-NET, which significantly improves 9.7% Hits@1 and 8.7% F1 on CWQ. However, the authors do not inject SR into BAMNet as the model requires entity types in the subgraph, which is not yet supported by SR.

The authors conclude that SR improves the QA performance when injecting it before a subgraph-oriented reasoner, and the proposed model equipped with NSM creates a new state-of-the-art model for embedding-based KBQA.

Challenges and limitations I encountered several challenges and limitations when extending this method to the USMLE dataset. Some of these limitations may include the following:

- Data complexity: USMLE data is much more complex and structured than the datasets used in the experiments described in this article, which may require additional data preprocessing and cleaning efforts.
- Data dimensionality: USMLE data may be much larger than the datasets used in this article, which may increase the problem's dimensionality. This may require dimensionality reduction techniques and greater attention to overfitting issues.
- Problem complexity: the problem in the medical field can be very complex and may require specialized domain knowledge to create an accurate and useful system. Additionally, answers to medical questions may vary depending on specific circumstances and contexts.
- Model performance: the effectiveness of the model will largely depend on the quality of the embeddings for paths. If the embeddings do not adequately represent the structure and relationships of the medical domain graph, the model may not perform well.

Chapter 3

Method

This chapter will analyze my solution, representing the last solution shown in the related works but extended and improved. The next chapter will detail the models and technologies adopted in the experiments.

3.1 KG embedding

First, the knowledge graph has been densified thanks to a graph embedding model, unlike the discrete space used by authors via SPARQL queries. This greatly improves the efficiency of predicting missing components in triples in the form of head, relation, and tail. Nevertheless, in dense space, the prediction of the components is approximately compared to the use of the queries and does not seem correct, but all this depends on the KG embedding model used.

3.2 Preprocessing

Next, several preprocessing steps on the data are used to prepare the dataset and train the language model. The preprocessing part can be defined as a sort of pipeline where the output of each phase is used as input for the next one.

3.2.1 K-hop path finding

As a first phase of the preprocessing, the training dataset is loaded, and for each instance, the question, the topic entities, and the answers to the question are extracted in a new structure used in the next phase.

The second phase of preprocessing takes as input the output of the first part in which, for each instance, we have the question, topic entities, and answers, and we try to obtain from each pair of topic entities and answer entities the

possible path that connects them using the initially defined embeddings. In particular, we try to obtain two types of paths for each pair:

- Single-hop: the possible relationship that connects the topic entity (head) to the response entity (tail) is sought. This way, all possible relationships are tested, from the topic entity to the interested tail.
- Double-hop: any two relationships that connect the topic entity are sought; in this case, there will be an intermediate entity between the two relationships which is called the "bridge node" which is the middle node between the relationships, the possible path will be like this defined: **topic entity - first relation - bridge node - second relation - answer entity**. There are two alternatives to finding the double-hop path:
  - The first is to start from the topic entity and predict all tails starting from all relationships; from the predicted tail, repeat the process with all relationships and predict all tails; if one or more tails coincide with the answer entity, we found a path.
  - The alternative, later used in the solution, could be to always start initially from the topic entity and predict all the tails from all the relations, then start from the answer entity, considering all tails and predicting the head using all the relations, if the predicted head coincides with the tail predicted by the topic entity then we have found a path.

Using the embeddings to predict the missing component, it was decided to use a **K** to obtain the first most probable K; in our case, the most probable entity is taken, i.e., the first, furthermore the results are filtered according to the distance in the prediction uses a **threshold**, the threshold allows us to have more true positives than false positives.

An example of pseudo-code in pathfinding could be the following:

```
function double_hop(topic_entity, answer_entity)
    global relations
    global embeddings

    k = 1
    threshold = 1

    paths = array()

    for relation in relations
        tail = embeddings.predict_topk_tail(topic_entity, relation, k, threshold)
        for relation_ in relations
            head = embeddings.predict_topk_heads(answer_entity, relation_, k, threshold)
            if head == tail
                p = array
                p.add(topic_entity)
                p.add(relation)
                p.add(tail)
                p.add(relation_)
                p.add(answer_entity)

                paths.add(p)

    relations_path = array
    for path in paths
        relations_path.add(array(path[1], path[3]))

    // Only pairs of relations of the hop
    return relations_path
```

```
function single_hop(topic_entity, answer_entity)
    global relations
    global embeddings

    k = 1
    threshold = 1

    paths = array()
    for relation in relations
        tail = embeddings.predict_topk_tails(topic_entity, relation, k, threshold)

        if tail == answer_entity
            paths.add(array(topic_entity, relation , answer_entity))

    relations_path = array
    for path in paths
        relations_path.add((path[1]))

    // Only relations are useful
    return relations_path

function path_finding(topic_entities, answer_entities)
    paths = array

    for topic_entity in topic_entities
        for answer_entity in answer_entities
            p = array
            p.add(single_hop(topic_entity, answer_entity))
            p.add(double_hop(topic_entity, answer_entity))

            paths.add(p)

    return paths
```

Different processes have been used to speed up the pathfinding process, in which each process takes part in instances that speed up a lot.

The solution through the embeddings can be extended to be used with multi-k-hop. Using more hops, the time increases.

After finding the possible paths of all the dataset instances, the single and double relationships used in the next phase are stored for each instance.

3.2.2 Path scoring

After finding the possible paths of each instance, a score from 0 to 1 is used for each path found, which indicates how much the hit of the response is compared to the result of the prediction obtained by instantiating the path. It indicates the reliability of the path, considering that a topic entity with a relationship could go to different tails. In this case, the instances with no path are discarded, and the others are kept.

For each path found in the instance, we try to find the entities with which we arrive at those relationships depending on whether the path contains a relationship or two (single-hop or double-hop) to carry out. The intersection of

the tails is found with the answer entities of the instance and, finally, calculate the hits.

In pseudo-code, it can be summarized as follows:

```
function deduce_leaves_by_path_one(src, rel)
    k = 100
    paths = array()
    for relation in relations
        if relation != rel
            continue

        tail = self.distmult_embedded_kg.predict_topk_tails(topic_entity, first_hop_relation, k,
            self.distance_threshold)
        if tail
            if k == 1:
                paths.add(array(topic_entity, first_hop_relation, tail))
            else:
                for tail_entity in tail
                    paths.add(array(topic_entity, first_hop_relation, tail_entity))

    relations_path = array()
    for path in paths
        relations_path.add((path[2]))

    return relations_path

function deduce_leaves_by_path(src, path)
    # Call method based on length of the path
    if len(path) == 0:
        return array(src)
    elif len(path) == 1:
        return deduce_leaves_by_path_one(src, path)
    elif len(path) == 2:
        return deduce_leaves_by_path_two(src, path)

function cal_path_score(knowledge_graph, topic_entity, path, answers)
    preds = knowledge_graph.deduce_leaves_by_path(topic_entity, path)
    preds = {}
    hit = preds & answers
    full = preds

    if full == None
        return 0

    return len(hit) / len(full)
```

The function deduce_leaves_by_path_two is identical to single-hop except that one further hop is made.

Also, in this case, it has been parallelized to speed up the process greatly.

The scores obtained will be used in the next (last) step of negative sampling.

3.2.3 Negative sampling

Negative sampling is a technique used in machine learning for training models, where one tries to find negative examples for the model. In this

case, the authors use negative sampling to generate training examples for a question-answering model that uses a knowledge graph.

The authors, for each instance, consider only the paths with a score of at least 0.5, i.e., reliable paths.

```
function generate_data_list(path_json_obj, json_obj, pos_rels, kg: KnowledgeGraphGraphDB)
    new_data_list = array()
    neg_num = 15

    path = path_json_obj["path"]
    path = path + [END_REL]
    question = json_obj["question"] + " [SEP]"
    topic_entities = json_obj["topic_entities"]

    filter_threshold = 5
    current_filter_threshold = 1
    filter_flag = False

    for rel in path[:-1]
        current_filter_threshold *= filter_threshold

        candidate_entities = set()
        for h in topic_entities
            candidate_entities |= set(kg.get_hr2t_with_limit(h, rel, current_filter_threshold + 1))
            if len(candidate_entities) > current_filter_threshold:
                break

        if len(candidate_entities) > current_filter_threshold
            filter_flag = True
            break

    if filter_flag
        return None

    prefix_list = array()
    for rel in path
        prefix = ",".join( prefix_list )
        prefix_list .add(rel)

        data_row = []
        data_row.add(question)
        data_row.add(rel)

        neg_rels = set()
        for h in topic_entities
            neg_rels |= set(kg.get_relation(h, limit=100))
            if len(neg_rels) > 100:
                break
        neg_rels = list(neg_rels)
        neg_rels.append(END_REL)
        neg_rels = [r for r in neg_rels if r not in pos_rels[prefix ]]

        if len(neg_rels) > 0
            sample_rels = []
            while len(sample_rels) < neg_num:
                sample_rels.add(neg_rels)

            neg_rels = random.sample(sample_rels, neg_num)
            neg_rels = neg_rels[:neg_num]
```

```
            data_row.extend(neg_rels)
            new_data_list.add(data_row)

        if rel != END_REL
            next_question = question + f" {rel} #"
            question = next_question
            topic_entities = kg.deduce_leaves_from_src_list_and_relation(topic_entities, rel)

    return new_data_list
```

Authors start with a question and a set of entities relevant to that question (topic entities) and try to generate pairs of questions and relationships that can be used as training examples for the question-answering model. In particular, they start from a relationship (rel) and try to generate a set of entities with that relationship with the entities in topic entities. Negative examples are also generated for each question-relation pair by selecting random relationships not present in the knowledge graph.

Negative sampling, therefore, consists of trying to generate negative examples as similar as possible to the positive ones but classified as negative by the model. This is important because it improves the model's ability to distinguish between positive and negative examples and avoids problems such as overfitting.

All these pre-processing phases, in which the output of each one becomes the input of the next one, generate a file used for model training with a biomedical language model.

3.3 Subgraph retrieval

Following the preprocessing phase, a language model is trained to be used as a model and tokenizer in the last phase of subgraph retrieval. The language model is trained on the preprocessing output phase.

Finally, the part relating to the subgraph retrieval is performed as follows:

- Initially, the data relating to the training, validation, and test of the dataset are loaded. It also retrieves the relationships and entity names used in the process.
- For each question:
  - The entities of the question are taken and the paths are searched in the knowledge graph using the Beam Search [17] technique, i.e., at each iteration the most promising paths are selected (based on the score assigned to the paths), and these are used as candidates for the search for subsequent paths. The search continues until all required paths are found, the maximum number of hops is reached, or no more paths are available to explore.

- Subsequently, a sub-graph is constructed for each candidate path
- Finally, all the subgraphs obtained are merged by merging all the nodes and triples present in the two graphs, eliminating any duplicate nodes and triples.

Chapter 4

Experimental Setup

This chapter first briefly describes the development environment in which the various experiments were tested; subsequently, the solution is studied much more technically in the individual aspects, from the dataset to the knowledge base used and the implementation changes made concerning the authors.

4.1 Environment

For the execution of the various experiments, the university cluster was used, made up of 3 servers, the main server used is a workstation with four NVIDIA GeForce RTX3090 GPUs with 24GB of dedicated memory, 124GB of RAM, and an Intel® Core™ i9-10900X.

SLURM [18] was used on the cluster to perform GPU-intensive tasks; that is a cluster manager that allows us to dynamically schedule and allocate tasks on our GPUs together with command sbatch allowing you to launch tasks on the cluster in a non-blocking way. This way, running multiple experiments simultaneously on the server was possible.

Google Colab was also used for training the embedding model.

The programming language mainly used for the implementation is Python, with small scripts in Bash; it was possible to connect to the server thanks to SSH using Visual Studio Code at first and then PyCharm for better use.

4.1.1 ChatGPT

ChatGPT [19] has been of great help during the development of the whole thesis starting from the configuration of the knowledge base to the various adaptations of the authors' SPARQL queries to ours; it has also been of great help in part relating to embeddings. ChatGPT was used as an aid and did not replace my work.

4.2 Dataset

Unlike the authors who use the WebQuestionSP and CWQ datasets, in our case, we use the **MedQA-USMLE (MedQA)**, a 4-way multiple-choice task containing the United States Medical License Exam questions. The dataset has 12,723 questions and in our case, the dataset was downloaded from the DRAGON [20] repository with the KG, which contains much more relations than the original of QA-GNN.

A database instance is composed as follows:

- "sent" represents the question
- "ans" represents the response entity label
- "ac" represents the response entity identifier
- "qc" is a list of identifiers of the entities present in the question (topic entities)

Example:


```
{
    "sent": "A 23−year−old pregnant woman at 22 weeks gestation presents with burning upon
        urination. She states it started 1 day ago and has been worsening despite drinking more water
        and taking cranberry extract. She otherwise feels well and is followed by a doctor for her
        pregnancy. Her temperature is 97.7\u00b0F (36.5\u00b0C), blood pressure is 122/77 mmHg,
        pulse is 80/min, respirations are 19/min, and oxygen saturation is 98% on room air. Physical
        exam is notable for an absence of costovertebral angle tenderness and a gravid uterus. Which of
        the following is the best treatment for this patient? Ampicillin",
    "ans": "Ampicillin",
    "qc": ["C0005823", "C0005824", "C0005903", "C0031809", "C0032961", "C0034107", "C0035203",
        "C0039476", "C0043047", "C0085624", "C0087111", "C0175627", "C0227812", "C0232117",
        "C0235634", "C0332197", "C0391850", "C0439228", "C0439230", "C0439475", "C0523807",
        "C0549206", "C0567091", "C0600457", "C0684271", "C0746961", "C1271104", "C1272641",
        "C1550678", "C1689985"],
    "ac": ["C0002680"]
}
```


There are 99 relationships in the KG, such as may_cause, may_treat, drug_family_of, etc...

The relationships between the entities are present in a CSV provided by DRAGON in the form of relationship, tail, and head.

The entities are uniquely identified with labels in a text file associating each identifier with the entity's name.

This information was used to organize the structure better and have a CSV containing all the triples in the classic form of subject, predicate, and object (head, relation, and tail).

This structure makes importing data in a triple format possible thanks to a knowledge base such as GraphDB.

HEAD	RELATION	TAIL
Percutaneous approach	access_of	Intrauterine transfusion
Drugs and hormones	belongs_to_the_category_of	Carbamazepine
Injury due to legal restraint	associated_with	Physical restraint procedure

Table 4.1: Caption

4.3 Knowledge base

The paper's authors used Virtuoso [21] knowledge base to import and use their knowledge graph.

Virtuoso is a platform for modern data access, integration, virtualization, and multi-model data management (tables and graphs) based on innovative support of existing open standards (e.g., SQL, SPARQL, and GraphQL).

Virtuoso is a powerful and efficient engine that is simultaneously complex and difficult to use. It is also recommended to run Virtuoso on a server with at least 100GB of RAM.

For these reasons, it was decided to use GraphDB as it is much simpler, more intuitive, and has already been used previously, so it didn't need too much time to use it correctly.

4.3.1 GraphDB

GraphDB [22] is a database that uses nodes and arcs to represent and store information. Graph databases using the **Resource Description Framework** (RDF) [23] model. The version of GraphDB used is 8.11.1.

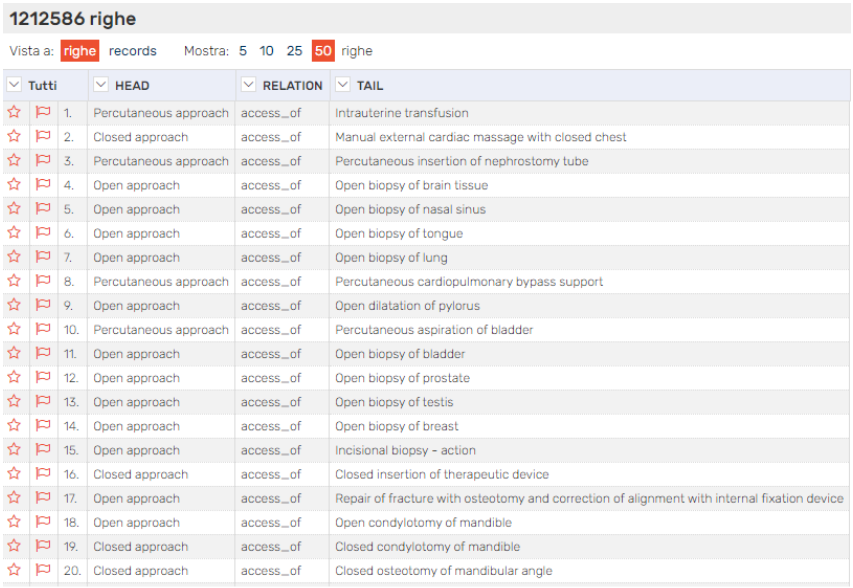
To query data in RDF on a graph database such as GraphDB, the **SPARQL** language is used, which allows the construction of queries based on patterns of triples, optional patterns, and logical conjunctions/disjunctions.

In this case, GraphDB was run in a docker container on the server via docker-compose with the following script:

```
{
    version: "3"
    services:
        graphdb:
            container_name: "graphdb"
            image: "ontotext/graphdb:8.11.1-se"
            restart: always
            ports:
              - "7200:7200"
            volumes:
              - "./graphdb/graphdb-home:/opt/graphdb/home"
}
```

The port used by default by GraphDB is 7200, and it has been used both as an internal and external port to docker; this is important when you want to connect to the GraphDB UI locally through SSH.

Another substantial reason why it was decided to use GraphDB instead of Virtuoso is that the interface provides the function of importing data from an RDF file or URL; alternatively, you can import data in a tabular format such as Excel, CSV, etc... In the case of USMLE, we can import the data through the file CSV file of triplets.



1212586 righe

Vista a: righe records Mostra: 5 10 25 50 righe

Tutti		HEAD	RELATION	TAIL
1.		Percutaneous approach	access_of	Intrauterine transfusion
2.		Closed approach	access_of	Manual external cardiac massage with closed chest
3.		Percutaneous approach	access_of	Percutaneous insertion of nephrostomy tube
4.		Open approach	access_of	Open biopsy of brain tissue
5.		Open approach	access_of	Open biopsy of nasal sinus
6.		Open approach	access_of	Open biopsy of tongue
7.		Open approach	access_of	Open biopsy of lung
8.		Percutaneous approach	access_of	Percutaneous cardiopulmonary bypass support
9.		Open approach	access_of	Open dilatation of pylorus
10.		Percutaneous approach	access_of	Percutaneous aspiration of bladder
11.		Open approach	access_of	Open biopsy of bladder
12.		Open approach	access_of	Open biopsy of prostate
13.		Open approach	access_of	Open biopsy of testis
14.		Open approach	access_of	Open biopsy of breast
15.		Open approach	access_of	Incisional biopsy - action
16.		Closed approach	access_of	Closed insertion of therapeutic device
17.		Open approach	access_of	Repair of fracture with osteotomy and correction of alignment with internal fixation device
18.		Open approach	access_of	Open condylotomy of mandible
19.		Closed approach	access_of	Closed condylotomy of mandible
20.		Closed approach	access_of	Closed osteotomy of mandibular angle

Figure 4.1: Import of data in usmle triple format in GraphDB.

4.3.2 SPARQL queries

The endpoint provided by GraphDB makes it possible to execute SPARQL queries on the imported knowledge graph.

Before the densification through embedding the knowledge graph, we tried to reproduce the work done by the authors through SPARQL queries. The queries were adapted to the use of GraphDB and simplified.

For example, our query adapted searches single-hop paths from a topic entity to an answering entity in SPARQL:

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
SELECT distinct ?r0 WHERE {{
    SERVICE <ontorefine:2087909607654> {{
        ?row a <http://rdf.graphdb.com/ns/Row> ;
            <http://rdf.graphdb.com/ns/HEAD> "{topic_entity}" ;
```

```
            <http://rdf.graphdb.com/ns/RELATION> ?r0 ;
            <http://rdf.graphdb.com/ns/TAIL> "{answer_entity}" .
    }}
}}
```

The query searches for all relationships with the topic entity and answer entity provided by labeling them as **?r0** and returning them uniquely.

While double-hop can be implemented in SPARQL as follows:

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
SELECT distinct ?r1 ?r2 WHERE {{
    SERVICE <ontorefine:2087909607654> {{
        ?row a <http://rdf.graphdb.com/ns/Row> ;
            <http://rdf.graphdb.com/ns/HEAD> "{topic_entity}" ;
            <http://rdf.graphdb.com/ns/RELATION> ?r1 ;
            <http://rdf.graphdb.com/ns/TAIL> ?e1 .

        ?row a <http://rdf.graphdb.com/ns/Row> ;
            <http://rdf.graphdb.com/ns/HEAD> ?e1 ;
            <http://rdf.graphdb.com/ns/RELATION> ?r2 ;
            <http://rdf.graphdb.com/ns/TAIL> "{answer_entity}" .

    }}
}}
```

In this case, we are looking for the two possible relationships that connect the topic entity to the answer entity. As many queries have been adopted.

4.4 Knowledge graph embedding

As mentioned in the method chapter, SPARQL queries are effective but equally very inefficient since the average execution time was at least 30/40 seconds depending on the type of queries (the more complex and the more logic they require, the longer they take time). Moreover, GraphDB didn't allow any optimization mechanism to speed them up. An attempt was made to use a disk caching mechanism to "skip" the previously executed queries, but the times remained too long.

Therefore, it was preferred to train a graph embedding model to significantly speed up the process at the expense of lower accuracy.

The Google Colab platform was mainly used and subsequently integrated and used on the cluster to train the knowledge graph embeddings.

Two embedding methods were mainly tested during the various tests: TransE and DistMult. The pykeen library in Python was used to train embedding models on knowledge graphs and use the preferred model.

4.4.1 TransE

TransE [24] was initially chosen as the embedding model, a knowledge graph embedding model used in natural language processing. TransE aims to learn a low-dimensional vector representation of each entity and relationship in a knowledge graph. The vectors capture the semantic meaning of the entities and relationships and can be used to predict missing relationships in the graph.

The key idea behind TransE is to represent each relationship in the graph as a translation operation between the vectors representing the two entities involved in the relationship. Specifically, for each triple of (head, relation, and tail), TransE learns a vector representation for the head and tail entities and a vector representation for the relationship. The vector representing the tail entity is obtained by adding the vector representing the head entity and the vector representing the relationship.

During training, TransE learns to minimize a distance-based loss function that measures the discrepancy between the predicted and actual relationships in the knowledge graph. The model uses the L1 or L2 distance between the predicted tail entity vector and the actual tail entity vector as the loss function and updates the entity and relationship vectors to minimize this distance.

The training encompassed 100 epochs using early stopping, which allows the training to stop if there is no improvement between epochs:

```
from pykeen.triples import TriplesFactory
from pykeen.pipeline import pipeline

# Split triplets in train and test set (80/20)
tf = TriplesFactory.from_path('/content/triplets.txt')
train, valid, test = tf.split([.8, .1, .1])

# Knowledge graph embedding using TransE
result = pipeline(
    training=train,
    validation=valid,
    testing=test,
    model='TransE',
    stopper='early',
    epochs=100
)
```

4.4.2 DistMult

DistMult [25] is another embedding method that uses a technique called "bilinear projection".

This involves projecting the entity and relationship vectors into a shared space, where the dot product can be computed. The bilinear projection function is learned during training. The resulting vectors are optimized to minimize a

loss function that measures the discrepancy between the predicted and actual relationships in the knowledge graph.

It has been shown that DistMult is one of the best graph embedding methods for the MedQA-USMLE dataset [26], and in fact, it has shown better results at TransE, In the pykeen pipeline change the model type to "DistMult":

```
from pykeen.triples import TriplesFactory
from pykeen.pipeline import pipeline

# Split triplets in train and test set (80/20)
tf = TriplesFactory.from_path('/content/triplets.txt')
train, valid, test = tf.split([.8, .1, .1])

# Knowledge graph embedding using DistMult
result = pipeline(
    training=train,
    validation=valid,
    testing=test,
    model='DistMult',
    stopper='early',
    epochs=100
)
```

4.4.3 Metrics

Three metrics were used to evaluate the trained embedding models:

- **Mean Rank (MR)**: measures how well a knowledge graph embedding model performs on a given set of test triples and is calculated as the average rank of the correct answer in the ranked list of predicted answers. A lower mean rank indicates better performance, as the correct answer is ranked higher in the list of predicted answers, and the range of values goes from 1 to the number of entities.

- **Mean Reciprocal Rank (MRR)**: like the mean rank metric, it measures the model's performance in predicting the correct answer for a given test triple in the knowledge graph. However, the MRR metric considers the order of the ranked list of predicted answers rather than just the rank of the correct answer. Specifically, for each test triple, the reciprocal rank of the correct answer is calculated as 1/rank, where rank is the position of the correct answer in the ranked list of predicted answers. The MRR is then calculated as the average of these reciprocal ranks across all test triples. A higher MRR indicates better performance, as the correct answers are ranked higher on average in the list of predicted answers, and the range of values goes from 0 to 1.

- **Inverse Geometric Mean Rank (IGMR)**: is a variant of the Mean Rank evaluation metric designed for the link prediction task in a knowledge graph containing inverse relations. Inverse relations are pairs of relationships in the knowledge graph that are the reverse of each other (e.g., "father_of" and "child_of"). The IGMR metric calculates the mean rank of the correct answer (i.e., the true tail entity) in the ranked list of predicted answers, considering both the forward and inverse relationships in the knowledge graph. It generates two ranked lists for each test triple: one using the forward relationship and one using the inverse relationship. Then, it calculates the rank of the correct answer in both ranked lists and takes the minimum rank as the final rank for that test triple. The IGMR metric then calculates the mean of these minimum ranks across all test triples. A higher IGMR indicates better performance, as the correct answers are ranked higher on average in the list of predicted answers, and the range of values goes from 0 to 1.

The Mean Rank metric was always very good in both methods, while the others were much less.

In addition to the metrics, a small script has also been developed to evaluate how many triples are correctly predicted:

```
def get_row_where(head=None, relation=None, tail=None):
  result = []
  # Only tail
  if tail and not head and not relation:
    for row in rows:
      if row[2] == tail:
        result.append(row)
  elif head and not relation and not tail: # Only head
    for row in rows:
      if row[0] == head:
        result.append(row)
  elif head and relation and not tail: # Head and relation
    for row in rows:
      if row[0] == head and row[1] == relation:
        result.append(row)
  elif relation and tail and not head: # Relation and tail
    for row in rows:
        if row[1] == relation and row[2] == tail:
          result.append(row)
  elif head and tail and not relation: # Head and tail
    for row in rows:
        if row[0] == head and row[2] == tail:
          result.append(row)
  elif relation and not head and not tail: # Only relation
    for row in rows:
        if row[1] == relation:
          result.append(row)

  return result
```

In this way, using the model to predict, for example, the tail, it is possible to obtain the missing component of a real triple of the dataset and compare the predicted tail to the real one. The results are quite variable; on some triples, the model performs well on others, less so, but in most cases, it always identifies the suitable space context even if the predicted entity is not the 100% correct one.

4.4.4 FAISS

Although graph embedding methods greatly sped up execution times, it was not enough. Indeed PyKeen performs a forward pass through a neural network model to compute the similarity scores between a given entity and all possible entities in the knowledge graph and returns the top-k highest-scoring entities, which can be computationally expensive. For this reason, **FAISS** (Facebook AI Similarity Search) [27] was used to speed up further, developed by Facebook AI Research provides several algorithms for similarity search, including brute-force search, k-nearest neighbors (KNN) graph construction, and approximate nearest neighbor (ANN) search using several indexing methods such as IVF (inverted file) and PQ (product quantization). These algorithms can be used to search large-scale databases of high-dimensional vectors efficiently.

A Python class has been designed and implemented that take the embeddings trained by PyKeen and use them for FAISS:

```
class DistMultEmbeddedKG:

    def __init__(self, model, triples_factory):
        # Entity and relation embeddings from pykeen's model
        self.entity_embs = model.entity_representations[0]().detach().numpy()
        self.relation_embs = model.relation_representations[0]().detach().numpy()

        # Create a FAISS index containing precomputed entity embeddings
        self.entity_idx = faiss.IndexFlatIP(self.entity_embs.shape[1])
        self.entity_idx.add(self.entity_embs)

        # Bi-directional dict for:
        # Label entity -> entity ID
        # Relation entity -> relation ID
        self.entity_dict = bidict(triples_factory.entity_to_id)
        self.relation_dict = bidict(triples_factory.relation_to_id)
```

Subsequently, a method was developed to predict the top k tails starting from a head, a relation, a parameter k which indicates the k most probable entities, and a distance threshold which indicates the minimum threshold to be considered in the similarity distance.

The embeddings are retrieved from the entity and the relationship. Being a DistMult model, the multiplication of matrices must be carried out between the

embeddings of the head with those of the relation to obtain the tail. Finally, the first k labels of the entities are obtained.

```
def predict_topk_tails(self, h, r, k=5, distance_threshold=1):
    # From labels to ids
    if h not in self.entity_dict:
        raise KeyError("Head does not exist")
    if r not in self.relation_dict:
        raise KeyError("Relation does not exist")

    h_id = self.entity_dict[!htb]
    r_id = self.relation_dict[r]

    h_emb = self.entity_embs[h_id]
    r_emb = self.relation_embs[r_id]
    query_emb = (h_emb * r_emb).reshape(1, -1) # Add extra wrapping dimension for FAISS

    # Search top-k nearest tail entities with FAISS
    D, I = self.entity_idx.search(query_emb, k) # distance, index
    tail_indeces = I.squeeze()

    if k > 1:
        tail_labels = []
        for idx, tail_index in enumerate(tail_indeces):
            if tail_index not in self.entity_dict.inv:
                continue

            if D[0][idx] >= distance_threshold:
                tail_label = self.entity_dict.inv[tail_index]
                tail_labels.append(tail_label)

        return tail_labels

    if D[0][0] >= distance_threshold:
        tail_index = I[0][0]
        tail_label = self.entity_dict.inv[tail_index] if tail_index in self.entity_dict.inv else None

        return tail_label

    return None
```

The same thing was done to obtain the top k heads; unlike the previous method, a matrix multiplication is performed between the embeddings of the relation and of the tail.

Thanks to FAISS, it was possible to carry out pathfinding in single-hop and double-hop much more quickly. Furthermore, a cache mechanism has been used, which saves to disk every 20 seconds and is loaded into memory at startup to speed up the process.

The single hop from a topic entity to an answering entity works in this way using FAISS: for each relationship, we try to get the queue starting from the topic entity, saving the result of the triple if it is not present in the cache, then we check if the found queue represents the answer entity if it is then we have found a path.

```
paths = []
```

```
for first_hop_relation in self.relations:
    first_hop_tail_entity = self._load_from_dict_faiss(topic_entity, first_hop_relation,
        self.distance_threshold)
    if not first_hop_tail_entity:
        first_hop_tail_entity = self.distmult_embedded_kg.predict_topk_tails(topic_entity,
            first_hop_relation, 1, self.distance_threshold)
        if first_hop_tail_entity:
            self._save_to_dict_faiss(topic_entity, first_hop_relation, first_hop_tail_entity,
                self.distance_threshold)

    if first_hop_tail_entity and first_hop_tail_entity == answer_entity:
        paths.append([topic_entity, first_hop_relation, answer_entity])

relations_path = []
for path in paths:
    relations_path.append((path[1]))

return relations_path
```

The double hop similarly to the single hop makes a further hop starting from the answer entity trying to find the common bridge node (ie the one in the middle between the two relationships).

```
paths = []

for first_hop_relation in self.relations:
    first_hop_tail_entity = self._load_from_dict_faiss(topic_entity, first_hop_relation,
        self.distance_threshold)
    if not first_hop_tail_entity:
        first_hop_tail_entity = self.distmult_embedded_kg.predict_topk_tails(topic_entity,
            first_hop_relation, 1, self.distance_threshold)
        if first_hop_tail_entity:
            self._save_to_dict_faiss(topic_entity, first_hop_relation, first_hop_tail_entity,
                self.distance_threshold)

    if first_hop_tail_entity:
        for second_hop_relation in self.relations:
            head_entity_prediction = self._load_from_dict_faiss(answer_entity,
                second_hop_relation, self.distance_threshold)

            if not head_entity_prediction:
                head_entity_prediction =
                    self.distmult_embedded_kg.predict_topk_heads(answer_entity,
                    second_hop_relation, 1, self.distance_threshold)

                if head_entity_prediction:
                    self._save_to_dict_faiss(head_entity_prediction, second_hop_relation,
                        answer_entity, self.distance_threshold)

                if head_entity_prediction and head_entity_prediction == first_hop_tail_entity:
                    p = []

                    p.append(topic_entity)
                    p.append(first_hop_relation)
                    p.append(first_hop_tail_entity)
                    p.append(second_hop_relation)
                    p.append(answer_entity)

                    paths.append(p)
```

```
relations_path = []
for path in paths:
    relations_path.append((path[1], path[-2]))
return relations_path
```

4.5 Subgraph retrieval

The authors have trained a Language Model such as RoBERTa or BERT on the preprocessing output to perform the subgraph retrieval in the best possible way. In our case, BioLinkBERT was used to make the most of the biomedical domain knowledge to be used as a tokenizer and model in the subgraph retrieval part.

```
load_data_path=${1}
dump_model_path=${2}
device=0

model_name_or_path="michiyasunaga/BioLinkBERT-base"

CUDA_VISIBLE_DEVICES=${device} python3 train.py \
    --model_name_or_path ${model_name_or_path} \
    --train_file ../${load_data_path} \
    --output_dir ../${dump_model_path} \
    --num_train_epochs 50 \
    --per_device_train_batch_size 2 \
    --learning_rate 5e-5 \
    --max_seq_length 32 \
    --metric_for_best_model stsb_spearman \
    --pooler_type cls \
    --overwrite_output_dir \
    --temp 0.05 \
    --do_train

python simcse_to_huggingface.py --path ../${dump_model_path}
```

A script is used to train the model by selecting the appropriate model in this case BioLinkBERT-base which will be loaded by hugging face and the various hyperparameters. The training process and the type of hyperparameters used by the authors were used equally, as BioLinkBERT is also a BERT model. Finally, a script developed by the authors is used, which allows the training output to be converted into a hugging face model to be used later. The input file used is the one generated by negative sampling.

4.5.1 Merge graph

Graph merge is concerned with merging two graphs into a single graph. Each graph comprises a list of nodes and edges (in this case, the edges are represented as pairs of nodes). The two graphs have two different roots, root_l and root_r.

Subsequently, the function intersects the nodes of graph_l with those of graph_r, creating a set of nodes common to the two graphs.

The function then performs a breadth-first search (BFS) for each common node in both graphs to find the ancestors and descendants of each node. The ancestors of a node are defined as the nodes with an edge entering the node, while the descendants are the nodes with an edge out of the node.

Finally, the function adds and returns all nodes found during these visits. The result will then be the union of the nodes of graph_l and graph_r, along with all their common ancestors and descendants.

If there are no nodes in common between the two graphs, the function will return the union of all the nodes of the two graphs, i.e., the sum of the nodes of the left and right graphs.

```
def _reverse_graph(G: Dict[str, List[str]]):
    r_G: Dict[str, List[str]] = dict()
    for u in G:
        for v in G[u]:
            r_G.setdefault(v, []).append(u)
    return r_G


def bfs_graph(G: Dict[str, List[str]], root):
    visited = set()
    currentLevel = [root]
    while currentLevel:
        for v in currentLevel:
            visited.add(v)
        nextLevel = set()
        for v in currentLevel:
            for w in G.get(v, []):
                if w not in visited:
                    nextLevel.add(w)
        currentLevel = nextLevel
    return visited

def merge_graph(graph_l, root_l, graph_r, root_r):
    assert root_l != root_r
    all_nodes = set()
    common_nodes = set(graph_l) & set(graph_r)
    all_nodes |= common_nodes
    reverse_graph_l, reverse_graph_r = _reverse_graph(graph_l), _reverse_graph(graph_r)
    for node in common_nodes:
        ancestors_l = bfs_graph(reverse_graph_l, node)
        ancestors_r = bfs_graph(reverse_graph_r, node)
        descendants_l = bfs_graph(graph_l, node)
        descendants_r = bfs_graph(graph_r, node)
        all_nodes.update(ancestors_l)
        all_nodes.update(ancestors_r)
        all_nodes.update(descendants_l)
        all_nodes.update(descendants_r)
    return all_nodes
```

Chapter 5

Results

As a result, the use of embeddings compared to SPARQL queries has shown a significant improvement in terms of time; in fact, a single SPARQL query takes on average (it varies depending on the query) 30 seconds to find, for example, a path from a topic entity to an answering entity with a single hop, with a single relationship, while in not even a few seconds thanks to the embeddings it is possible to search for all the paths starting from a topic entity with all the relationships to try to get to that answer entity. Using SPARQL queries, it is very difficult, if not impossible, to execute very complex queries where perhaps several k-hops are used; in the case of embedding, this is almost indifferent. Finally, using FAISS, the search performance of the paths in the knowledge graph was further improved, and it was possible to use a threshold for the distance.

Furthermore, BioLinkBert was used as a model on biomedical data to be used in the subgraph retrieval phase, and in addition to this, the use of embeddings allows a simpler and more rapid use of search in multiple k-hop where instead the authors limited themselves to two hops probably because using SPARQL queries took a long time and at the same time there could be the problem of not being able to find a path with only 2 hops.

Conclusions and Future Challenges

In conclusion, the thesis project proved to be very interesting and, at the same time challenging, on a delicate domain such as that of medicine in which research in AI still fails to obtain excellent results as in other fields. Still, thanks to these projects and to much more research will be possible to get there. The implementations made were effective and time-efficient compared to the authors' solution, although the timing of the various experiments and tests took hours, if not days.

The project will continue beyond the thesis to obtain a publication as future developments. Furthermore, the entire MedQA-USMLE dataset was not used but a much smaller part; therefore, the work can be further expanded in terms of complexity and using a multi-k-hop and other different, more complex techniques to get better results.

Ringraziamenti

A conclusione di questo elaborato, desidero menzionare tutte le persone, senza le quali questo lavoro di tesi non esisterebbe nemmeno.

In primis, desidero ringraziare la mia relatrice Antonella Carbonaro, che mi ha consigliato e permesso di intraprendere un percorso di approfondimento in questo campo così interessante e sfidante.

Ringrazio vivamente il mio co-relatore Giacomo Frisoni per avermi accompagnato nella realizzazione di questo progetto e lavoro in questi mesi, per la sua disponibilità, per la sua gentilezza e per la sua grande professionalità.

Ringrazio di cuore chi ha sempre creduto in me lungo questi anni, la mia famiglia, i miei nonni materni/paterni ed anche i miei amici.

I ringraziamenti vanno in parte anche a ChatGPT, grazie a questa tecnologia straordinaria che rappresenta una grande innovazione nel campo dell'intelligenza artificiale, ho potuto sviluppare la mia tesi in modo accurato e sofisticato.

Ultimo ma non meno importante, voglio ringraziare me stesso per averci sempre creduto e non aver mollato in questi anni.

Mattia
17 Marzo 2023

Acknowledgments

After this paper, I would like to mention all the people without which this thesis work would not even exist.

First of all, I would like to thank my supervisor Antonella Carbonaro, who advised and allowed to undertake an in-depth study in such an interesting and challenging field.

I sincerely thank my co-supervisor Giacomo Frisoni for accompanying me in realizing this project and work during these months and for his availability, kindness, and professionalism.

I sincerely thank those who have always believed in me, my family, my maternal/paternal grandparents, and my friends.

Thanks also go in part to ChatGPT, thanks to this technology extraordinary that represents a great innovation in the field of intelligence artificial, I was able to develop my thesis in an accurate and sophisticated way.

Last but not least, I want to thank myself for having us always believing in me and never giving up over the years.

Mattia
March 17, 2023

Bibliography

[1] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding, 2018. cite arxiv:1810.04805Comment: 13 pages.

[2] Jinhyuk Lee, Wonjin Yoon, Sungdong Kim, Donghyeon Kim, Sunkyu Kim, Chan Ho So, and Jaewoo Kang. BioBERT: a pre-trained biomedical language representation model for biomedical text mining. *Bioinformatics*, 36(4):1234–1240, sep 2019.

[3] Amirsina Torfi, Rouzbeh A. Shirvani, Yaser Keneshloo, Nader Tavaf, and Edward A. Fox. Natural language processing advancements by deep learning: A survey. *CoRR*, abs/2003.01200, 2020.

[4] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. In Isabelle Guyon, Ulrike von Luxburg, Samy Bengio, Hanna M. Wallach, Rob Fergus, S. V. N. Vishwanathan, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, pages 5998–6008, 2017.

[5] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *NIPS*, pages 5998–6008, 2017.

[6] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Remi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander Rush. Transformers: State-of-the-art natural language processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 38–45, Online, October 2020. Association for Computational Linguistics.

[7] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. Roberta: A robustly optimized bert pretraining approach, 2019. cite arxiv:1907.11692.

[8] Xin Rong. word2vec parameter learning explained, 2014. cite arxiv:1411.2738.

[9] Antoine Bordes, Nicolas Usunier, Alberto Garcia-Duran, Jason Weston, and Oksana Yakhnenko. Translating embeddings for modeling multi-relational data. In C.J. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K.Q. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 26. Curran Associates, Inc., 2013.

[10] Maximilian Nickel, Lorenzo Rosasco, Tomaso A Poggio, et al. Holographic embeddings of knowledge graphs. In *AAAI*, pages 1955–1961, 2016.

[11] Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. DeepWalk. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, aug 2014.

[12] Michihiro Yasunaga, Hongyu Ren, Antoine Bosselut, Percy Liang, and Jure Leskovec. Qa-gnn: Reasoning with language models and knowledge graphs for question answering, 2021.

[13] Fangyu Liu, Ehsan Shareghi, Zaiqiao Meng, Marco Basaldella, and Nigel Collier. Self-alignment pretraining for biomedical entity representations, 2020.

[14] Xikun Zhang, Antoine Bosselut, Michihiro Yasunaga, Hongyu Ren, Percy Liang, Christopher D. Manning, and Jure Leskovec. Greaselm: Graph reasoning enhanced language models for question answering, 2022.

[15] Michihiro Yasunaga, Antoine Bosselut, Hongyu Ren, Xikun Zhang, Christopher D Manning, Percy Liang, and Jure Leskovec. Deep bidirectional language-knowledge graph pretraining, 2022.

[16] Jing Zhang, Xiaokang Zhang, Jifan Yu, Jian Tang, Jie Tang, Cuiping Li, and Hong Chen. Subgraph retrieval enhanced model for multi-hop knowledge base question answering. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Association for Computational Linguistics, 2022.

[17] Clara Meister, Tim Vieira, and Ryan Cotterell. Best-first beam search. *CoRR*, abs/2007.03909, 2020.

[18] Anonymous SLURM author. Simple linux utility for resource management. pages 1–4, 2011.

[19] OpenAI. We've trained a model called chatgpt which interacts in a conversational way. the dialogue format makes it possible for chatgpt to answer followup questions, admit its mistakes, challenge incorrect premises, and reject inappropriate requests.

[20] Michihiro Yasunaga, Antoine Bosselut, Hongyu Ren, Xikun Zhang, Christopher D Manning, Percy Liang, and Jure Leskovec. Deep bidirectional language-knowledge graph pretraining codebase.

[21] OpenLink Software. Virtuoso.

[22] Ontotext. Ontotext graphdb is a highly efficient and robust graph database with rdf and sparql support.

[23] Graham Klyne and Jeremy J. Carroll. Resource description framework (rdf): Concepts and abstract syntax. W3C Recommendation, 2004.

[24] Antoine Bordes, Nicolas Usunier, Alberto Garcia-Duran, Jason Weston, and Oksana Yakhnenko. Translating embeddings for modeling multi-relational data. In C.J. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K.Q. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 26. Curran Associates, Inc., 2013.

[25] Bishan Yang, Wen-tau Yih, Xiaodong He, Jianfeng Gao, and Li Deng. Embedding entities and relations for learning and inference in knowledge bases, 2014.

[26] David Chang, Ivana Balazevic, Carl Allen, Daniel Chawla, Cynthia Brandt, and Richard Andrew Taylor. Benchmark and best practices for biomedical knowledge graph embeddings, 2020.

[27] Jeff Johnson and Matthijs Douze. Billion-scale similarity search with GPUs. In *Proceedings of the 2017 ACM on International Conference on Multimedia Retrieval*, pages 427–430. ACM, 2017.