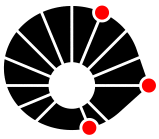

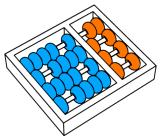

Learning to build SPARQL queries from natural language questions

Mateus de Carvalho Coelho, Julio Cesar dos Reis

Relatório Técnico - IC-PFG-19-33
Projeto Final de Graduação
2019 - Dezembro

UNIVERSIDADE ESTADUAL DE CAMPINAS
INSTITUTO DE COMPUTAÇÃO

Learning to build SPARQL queries from natural language questions

Mateus de Carvalho Coelho, Julio Cesar dos Reis*

December 2019

Abstract

The amount of information available in the web of data using the semantic web standards increased tremendously in the last decade. The Linked Open Data Cloud, for example, has over 1,200 datasets and 16,000 links. Despite that, the access to this data is still difficult because users must have specialised knowledge to query knowledge graphs. Question answering systems have been proposed as an alternative to address this problem in order to provide the benefits of interconnected data further accessible to people. In this approach, systems translate natural language (NL) questions into structured queries. In this work, we study, construct and evaluate a query builder component called MSQG to learn how to build SPARQL queries from NL questions. Our solution explores the combination of distinct sentence encoder to provide better latent sentence representations in the query construction. We evaluate the solution based on Lc-Quad dataset. Obtained results indicate the benefits of our approach in handling several types of questions as input.

1 Introduction

With the popularisation of the semantic web technologies in the last decade, many graph-oriented knowledge bases (KBs), also known as knowledge graphs (KGs), have been shared in the web covering diverse domains, such as geography, biology, medicine and politics. To the best of our knowledge, The Linked Open Data Cloud[1] is the biggest project aggregating linked KGs, having over 1200 datasets and about 16000 links amid them.

In order to access the data in these KGs, one needs to know concepts about graph theory, databases and the specific vocabularies used to encode the meaning of data.

*Instituto de Computação, Universidade Estadual de Campinas, 13081-970 Campinas, SP

[1]Available at `https://lod-cloud.net/`

Additionally, one needs to have some practice in writing queries, since the retrieval of information may be complex due to KB schemes. In other words, only specialised users can access this immense data cloud.

In the direction of addressing this problem, numerous open-domain question answering systems over linked data have been proposed aiming to be an efficient and transparent way of retrieving information to final users. Such systems should be able to answer any type of factual, yes-no and list questions, receiving a natural language (NL) question and returning a set of possible answers collected in KBs endpoints.

The core problem of these systems is translating the user intentions in NL to a structured query (typically SPARQL) that can be executed over knowledge bases. This task is very complex and involves a number of different techniques ranging from classic NL processing techniques, such as POS-tagging, semantic parsing and dependency parsing, to deep neural networks. To easily understand the pipeline of tasks that compose the translation, we can split this complex problem in two parts, namely resource identification and query building.

Although several questions answering systems focusing on different aspects of the translation process have been proposed, little importance was given to modularisation. As result, the creation of new systems is inefficient because they require efforts to redevelop question answering components that were previously proposed, but cannot be easily integrated. In addition, since most systems focus on particular aspects of the QA process, while other parts are simple implementations, they suffer from low score benchmarks that undervalues their contributions [14].

In this work, we propose MSQG, a new query building component for question answering systems. This component can be utilized with Qanary [7], a Java framework for rapidly assembling new QA systems with predefined components. Thus, one of our goals is to make it reusable and easily attachable to new qustion answering systems. Our solution relies on the previous work conducted by Zafar et al. [32] for encoding candidate graphs by using deep neural networks in the query construction.

MSQG is composed by four steps: the first one predicts the NL question type; the second generates possible query graphs; the third one evaluates and sorts them by question similarity; and the last one uses SPARQL templates to construct the query according to the information processed by the other parts. In the third step, our approach tested a double-encoder architecture with Bidirectional Encoder Representations from Transformers (BERT) [9] and Child-Sum Tree-LSTM [33] to provide vectorial latent representations of sentences and queries.

The objective was to verify if using different combinations of sentence and query encoders can lead to improvements in comparison to the previous work. The rational behind testing BERT is that using pre-trained weights may yield better results than training a Child-Sum Tree-LSTM network [25] from scratch. Moreover, Zafar et al. [32] performed dependency parsing on the question to serve as input to the Tree-LSTM. It may introduce error to the system because dependency parsers

are also machine learning models. In our approach, we remove this step and use the question directly as an input to BERT.

Regarding experimentation, we conducted two experiments with the double-encoder architecture. The first one employed two Tree-LSTMs and the second one used Tree-LSTM and BERT. The dataset adopted to evaluate the component effectiveness is Lc-Quad, which is a popular complex set of question and queries pairs. The first experimental setting performed the best, yield an F1-score of 0.87, which is slightly above the results of existing systems evaluated with the same dataset.

The remaining of this document is organised in the following structure: section 2 presents fundamental concepts related to knowledge graphs in addition to question answering systems; section 3 discusses related work; section 4 describes the proposal of a query build component for question answering systems; section 5 explains how to evaluate it; section 6 discusses the obtained results; section 7 provides conclusion remarks.

2 Theoretical Background

In this section we briefly review core concepts of knowledge bases and QA systems. First, in subsection 2.1 we explain how web semantic technologies works, like RDF and SPARQL. Then, in subsection 2.2, the most used processes of each phase of a QA pipeline are presented.

2.1 Structured Knowledge Bases

The structured knowledge bases are those created with a set of standards to model and distribute data in the web, by including: Resource Description Framework (RDF), Resource Description Framework Schema (RDFS), SPARQL Protocol and RDF Query Language (SPARQL) and Ontology Web Language (OWL). These technologies were proposed in the context of the Semantic Web [12]. With the popularity of non-relational databases increasing in the last couple years [1], they have been further used in several domains.

Resource Description Framework (RDF) is a graph-based data model that describes how data should be structured. It has numerous types of serialisation formats. This work explores Turtle. Information in RDF is declared in small facts represented by triples, each one having a subject, a predicate and an object. Each triple expresses some knowledge about the subject, the predicate specifies the kind of information, and the object specifies the information itself. Thus, a dataset in RDF can be seen as a set of triples $T = \{t_i | t_i = (s_i, p_i, o_i)\}$ containing both data and metadata. Moreover, it can also be seen as a graph $G = (V, E)$, where $V = \{s_i, o_i\}$ and $E = \{p_i\}$ [12]. Actually, this formalisation of RDF datasets as a graphs is so important that we

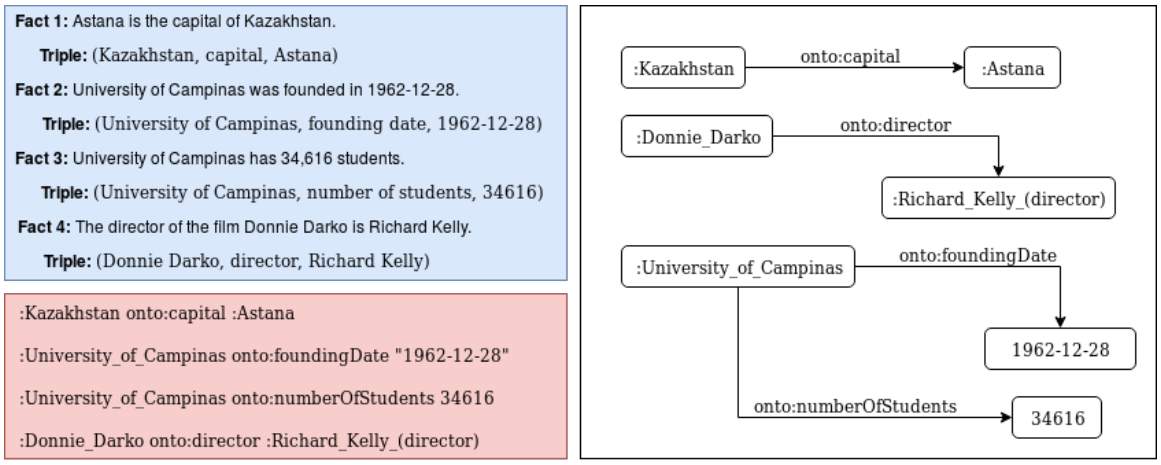

Figure 1: The blue box contains 4 facts modelled in natural language and in triple format. The red box is a list of triples with resources representing the entities described in the facts. The white box shows the graph interpretation of the RDF data.

usually refer to them just as knowledge graphs (KGs). Figure 1 presents examples of how information is modelled in these two views. These examples and the remaining ones in this work explores the DBPedia [16] resources.

As the fact 4 shows, the use of literals to refer to entities in RDF is not recommended because it can be ambiguous. For instance, the word director may have two different meanings according to the context: a film director or a company director. That's why RDF users are encouraged to work with web resources. We can also refer to abstract resources, *i.e.*, resources that does not have something physical associated, but that only exist for the sake of representing entities with Uniform Resource Identifications (URIs) [5]. URIs are simply identifiers that ensure the uniqueness by using the domain of websites. For example, the URI `http://dbpedia.org/ontology/director` links to the film director resource. The red box of the Figure 1 shows how resources are applied. The symbols *:* and *onto:* used there are prefixes that substitute part of the resource identification just to write tuples shortly. They stand for, respectively, `http://dbpedia.org/resource/` and `http://dbpedia.org/ontology/`.

Typically, subjects and predicates are web resources whereas objects can be either a literal or a resource. When resources have the same URI domain, they are from the same namespace or they form a vocabulary. The creation of these groups is important to facilitate the spread of generic resources to be reused by other datasets [12]. Resource Description Framework Schema (RDFS) is an example of standardised vocabulary widely used. It expands the standard RDF vocabulary by including predicates and objects that support the creation of simple ontologies about a knowledge

domain [3]. Ontologies are knowledge representations formalism to describe a knowledge domain in a similar way to the object-oriented model of programming languages. With ontologies, it is possible to model different objects to concepts with classes and a set of properties [2].

The Ontology Web Language (OWL) expands RDFS with the objective of inserting more complex logic into the data. This happens with the use of properties, such as `owl:SymmetricProperty` and `owl:Cardinality`. Based on such logical constraints, reasoners infers information which were not explicit stated in the dataset [8]. The code 1 shows an example of how RDFS and OWL resources are applied to model an use case of students and courses. Here good practices are shown, like for each new resource declare a label and a comment. Also, three things could be inferred from the data: `univ:st87` is a `univ:student`, he takes a `univ:course` and `univ:stats101` has one enrolled student equal to `univ:st87`.

As a way to query information in a knowledge graph, there is SPARQL Protocol and RDF Query Language (SPARQL). A simple query can be seen in the first example of the code 2. In this example, we are looking for the resource that represents University of Campinas, as facts 2 and 3 of the Figure 1. Notice that it contains a variable in the subject of both triples in the section `WHERE`. Since this variable is being selected, this query returns all resources bound to it. Actually, the triple set placed in the section `WHERE` serves as a pattern to be searched in the data. All sets that match this pattern are used by the SPARQL processor to retrieve the results [4]. In this case, the two triples in the red box (Figure 1) corresponding to facts 2 and 3 form an example of such pattern. Thus the SPARQL processor binds the resource `:University_of_Campinas` to `?univ` and returns it.

Additionally, this pattern can be modelled as a graph, so that a query graph $QG = (V', E')$ is a subgraph of the data graph $G = (V, E)$, i.e. $V' \in V \wedge E' \in E$. For example, the query graph of the example in the code 2 looks like the bottom graph in the white box of Figure 1, but with the difference that instead of a node representing the University, we have a node representing a variable.

SPARQL is a powerful language capable of defining complex queries to retrieve data in complex schemes. For example, there are three other types of queries, namely `ASK`, `CONSTRUCT` and `DESCRIBE`. Also it contains several kinds of filters and operators, like `UNION`, `FILTER` and `OPTIONAL`. We explore the second and third queries from the code 2 to explain some of these features by exploring musical knowledge in DBPedia.

The objective of the second query in the code 2 is to show the first post-rock bands. The information returned are the band names and their inception year. First, the query binds `?ent` to resources of the type Band and, in the next two lines, it binds `?name` to the band name. Notice that the query uses the resource `rdfs:label` to get the real name because not always resources have pretty recognisable identifications. It

Listing 1: Use case of RDFS and OWL with an example vocabulary

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX : <http://dbpedia.org/resource/>
PREFIX univ: <http://thispapersvocabulary.com/resources/>

univ:course rdf:type rdfs:Class
univ:course rdfs:label "Course"
univ:course rdfs:comment "A college course"
univ:course owl:sameAs :Course_(education)
univ:stats_course rdfs:subClassOf univ:course
univ:stats_course rdfs:label "Statistics Course"

univ:student rdf:type rdfs:Class
univ:student rdfs:label "Student"
univ:student rdfs:comment "A college student"

univ:takes rdf:type rdf:Property
univ:takes rdfs:label "Takes a course"
univ:takes rdfs:range univ:course
univ:takes rdfs:domain univ:student
univ:enrolled rdf:type rdfs:Property
univ:enrolled owl:inverseOf univ:takes

univ:stats101 rdf:type rdf:stats_course
univ:st87 rdf:type rdfs:Resource
univ:st87 univ:takes univ:stats101
```

Listing 2: Exploring DBPedia with SPARQL examples

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX : <http://dbpedia.org/resource/>
PREFIX onto: <http://dbpedia.org/ontology/>

SELECT ?univ
WHERE {
    ?univ onto:foundingDate "1962-12-28" .
    ?univ onto:numberOfStudents 34616 .
}


SELECT ?name ?year
WHERE {
    ?ent rdf:type onto:Band .
    ?ent rdfs:label ?name .
    FILTER (lang(?name) = "en")
    ?ent onto:genre :Post-rock .
    ?ent onto:activeYearsStartYear ?year .
} ORDER BY ?year LIMIT 10


SELECT ?name ?count
WHERE {
    {
        SELECT ?name (count(?ent) as ?count)
        WHERE {
            ?ent rdf:type onto:Band .
            ?ent onto:genre ?genre .
            ?genre rdfs:label ?name .
            FILTER (lang(?name) = "en")
        } GROUP BY ?name
    }
    FILTER(?count > 1000)
} ORDER BY DESC(?count)
```

filters the labels to get only the English name because there are several representations of the same thing in different languages, like Russian or Japanese. In the next line, the query specifies that only bands of post-rock are required. In the last line of the section `WHERE`, it binds the variable `?year` to the starting year of each band. This last binding is important because in the query's last line, we use it to order the results and show only the first 10 entries. In summary, we shown how to search for specific information and how to better exhibit the output.

The third query's intention is to answer which are the most popular musical genres by number of bands. To do that, we use nested queries, where the result of the inner query is available to the outer one. The inner one starts by binding `?ent` to band resources and `?name` to genre labels. Then, we group the results by genre, making it possible to count how many bands are in each genre. Thus, the results of this query are bindings to genre names and their "popularity". What we do in the outer query is straightforward: filter genres with more than 1000 bands and sort them in a descending order. This query shows how to make the output more human-friendly and how to use group by and nested queries to make more complex extractions.

2.2 Question Parsing and Query structuring

The translation of a natural language (NL) question to a SPARQL query refers to a very difficult problem, since it requires different analysis of the text. According to Diefenbach et al. [10], the whole process can be organized in five steps (cf. Figure 2), by including: question analysis, phrase mapping, disambiguation, query construction and querying distributed knowledge.

Question analysis. In the first phase, the question is analysed to obtain information like the question-type (what, who, where), POS-taggings, named entities and semantic and syntactic representations. This step is essential because it gathers information which serve as input to other algorithms in the subsequent phases, like the syntactic parser that consumes POS-tagging. The named entity recognition (NER) is a particularly special task because it extracts substantives that are converted to resources in the phrase mapping step. Regarding the algorithms to perform NER and POS-tagging, since they are both sequence labelling problems, the use of maximum-entropy Markov models (MEMM), Conditional random fields (CRFs) and Recurrent Neural Networks (RNN) is quite common [15].

It is common the use of syntactic trees, dependency trees and dependency directed acyclic graphs (DAGs) to create syntactic representations of sentences [10]. Syntactic trees are graphs whose root node is a symbol `S` representing a sentence and leaves are words of the sentence. All nodes between the root and the leaves are syntactic structures that represents chunks of text or single words, like noun phrases or nouns. They are built using parsers with context-free grammars, which is a set of productions containing words and tags. For example, the production `NP` $\Rightarrow$ `DT NN` states that a

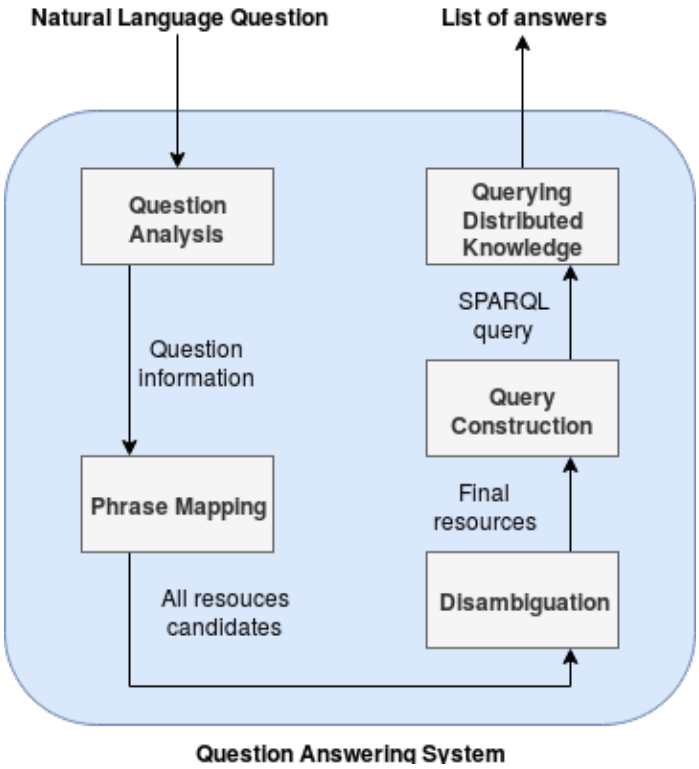

Figure 2: Pipeline of a question answering system. Adapted from [10].

noun phrase is composed by a determiner and a noun. To generate such tree, parsers have to identify what is the sequence of productions that could generate the target phrase. Each production, thus, constitutes a subtree of the syntactic tree [15].

Dependency trees and DAGs are used to represent relations between words. This is useful because there are sometimes hidden relations between words that are far from each other in a sentence. Its nodes are words and edges represents types of syntactic relations. Dependency structures are usually generated using transition-based parsers that sequentially builds the tree or DAG using a set of predefined operations [15].

Phrase mapping. In the second phase, the objective is identifying all possible resources candidates for the phrases recognised in the previous stage. The main source of information about the resources is the RDFS property *label*, which is the standard resource to provides human-readable names. Usually, systems consider three types of phrases that they want to provide candidates: classes, instances and properties. Instances and, specially, properties are the hardest types to find candidates, because sometimes the vocabulary used in the question is not used in the KB schema. This problem is called lexical gap in the literature. Also, it is common to find misspelled words, requiring additional effort to normalise strings in order to correct them. So, to solve this problem, systems utilise various techniques and external data to augment

	What	**is**	**the**	**capital**	**of**	**Kazakhstan**	**?**
Lowercase	what	is	the	capital	of	kazakhstan	?
Stem	what	is	the	capit	of	kazakhstan	?
Lemma	what	be	the	capital	of	kazakhstan	?
POS tags	WP	VBZ	DT	NN	IN	NNP	.
NER tags	O	O	O	O	O	B	O

Table 1: All extracted information about the sentence *What is the capital of Kazakhstan?* in the question analysis and phrase mapping steps.

and normalise the vocabulary [14].

The most common approach is to use string similarity functions, such as Levenshtein distance and Jaccard distance, to select and estimate the probability of the resource being a valid candidate [10]. In order to facilitate this task, words are usually normalised removing uppercase letters and applying stemming or lemmatization [14]. Several systems use auxiliary databases that contains lists of synonyms to expand the vocabulary, such as WordNet [18], for classes and instances, and PATTY [19], for properties. Domain-dependent auxiliary bases can be employed as well, being BOA framework [13] a frequently used example. Given a corpus and a KB, BOA uses regular expressions to extract NL representations of the KB's predicates. Finally, semantic vectors, like Word2Vec [17], can be applied. Given a word, they provide a set of semantic related words learned by a neural network, which is trained in a very large corpus.

Figure 3 and Table 1 show all types of information so far explained that can be extracted from the sentence *What is the capital of Kazakhstan?*. POS-tagging uses the OntoNote 5 tagset [29], NER uses IOB tagset, the syntactic tree uses Penn treebank tagset[2] and dependency tree uses the ClearNLP[3] tagset. To perform stemming and syntactic parsing, this work explores the Python NLP library NLTK [6]. For the remaining tasks we utilised another Python library called Spacy [4].

Disambiguation. The disambiguation stage selects from all phrases candidates the most probable ones. It can be done separately, resource by resource, or jointly, where sets of resources are evaluated together. This part is important because natural language is ambiguous and word meanings can change due to context. For example, in the sentence *What's the best bank?*, bank could refer to a building or to an institution.

There are numerous mathematical techniques to perform this task, but the features employed in these models are very similar [10]. They are: similarity functions, popularity, graph distance and type consistency check. Similarity functions are employed in the phrase mapping step to select candidates, but it is also useful to rank

[2]Available at `http://www.cis.upenn.edu/~treebank/`
[3]Available at `https://emorynlp.github.io/nlp4j/`
[4]Available at `https://spacy.io/`

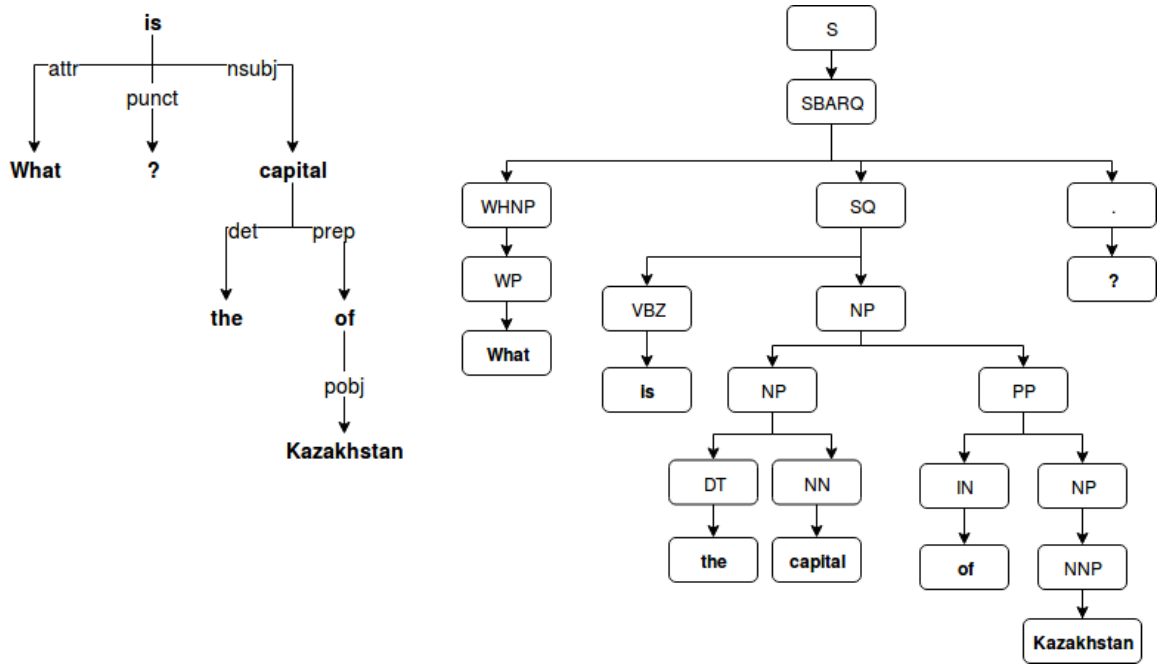

Figure 3: The dependency tree of the sentence *What is the capital of Kazakhstan?* can be seen in the left-hand side of the Figure; the syntactic tree is in the right-hand side.

them since the probability of the resource being the correct one increases as much as the similarity function. Popularity can be modelled, for example, calculating the degree of connectivity between candidates of different resources. The minimum graph distance between two resource is useful as well, since resources connected with a long path probably does not have much context in common. Finally, the type consistency check consists of ensuring that the candidate properties have domain and range types compatible with the subject and object candidate resources. This is capable of discard numerous properties, but has the drawback that not all properties have metadata.

Regarding models to perform phrase disambiguation, one common choice is Hidden Markov Models, where observations are words and the hidden states are resources. Other options are integer linear programs, Markov logic networks and neural networks [10].

Query construction. The query building step is the most important one, since the question interpretation to generate SPARQL queries is done in this stage. It is the component responsible for defining the number of triples, adding filters or operators, like `COUNT` and `ORDER BY`, and deciding which resources belong to the same triples. We organize the approaches in this step in two classes: (1) those that use only information extracted from the user question; and (2) those that use KB structure information as well.

Template-based methods are examples of the first class, where information from the question, such as POS-tags and syntactic and semantic representations, is utilised to choose SPARQL templates. In these methods, templates are previously created as SPARQL queries that may express the user intentions. They contain slots representing natural language expressions (extracted from the question) that will be disambiguated latter [27]. Thus, the core steps within template-based approaches are the creation of expressive templates, their selection and, finally, choose which one is the most appropriated. Another common technique from the first class is to utilise syntactic and dependency trees, as well as semantic parsing to derive the question structure to build the query [10].

Regarding the second class of algorithms, there are few systems that explore them since they are computationally expensive. The objective of these algorithms is, given a set of related resources, explore the KB graph to discover all the possible KG subgraphs that contains them and that could be query graphs. As an example, the system SINA [21] employs a technique of this class that will be explained in details in Section 3.

Querying. The last stage, querying a distributed knowledge, is an optional one, since not all systems are design to work with more than one target KB. The challenge in this part is query information that is distributed, requiring phrase disambiguation with two or more different KB schemes and a query building component capable of making links between KBs. One common approach to solve this task is to use the property `owl:sameAs` to exploit explicit links between resources of connected KBs. However, in case of disjoint bases, usually different queries are made and the results grouped [10].

The question answering pipeline described here follows a logical list of steps. However, not all systems implement it, since there is no standardisation. Very different systems require custom architectures. In fact, some stages are usually combined, such as phrase mapping and phrase disambiguation. In some cases, the first step does not even exist, being performed throughout the entire process. Despite that, using this pipeline is a good practice that can improve the modularisation of QA systems and helps the question answering community providing well defined components.

3 Related Work

Throughout the last years, several approaches have been proposed to solve the problem of structuring a query given a NL question. This section highlights the key QA systems that implement different approaches regarding this task.

In *TBSL* [27], their authors propose to translate the question in two steps. The first one analyzes the question structure to generate SPARQL templates with slots to be filled with resources. The second one searches and disambiguate the resources

that best fit the phrases detected in the first phase. The query building part starts with the question being processed with a POS tagger. Then, based on the tags, the question is parsed, generating syntactic and semantic representations. Representation of domain-independent lexicon mapping expressions is used, composed by expressions such as *all*, *where*, *minimum* and *no*. With these representations, a set of hand-made rules is applied to generate domain-independent templates. Later on in the process, these templates are ranked based on how the disambiguated resources are related in the target KB.

The system *Xser* [31] employs a three-phase process. It first creates a dependency representation of the phrase using a directed acyclic graph. Then disambiguates resources related to the phrases recognised in the question and, finally, builds the final query. One interesting insight is that the query graph structure is generated directly from the DAG connections. The graph is generated using two machine leaning (ML) algorithms: one to capture four types of phrases in the target sentence and one to build the DAG. The first algorithm is a structured perceptron with Viterbi decoding algorithm to solve a sequence labelling problem of identifying entities, relations, categories and variables. The second algorithm is a transition-based parser to estimate the best operation in each step of the graph creation. In both approaches lexical and semantic features are used, in addition to specific ones, such as NER and structure-related features.

SINA [21] is a different system since it receives a full question, but only analyses their keywords. For example, the sentence *What is the side effects of drugs used for Tuberculosis?* would be transformed in a tuple of 4 keywords: (side, effects, drugs, Tuberculosis). So, the system's first step is to clean the user question. Afterwards, it utilises a Hidden Markov Model to disambiguate resources. The emission and transition parameters were defined using bootstrapping. Then, the question building part starts with a set of resources, which can be classes, properties or instances. The system first maps all superclasses of instances and all domain and range types of properties. This information is used to assemble an incomplete query graph (IQG), which can be disconnected if nodes are from different vocabularies. In this IQG, all input classes and instances are instantiated as nodes and all properties as edges. If an input property cannot connect two nodes of the IQG, then new nodes are created so that the property can be created. Next, the system uses an adaptation of the minimal spanning tree algorithm to connect the disconnected components of the IQG. For instance, if nodes in the IQG components are from different vocabularies, then the system searches for `owl:sameAs` links between schemas. Notice that we can have more than one graph in the IQG generation, due to ambiguity, and more than one final query graph, due to various possibilities of links between IQG components.

Zafar et al. [32] introduced SPARQL query generator (SQG), which is a query building component similar to the one used in SINA, but with two differences. First, the query generator receives a list of candidate resources for each entity, class or

property instead of already disambiguated resources. Second, it has a query ranking system, which selects the most probable query graph according to the user intentions. So, the SPARQL graph generator works in two steps: generation of walks in the KG, which corresponds to SPARQL queries, and assortment of candidate walks. The approach of the first step is similar to what is done in SINA. Nevertheless, instead of assembling more than one QG, SQG assembles one subgraph of the KG and, then, generates graph walks that correspond to queries. These candidates are fed into the ranking system. The assumption is that the structure of the NL question should be similar to the candidate walk structure. In order to measure this similarity, cosine function is applied over two vector representations of two trees. The first one is a dependency tree of the NL question and the second one is a tree generated from each candidate walk. These vectors are produced using a Child-Sum Tree-LSTM network, whose architecture is adapted to have trees as inputs.

4 MSQG: A SPARQL query builder

According to Diefenbach et al. [10] and Höffner et al. [14], several question answering systems have been developed since 2010. Despite all the efforts made by researchers to provide well performing QA systems, many of them do not present a clear separation between the components described in section 2.2. Moreover, their inputs, outputs and exchange languages are not standardised, making it difficult to propose new systems reusing older techniques. Thus, our proposal is to build a query constructor component that can be easily integrated in other QA systems. Also, its inputs and outputs are expressed in a standardised exchange format.

Regarding the component itself, we follow the steps of Zafar et al. [32] and propose a modification of SQG, which we call MSQG from now on. We modified the architecture of the neural network that evaluates similarities between question and query graphs. Instead of using one single encoder to make dense representations of both the question and the query, we investigate two separated encoders. Moreover, we evaluate the use of the neural network BERT to generate dense vectors of the questions. In the following, we detail this new component showing how it works and pointing out what changed from SQG. Additionally, subsection 4.4 provides implementation details.

Figure 4 presents a general view of the MSQG showing all its subtasks and how data flows from one subtask to other. It has 2 inputs: a natural language question; a list of tuples containing question utterances and their respective candidate resources. The later is basically the result of the preceding components of a QA pipeline, like a NER or a relation extractor. There is 1 output: the SPARQL query that should resolve the user question. Also, there are four key parts that correspond to black boxes in the figure: question type classifier (cf. subsection 4.1), graphs generator (cf.

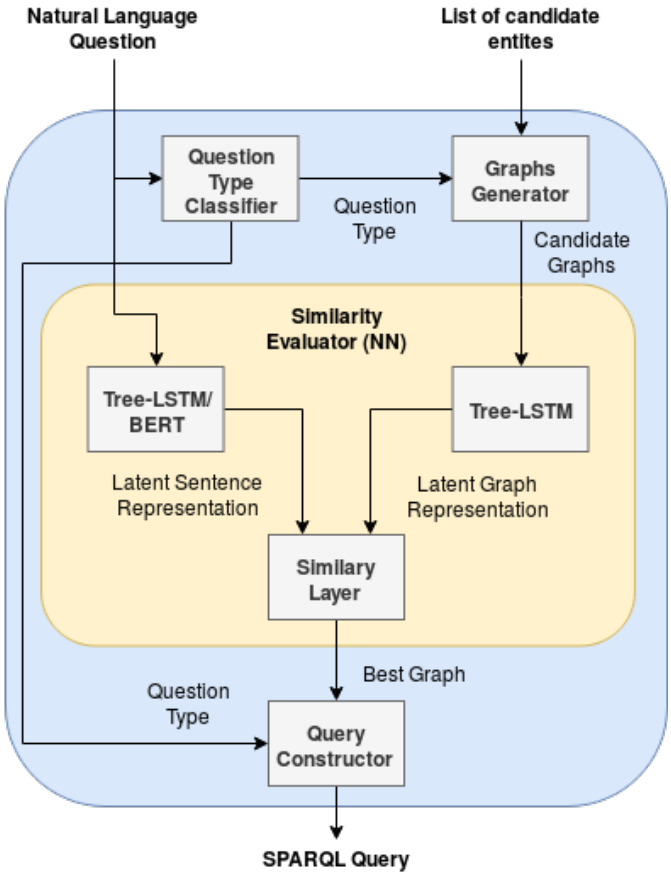

Figure 4: Our proposed query builder component. The blue rectangle represents our component and the yellow rectangle represents the neural network that evaluates question/graph similarity.

subsection 4.2), similarity evaluator (cf. subsection 4.3) and query constructor (cf. subsection 4.3.1).

4.1 Question Type Classifier

The Question Type Classifier is a machine learning classifier that predicts the type of answer a given question is asking for. For instance, the question *How many mammals are in the Chordate phylum?* (taken from the dataset Lc-Quad [26]) requires a count, thus it is a count question. This kind of information is necessary so that the Query Constructor can assemble the final SPARQL query with the right aggregation function. Zafar et al. [32] tested two classifiers, namely logistic regression and naive Bayes, over TF-IDF sentence representations to predict three classes: count, boolean and list questions. We kept this part unchanged in MSQG.

4.2 Graphs Generator

Graphs Generator is a subcomponent that outputs candidate query graphs. The intuition behind this process is that, since we only know the resources our question contains, we cannot determine what is the correct query graph that should compose the SPARQL query. Therefore, we can only enumerate possible graphs and, latter on, rank them according to their similarity to the user question. This part is composed by two steps. The first one is the construction of a subgraph based on the underlying KB containing all input resources as well as all possible connections between them. Then, based on this graph, the second step performs the generation of all valid candidate query graphs. In this context, a candidate query graph is considered valid with respect to a set R of resources if its set of nodes contains all resources of R.

Algorithm 1 (took from algorithm 1 in [32]) shows how to construct such graph, where E and P are the set of entities and predicates given as input, K is knowledge graph and G is the resulting subgraph of K. The first part of the algorithm initialise G with nodes E and add predicates to G if they connect one of the entities E (lines 3 - 8). Note that if a predicate p connects two entities of E, the algorithm only adds p. But, if p connects a node of E and another node e not included in E, then the algorithm adds both p and e. Such nodes e are called unbound nodes. The second part (lines 10 - 22) expands G by including nodes that are two-hop distant from the entities E. Also, it expands G with predicates that connect either some node E or some unbound node.

The generation of candidate query graphs is performed by enumerating all subgraphs of G that are valid. If we have a normal question, for example, the candidates need to have at least one unbound node, since they represent a possible answer node. In MSQG, this entire subcomponent remained unchanged.

4.3 Similarity Evaluator

The role of Similarity Evaluator is to compare the natural language question provided by the user against each candidate query graph. Each query is independently evaluated by this component and it outputs the probability of correctly answering the question. To calculate the level of similarity between them, we first compute the latent vectorial representations of the graph and the sentence. Then, question-graph vector pairs are fed into a similarity function, which can be, for example, the cosine similarity. In our case, it consists of two feed-forward neural network layers in which we feed the multiplication and the absolute difference between the two vectors.

Our solution explores deep neural network by using two different encoders: **Child-Sum Tree-LSTM network** (TLSTM) [25] and **BERT** [9]. The main difference between them is the kind of input they take. Child-Sum Tree-LSTM networks are designed to receive trees, but BERT receives sequential inputs. Also, since TLSTM

Algorithm 1: Construction of subgraph

Input: E, P, K
Output: G

Initialise G as an empty graph
Add E in G as nodes
foreach $e \in E, p \in P$ **do**
  if $(e, p, ?) \in K$ **then**
    Add $(e, p, ?)$ to G
  else if $(?, p, e) \in K$ **then**
    Add $(?, p, e)$ to G
  end
end
foreach (e_1, p, e_2) **do**
  foreach $p' \in P \wedge p \neq p'$ **do**
    if $(e_2, p', ?) \in K$ **then**
      Add $(e_2, p', ?)$ to G
    else if $(?, p', e_2) \in K$ **then**
      Add $(?, p', e_2)$ to G
    else if $(e_1, p', ?) \in K$ **then**
      Add $(e_1, p', ?)$ to G
    else if $(?, p', e_1) \in K$ **then**
      Add $(?, p', e_1)$ to G
    end
  end
end

networks are a kind of Recurrent Neural Network, their computation is done sequentially. BERT, however, uses self-attention units, which are processed in parallel. In our experiments (cf. Section 5), we tested two different approaches. In the first one we used two Tree-LSTMs to generate vectorial representations of the candidate query graph and the sentence. In the second one, we utilized a Tree-LSTMs for the query and BERT for the sentence. The rational behind it is analyzing to which extent providing different representations for sentence and for the query graph may help in choosing the adequate query graph.

In order to learn the best representations for graphs and sentences, each encoder has a set of parameters to be tuned. Stochastic Gradient Descendent is the algorithm utilised to update these parameters and Kullback-Leibler Divergence (equation 1) is the loss function we want to minimise. It measures the similarity between two continuous probability distributions. In this case, the output of the similarity function. When they are equal, it returns 0.

$$D_{KL}(P||Q) = \int P(x) \log \frac{P(x)}{Q(x)} dx \tag{1}$$

Regarding both experiment, one key difference between MSQG and SQG is the double-encoder architecture. In SQG, the TLSTM that provides sentence representations is also employed to generate the query representations, which is not ideal because they are different inputs. The vocabulary of the queries (consisting of predicates and entities) and the sentences (English words) are different, so that the same TLSTM has to learn parameters to parse both "languages". In tasks where two different languages need to be processed, like machine translation, two separated components are used. Certainly, the way that the two "languages" work is not the same, so our hypotheses is that using two neural networks may improve the component effectiveness.

In the second experiment, one important difference between MSQG and SQG is the sentence encoding. Zafar et al. first applies dependency parsing to the sentence. Then, he computes the sentence representation applying the resulting dependency graph in a TLSTM network. Because the dependency parser is also a machine learning algorithm, it may introduce error in the system by providing incorrect dependency graphs. In this case, we use BERT to avoid this dependency parsing process. Another reason to use BERT is that employing transfer learning can improve sentence representations. Therefore, improving the overall component performance. Due to the big size of BERT, using pre-trained parameters avoid spending several hours of training time. Thus, in MSQG we use pre-trained weights in BERT to speed up training time and improve sentence representations.

In the following, we describe how each encoder works.

Child-Sum Tree-LSTM. It is a slightly modified version of the LSTM implementation proposed by Zaremba et al. [33]. Each TLSTM (also known as memory

cells or memory units) represents a node in the query graph and receives the hidden states ($\{h_k|k \in C\}$) and memories ($\{c_k|k \in C\}$) of all its children. Additionally, each unit takes the correspondent resource label of the query graph as input i. The set of children of a given memory unit is C. Figure 5 shows a Child-Sum Tree-LSTM network that could represent a query candidate.

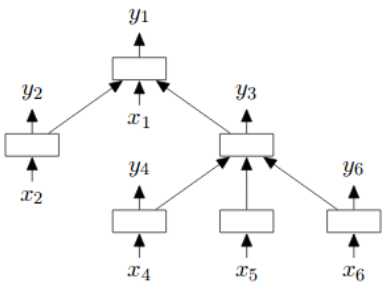

Figure 5: An example of Child-Sum Tree-LSTM network

Roughly, each memory cell outputs part of its internal memory c as the hidden state h (equation 8). This internal memory, in turn, is calculated mixing the internal memory of its children c_k and its own temporary internal memory $\widetilde{c}$ (equation 6). The vectors that control how these operations works are called gates. The output gate o limits how much of the memory is passed to the hidden state (equation 7). The input gate i rules the impact of the temporary internal memory in the final internal memory c (equation 3). Finally, forget gates f_k (one for each children) controls the influence of the children internal memory over the internal memory (equation 5). All the input gates and the temporary internal memory are calculated based on the TLSTM input i and the sum of the children hidden state $\widetilde{h}$. These dependencies make sure that each TLSTM outputs a representation of both the input and the children.

In our experiments, each Tree-LSTM cell receives 300-dimensional input vectors corresponding to DBPedia vocabulary embeddings and outputs 150-dimensional vectors corresponding to internal memory and hidden state.

$$\widetilde{h} = \sum_{k \in C} h_k \tag{2}$$

$$i = \sigma(W^{(i)}x + U^{(i)}\widetilde{h} + b^{(i)}) \tag{3}$$

$$\widetilde{c} = \tanh(W^{(c)}x_j + U^{(c)}\widetilde{h} + b^{(c)}) \tag{4}$$

$$f_k = \sigma(W^{(f)}x + U^{(f)}h_k + b^{(f)}) \tag{5}$$

$$c = i \odot \widetilde{c} + \sum_{k \in C} f_k \odot c_k \tag{6}$$

$$o = \sigma(W^{(o)}x + U^{(o)}\widetilde{h} + b^{(o)}) \tag{7}$$

$$h = o \odot \tanh c \tag{8}$$

Bidirectional Encoder Representations from Transformers. BERT is a deep neural network based on the Transformer [28] architecture. Transformer uses Sequence to Sequence paradigm [24] to perform, for instance, machine translation. But, since BERT is an encoder, it only uses the Transformer encoder. The main purpose of BERT is to be a pre-trained neural network that can be applied to numerous NLP tasks. In order to do so, it was previously trained in 2 tasks: a language model, which can predict what are the correct words in sentences with missing words, and a next sentence predictor, which can learn the relation between two sequential different sentences. With transfer learning, BERT authors improved the state-of-the-art in 11 NLP tasks, such as sentiment analysis and POS-tagging, at time of its publication.

BERT is made of a stack of layers, each one having, sequentially, a multi-head self-attention mechanism and a fully-connected feed-forward network (Figure 6). Additionally, there are normalisation functions and residual connections at the end of each part. Unlike RNNs, whose input words must be sequentially processed, BERT can process word representations of a sentence in parallel. So, the input and output of each layer is a set of word representations.

The inputs of the first encoder layer are word representations created by summing word embeddings and positional encodings. The latter is a vector generated by an analytical function (equation 9) that changes as the sequence increases, where *pos* refers to the word position, d is the number of dimensions the word embedding and i stands for the dimension. This is the only notion of sequence introduced in the network. Then, the inputs of the following layers are the outputs of the previous layers.

$$\begin{aligned} PE_{(pos,2i)} &= \sin \frac{pos}{10000^{\frac{2i}{d}}} \\ PE_{(pos,2i+1)} &= \cos \frac{pos}{10000^{\frac{2i}{d}}} \end{aligned} \tag{9}$$

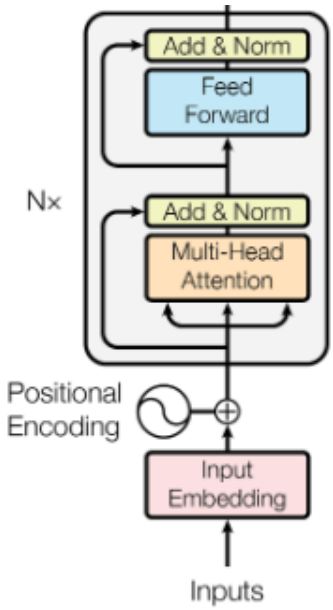

Figure 6: The components of BERT (Transformer encoder). Taken from Figure 1 of [28]

.

A layer execution starts with the calculation of three linear transformations of the inputs, generating three vectors that we call query q, key k and value v. These vectors are fed into the multi-head attention mechanism, where h new representations of q, k and v are created. Then, these h sets of vectors are provided to h different Scaled Dot-Product Attention. Afterwards, the results are concatenated into one matrix and projected to a matrix of lower dimension, according to the equation 10, where W is a matrix of parameters. The left-hand side of Figure 7 show the a diagram of these computations.

$$\text{Multi-Head}(Q, K, V) = \text{concat}(\text{head}_1, \ldots, \text{head}_h)W \tag{10}$$

The Scaled Dot-Product Attention mechanism starts with the dot product of each query by each key. The dot product of a query q_i by a key k_j is the weight of the word j in the representation of i. Thus, the representation of a word i is the sum of vectors v, each one multiplied by the dot products calculated previously. We can perform these calculations in parallel according to the equation 11, where Q, K and V are matrices packaging the query, key and value vectors. The right-hand side of Figure 7 shows the sequence of computations described above.

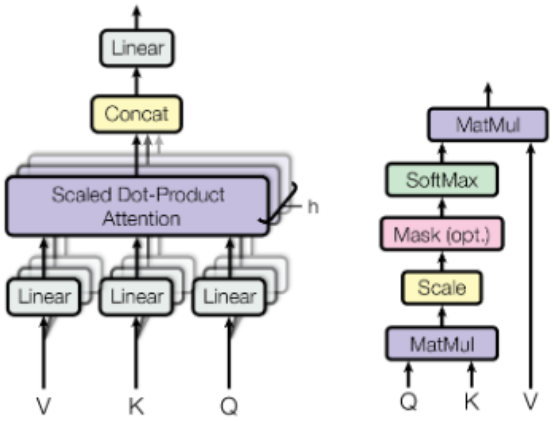

Figure 7: Multi-head self-attention mechanism. Taken from figure 2 of [28]

$$\text{Attention}(Q, K, V) = \text{Softmax}(\frac{QK^T}{\sqrt{d_k}})V \tag{11}$$

Finally, after the calculation of word representations provided by the multi-head self-attention mechanism, the encoder applies each representation separately to the same feed-forward network, composed by two linear transformations and the activation function ReLu (equation 12). The outputs of this step are the outputs of each layer.

$$\text{FFN}(x) = \max(0, xW_1 + b_1)W_2 + b_2 \tag{12}$$

In our experiments, we employed a reduced version of BERT, which contains 12 layers, 12 heads and hidden vectors of 768 dimensions. About 110M parameters had to be adjusted in training phase. Moreover, in order to adapt BERT to SQG architecture, on top of the last BERT layer we added a feed forward layer to convert the 768-dimensional output to a 150-dimensional vector.

4.3.1 Query Constructor

This subcomponent gather the information provided by Similarity Evaluator and Question Type Classifier to actually build the SPARQL query. It sorts the candidate queries by probability and picks the most probable. Also, it uses SPARQL templates, which have some slots to be filled. Each question type has its own template. For example, listing 3 shows the template for a COUNT question. There, `ans_slot` and `graph_slot` are slots for the answer variable and the final query graph. Thus, the

Listing 3: SPARQL template for a COUNT question

```
SELECT COUNT(DISTINCT ans_slot)
WHERE {
    graph_slot
}
```

role of Query Constructor is putting it all together and returns a SPARQL query to answer the user question.

4.4 Implementation aspects

To integrate our component into a full system, we used Qanary [7], a Java framework designed to quickly construct new QA systems from modular components introduced by the community. It follows a microservices architecture in which each component is a service instantiated by an orchestrator, totally independent from the others. Their communication enhances the modularity by using an external triplestore to accumulate information provided by each microservice, such as named entity annotations. The RDF vocabulary employed is called `qa` [22]. It has resources modelling core concepts that QA systems have in common. For example, it provides classes to model questions, annotations, datasets and answers.

As of December 2019, there are 30 components available for testing. 11 perform named entity recognition, 10 implement named entity disambiguation, 5 execute relation recognition, 2 execute class recognition and 2 perform query building.

For the development of MSQG itself, we used Python 3. We made this choice due to its ease of prototyping and because it has a variety of packages to handle structured data and machine learning algorithms. To code, training and test our deep neural network, we used a framework called PyTorch [20]. It is a flexible package to quickly build all sorts of neural networks with support for both CPU and GPU. In addition to that, we employed Transformers [30], a Python package built on top of PyTorch which implements state-of-the-art natural language processing models, including BERT. Source code can be found in our GitHub repositories[5] [6].

5 Evaluation

To test our solution, we did an intrinsic evaluation comparing our query building component with SQG. We utilised a dataset called *Top-5 EARL+correct* for training

[5]MSQG: `https://github.com/mateusccoelho/SQG`

[6]Qanary component: `https://github.com/mateusccoelho/sqg-qanary-component`

and testing. This dataset was constructed by Zafar et al. [32] and is based on Lc-Quad (Large-Scale Complex Question Answering Dataset), which is widely used by the question answering community and it is one of the largest datasets available. It has over 5000 entities and 600 predicates, generated by 38 unique SPARQL templates. Moreover, only about 18% of the questions are simple, *i.e.*, can be translated to a SPARQL query with just one triple. The remaining 82% are natural language queries that need to be translated to structured queries with either more than one triple or ASK/COUNT keywords.

Since we need to evaluate only the query ranking system, *Top-5 EARL+correct* consists of sets of questions, dependency trees, queries and targets. For each question available in Lc-Quad, a dependency tree and candidate queries were produced using Graphs Generator and a set of resources. This set includes the correct ones and the top-5 most probably resources found by EARL [11], which is a commonly used entity and relation linking component. Then, the wrong queries have target equals 0 and the correct ones have target equals 1.

The resulting dataset has 11,257 cases, split in three folds: 70% for training, 20% for validation and 10% for testing. These splits are the same used to measure the performance of SQG's ranking system. Since we have 5,000 correct and 6,257 wrong cases, the dataset has a good balance regarding the class distribution, so that the models can fit well their parameters according to positive and negative instances.

We use 3 metrics in which our results are reported. F1 score (equation 13), that is a geometric mean of two other metrics: precision and recall. Precision tells us the percentage of correct positive answers given by system over all positive answers returned (equation 14). Recall measures the percentage of correct positive answers returned over all the positive answers that should be returned (equation 15). In other words, precision evaluates the impact of false positives and recall evaluates the impact of false negatives.

$$F_1 = 2 \cdot \frac{\text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}} \tag{13}$$

$$\text{precision} = \frac{\text{true positive}}{\text{true positive} + \text{false positive}} \tag{14}$$

$$\text{recall} = \frac{\text{true positive}}{\text{true positive} + \text{false negative}} \tag{15}$$

Table 2 presents the results of the two experiments exposed in section 4 (1 - MSQG with two Tree-LSTM; 2 - MSQG with one Tree-LSTM and BERT). These metrics were evaluated on the test set and the best models from each experiment were chosen

based on the development set F1 score. Note that in all metrics our approaches yield better results than SQG. But to assure that our models really improved the query ranking mechanism, it would be necessary more tests in different training/test samples of the dataset. Since our objective was to compare MSQG and SQG in the same experimental settings, we conclude that MSQG improved SQG's ranking system.

Table 2: Results of SQG and MSQG.

Experiments	Precision	Recall	F1
SQG	0.84	0.84	0.84
MSQG: two Tree-LSTM	0.87	0.87	0.87
MSQG: one Tree-LSTM and BERT	0.87	0.86	0.86

6 Discussion

We found that experimenting with different architectures slightly improved SQG's ranking system (cf. Table 2). We can affirm that using two encoders, one for each "language", is a valid research path that if more explored can lead to further improvements. Also, the use of state-of-the-art natural language understanding (NLU) models, such as BERT, is possible and can result in competitive metrics. But, even though it is possible, the second experiment did not lead to much better results than our first approach.

To enrich our discussion, we present two questions from the test set that were correctly and wrongly answered by MSQG with two LSTMs. The candidate queries generated from subgraphs of the knowledge base are below each question. `?u_0` is always the entity that solves the question and `#ent` is a placeholder for entities removed from the pairs.

1. Who owns the tunnels operated by the #ent?

- ```
  ?u_0 operatedBy #ent .
  ?u_0 type RoadTunnel .
  ?u_0 owner ?u_1
  ```

  Score: 0.9273

- ```
  ?u_0 operatedBy Massachusetts_Department_of_Transportation .
  ?u_0 type RoadTunnel .
  ?u_0 owner Massachusetts
  ```

  Score: 0.3104

- ```
  ?u_0 operatedBy ?u_1 .
  ?u_0 type RoadTunnel .
  ?u_0 owner Massachusetts
  ```
  Score: 0.3313

2. To which company is the service #ent associated with?

- ```
  ?u_1 company ?u_0 .
  ?u_0 services #ent
  ```
  Score: 0.9238

- ```
  ?u_1 company ?u_0 .
  ?u_0 services Nintendo_eShop
  ```
  Score: 0.5312

- ```
  ?u_0 type Company .
  Nintendo services ?u_0
  ```
  Score: 0.2490

- ```
  Love_Tester_(Nintendo) company ?u_0 .
  ?u_0 services ?u_1
  ```
  Score: 0.5276

- ```
  Love_Tester_(Nintendo) company ?u_0 .
  ?u_0 services Nintendo_eShop
  ```
  Score: 0.5162

Notice that in these two cases, queries with different structures were generated based on the same question. Since each query is independently evaluated in the ranking system, each one has a score that is the probability of correctly answering the question. Bold scores are the correct ones, so MSQG picked the correct query to answer the first question, but failed in the second example.

Regarding the training phase, one aspect to note is that in both approaches the weights obtained after the first epoch of training were the ones that performed the best. Given the size of the training split (about 7,000 pairs), we conclude that our models started to overfitting at this point, as the F1 score over the development set decreased after each epoch. To solve this problem, we could add regularization mechanisms, such as Dropout [23] to the Tree-LSTM models. Also, BERT, even in its smaller version, is a huge neural network with over 100M parameters to be tuned. One epoch took couple of hours to be adjusted in a Google Colab server with GPU.

So our premise that using BERT would speed up the training phase turns out to be wrong.

We list some advantages and disadvantages of MSQG. First, MSQG showed to be a powerful component capable of recognizing hard connections between the input entities and predicates. It happens thankful to its capacity of exploring the target knowledge base to extract all subgraphs containing these inputs. Also, in order to decide which subgraph fits the best the input question, it employed a deep neural network capable of extracting features based on the input words and their context. However, its performance in practice is very dependent of the inputs given by previous QA components. For instance, if a NER gives entities that are not connected in the KB or that are very distant from each other, MSQG will not be able to extract any subgraph.

In addition, MSQG is not a component design to be used in open domain QA systems due to its limited vocabulary. It was trained in a fixed query vocabulary, which means that predicates that were not seen in the training phase will not have learned embeddings. This represents a big disadvantage because if we wanted to expand the model vocabulary, we would need to create a new dataset with questions containing this new set of entities and predicates. One possible solution for this problem would be to use pre-trained generic graph embeddings of the knowledge base. This way we could adjust the embeddings to our task as we did when adjusting BERT's word embeddings.

7 Conclusion

Question answering systems based on natural language questions can be valuable to people benefit from encoded knowledge in RDF knowledge bases. However, the automatic generation of SPARQL queries from the natural language questions is a very difficult task. In this work, we proposed a new modular query building component capable of being attached to the end of a QA pipeline. In order to leverage reuse, the component was designed to be accessible via Qanary, a QA framework built in Java. Our developed component named MSQG was focused on advancing a query ranking system. Our solution reused remaining parts from existing state-of-the-art methods and tools. The conducted experiments assessed a double-encoder architecture in which questions and queries representations were created separately. The first one employed two Tree-LSTMs and the second one utilized one Tree-LSTM and BERT. The former experiment performed the best, with an F1 score of 0.87, which surpassed the results of existing system evaluated with the same dataset. Future work will involve testing new encoder architectures and pre-trained graph embeddings to improve even further the developed component.

References

[1] Db-engines ranking - trend popularity. Available at `https://db-engines.com/en/ranking_trend`. Accessed: 2019-06-01.

[2] Ontology (information science). Available at `https://en.wikipedia.org/wiki/Ontology_(information_science)`. Accessed: 2019-06-01.

[3] Rdf schema. Available at `https://en.wikipedia.org/wiki/RDF_Schema`. Accessed: 2019-05-19.

[4] Sparql). Available at `https://en.wikipedia.org/wiki/SPARQL`. Accessed: 2019-05-22.

[5] Web resource. Available at `https://en.wikipedia.org/wiki/Web_resource`. Accessed: 2019-05-15.

[6] Steven Bird, Ewan Klein, and Edward Loper. *Natural Language Processing with Python*. O'Reilly, 1st edition, 2009.

[7] Andreas Both, Dennis Diefenbach, Kuldeep Singh, Saeedeh Shekarpour, Didier Cherix, and Christoph Lange. Qanary – a methodology for vocabulary-driven open question answering systems. 05 2016.

[8] Owen Conlan and Athanasios Staikopoulos. Introduction to web ontology language (owl). University lecture available at `https://www.scss.tcd.ie/Owen.Conlan/CS7063/06\%20Introduction\%20to\%20OWL\%20(1\%20Lecture).ppt.pdf`, 2015. Accessed: 2019-05-22.

[9] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. *CoRR*, abs/1810.04805, 2018.

[10] Dennis Diefenbach, Vanessa Lopez, Kamal Singh, and Pierre Maret. Core techniques of question answering systems over knowledge bases: a survey. *Knowledge and Information Systems*, 55(3):529–569, Jun 2018.

[11] Mohnish Dubey, Debayan Banerjee, Debanjan Chaudhuri, and Jens Lehmann. EARL: joint entity and relation linking for question answering over knowledge graphs. *CoRR*, abs/1801.03825, 2018.

[12] Bob DuCharm. *Learning SPARQL*. O'Reilly, 2nd edition, 2013.

[13] Daniel Gerber and Axel-Cyrille Ngonga Ngomo. Extracting multilingual natural-language patterns for rdf predicates. In Annette ten Teije, Johanna Völker, Siegfried Handschuh, Heiner Stuckenschmidt, Mathieu d'Acquin, Andriy Nikolov, Nathalie Aussenac-Gilles, and Nathalie Hernandez, editors, *Knowledge Engineering and Knowledge Management*, pages 87–96, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[14] Konrad Höffner, Sebastian Walter, Edgard Marx, Ricardo Usbeck, Jens Lehmann, and Axel-Cyrille Ngonga Ngomo. Survey on challenges of question answering in the semantic web. *Semantic Web*, 8, 11 2016.

[15] Daniel Jurafsky and James H. Martin. *Speech and Language Processing*. 2018. 3rd edition draft.

[16] Pablo N. Mendes, Max Jakob, Andrés García-Silva, and Christian Bizer. Dbpedia spotlight: Shedding light on the web of documents. In *Proceedings of the 7th International Conference on Semantic Systems*, I-Semantics '11, pages 1–8, New York, NY, USA, 2011. ACM.

[17] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg Corrado, and Jeffrey Dean. Distributed representations of words and phrases and their compositionality. In *Proceedings of the 26th International Conference on Neural Information Processing Systems - Volume 2*, NIPS'13, pages 3111–3119, USA, 2013. Curran Associates Inc.

[18] George A. Miller. Wordnet: A lexical database for english. *Commun. ACM*, 38(11):39–41, November 1995.

[19] Ndapandula Nakashole, Gerhard Weikum, and Fabian Suchanek. PATTY: A taxonomy of relational patterns with semantic types. In *Proceedings of the 2012 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning*, pages 1135–1145, Jeju Island, Korea, July 2012. Association for Computational Linguistics.

[20] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in PyTorch. In *NIPS Autodiff Workshop*, 2017.

[21] Saeedeh Shekarpour, Edgard Marx, Axel-Cyrille Ngonga Ngomo, and Sören Auer. Sina: Semantic interpretation of user queries for question answering on interlinked data. *Journal of Web Semantics First Look.*, 01 2015.

[22] K. Singh, A. Both, D. Diefenbach, and S. Shekarpour. Towards a message-driven vocabulary for promoting the interoperability of question answering systems. In

2016 IEEE Tenth International Conference on Semantic Computing (ICSC), pages 386–389, Feb 2016.

[23] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15:1929–1958, 2014.

[24] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. Sequence to sequence learning with neural networks. *CoRR*, abs/1409.3215, 2014.

[25] Kai Sheng Tai, Richard Socher, and Christopher D. Manning. Improved semantic representations from tree-structured long short-term memory networks. *CoRR*, abs/1503.00075, 2015.

[26] Priyansh Trivedi, Gaurav Maheshwari, Mohnish Dubey, and Jens Lehmann. Lc-quad: A corpus for complex question answering over knowledge graphs. In *International Semantic Web Conference*, pages 210–218. Springer, 2017.

[27] Christina Unger, Lorenz Bühmann, Jens Lehmann, Axel-Cyrille Ngonga Ngomo, Daniel Gerber, and Philipp Cimiano. Template-based question answering over rdf data. *WWW'12 - Proceedings of the 21st Annual Conference on World Wide Web*, 04 2012.

[28] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. *CoRR*, abs/1706.03762, 2017.

[29] Ralph Weischedel, Martha Palmer, Mitchell Marcus, Eduard Hovy, Sameer Pradhan, Lance Ramshaw, Nianwen Xue, Ann Taylor, Jeff Kaufman, Michelle Franchini, Mohammed El-Bachouti, Robert Belvin, and Ann Houston. Ontonotes release 5.0 ldc2013t19. 2013.

[30] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, R'emi Louf, Morgan Funtowicz, and Jamie Brew. Huggingface's transformers: State-of-the-art natural language processing. *ArXiv*, abs/1910.03771, 2019.

[31] Kun Xu, Sheng Zhang, Yansong Feng, and Dongyan Zhao. Answering natural language questions via phrasal semantic parsing. In Chengqing Zong, Jian-Yun Nie, Dongyan Zhao, and Yansong Feng, editors, *Natural Language Processing and Chinese Computing*, pages 333–344, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.

[32] Hamid Zafar, Giulio Napolitano, and Jens Lehmann. Formal query generation for question answering over knowledge bases. In Aldo Gangemi, Roberto Navigli, Maria-Esther Vidal, Pascal Hitzler, Raphaël Troncy, Laura Hollink, Anna Tordai, and Mehwish Alam, editors, *The Semantic Web*, pages 714–728, Cham, 2018. Springer International Publishing.

[33] Wojciech Zaremba and Ilya Sutskever. Learning to execute. *CoRR*, abs/1410.4615, 2014.