Complex Query Augmentation for Question Answering Over Knowledge Graphs

Abdelrahman Abdelkawi[1], Hamid Zafar[2],
Maria Maleshkova[2], and Jens Lehmann[2,3]

[1] Computer Science Institute, RWTH Aachen University, Germany
abdelrahman.abdelkawi@rwth-aachen.de
[2] Computer Science Institute, University of Bonn, Germany
{hzafarta,maleshkova,jens.lehmann}@cs.uni-bonn.de
[3] Fraunhofer IAIS, Dresden, Germany,
jens.lehmann@iais.fraunhofer.de

Abstract. Question answering systems have often a pipeline architecture that consists of multiple components. A key component in the pipeline is the query generator, which aims to generate a formal query that corresponds to the input natural language question. Even if the linked entities and relations to an underlying knowledge graph are given, finding the corresponding query that captures the true intention of the input question still remains a challenging task, due to the complexity of sentence structure or the features that need to be extracted. In this work, we focus on the query generation component and introduce techniques to support a wider range of questions that are currently less represented in the community of question answering.

Keywords: Question Answering · Knowledge Graphs · Query Augmentation

1 Introduction

Question answering (QA) has been an active field of research for many decades in different areas such as information retrieval, natural language processing and machine learning. It provides users with a convenient interface to ask their question in a natural way.

As semantic web technologies developed in recent years, vast sources of structured data became available, for instance, domain-specific Knowledge Graphs (KGs) (such as UMLS [17], GeoNames [30], WordNet [20], etc.) and open-domain KGs (e.g DBpedia [16], Freebase [5], etc.). Given these well-structured sources of information, QA over KG is able to provide concise answers not only to simple but also to more complicated questions, including the traversal of multiple relevant (triple) patterns in the KG.

Often *Semantic parsing* approaches are employed to build QA over KG, in which multiple components can be orchestrated in a pipeline architecture. This pipeline transfers the input question into a formal query representation of the

question, which captures the intention of the user. As opposed to end-to-end methods [18] that work as a black-box, semantic parsing approaches provide a modular solution, which enables researchers to find out the reasons for the success and failure cases by investigating the components individually. Hence, it is also easier to improve and re-use the existing work as well. Furthermore, end-to-end methods, in general, are not applicable in cases where the training dataset is not large enough.

Semantic parsing approaches mostly consist of five components that are responsible for the following tasks [25]: *Shallow parsing* (a.k.a chunking), *Entity linking*, *Relation linking*, *Query generation* and *Ranking*. The first component analyzes the input question in order to partition it into entity and relation spans. These spans are the main clues for the entity and relation linking components to find the corresponding items in the knowledge graph. Given all the linked items, the query generator searches for the possible valid combinations of the linked items, which later would be compared to the input question in order to arrange them according to their similarity to the intention of the user.

Although these components are necessary to build a QA system, researchers mostly focus on earlier steps and limited attention is paid to the query generation and ranking modules, due to the fact that most of Question/Answering (Q/A) datasets contain mostly questions with a simple corresponding formal query. Therefore, most of the existing approaches fail to correctly comprehend the challenging questions, in which the query generation task is more demanding. The performance of the query generator depends on the complexity of the input question and the distinct features from the underlying formal query language, which should be supported by the query generator. Nevertheless, given the fact that it is burdensome to define a concrete metric to measure the level of complexity of a natural language question, we establish the complexity of a question based on two features from its corresponding formal query: Type of the formal query (enumerated in Table 1) and the number of linked items used in formal query, where the queries that use more linked items, correspond to more complex questions.

In the simplest case, a question can be answered using one entity and one relation from the underlying knowledge graph. In this case, the number of candidate formal queries are limited. For instance, consider the question `Who are the children of Barak Obama`, where the only entity is `Barak Obama` and the relation is `children`. In this example, there are just two possible formal queries that can be built: `SELECT ?c WHERE{?c dbo:Children dbr:Barak_Obama}` and `SELECT ?c WHERE{dbr:Barak_Obama dbp:children ?c}` where the first one is the latter interpretation of the question. However, as the number of linked items increases, the search space might explode and it would be challenging to explore the search space in order to find the candidate queries. SimpleQuestions [7] and WebQuestions [3] are the de facto standard Q/A datasets based on Freebase [5] as they are used in many of QA over Freebase systems [4,6,31,32]. All the questions in SimpleQuestions and more than 80% of WebQuestions can be answered using a single relation in the underlying knowledge graph. Furthermore, there are only

Table 1. Various types of Queries and their corresponding sample question

Type	Description/Example/SPARQL
List	The question is a factoid question (single or multiple relations) Example: Who are the children of Barack Obama? SELECT ?child where { dbr:Barack_Obama dbp:children ?child}
Boolean	The question is a yes or no question Example: Is Paris the capital of France? ASK WHERE { dbr:France dbo:capital dbr:Paris ; rdf:type dbo:Place }
Count	The intention of the question is to count the number of the possible results Example: How many cities are in Germany? SELECT COUNT(?city) WHERE {?city dbo:country dbr:Germany ; rdf:type dbo:City}
Ordinal	The question requires ordering of the results over a certain criteria Example: What is the most populated city in Italy? SELECT ?city WHERE{?city dbo:country dbr:Italy ; dbo:populationTotal ?population ; rdf:type dbo:City } ORDER BY DESC(?population) LIMIT1
Filter	The question requires the results to be restricted using a certain criteria Example: List all cities with more than a million population in Egypt? SELECT ?city WHERE{?city dbo:country dbr:Egypt ; dbo:populationTotal ?p ; rdf:type dbo:City. FILTER (?p > 1000000)}

3% *Ordinal* questions and no *Boolean* question in WebQuestions. Consequently, most of the introduced approaches mainly focus on the first type (see *List* in Table 1). However, LC-QuAD dataset filled the gap to some extent by providing 7% *Boolean* and 12% *Count* questions out of a total 5,000 questions. As a result, more researchers concentrate on these two types as well [8,19,33]. Yet, very limited effort has been spent on the last two categories, in spite of the fact that the number of *Ordinal*/*Filter* questions is increasing in QA datasets. Given that there are already advanced approaches to deal with the first three groups, we aim to enhance an existing query generator in order to not reinvent the wheel and benefit from the existing infrastructures. Among others, SQG [33] reports significantly better accuracy in comparison to other query generator components on various datasets. Thus, in this work we concentrate on extending SQG to support more complex queries, namely *Ordinal* and *Filter*.

The remainder of the paper is structured as follows: Section 2 briefly discusses the related works on various techniques, which have been employed to support complex features such as *Ordinal* and *Filter*. We then introduce the overall architecture as well as the details of the approach in Section 3 and present the evaluation results in Section 4. Section 5 concludes our findings.

2 Related Work

The main-stream question answering systems over knowledge graphs can be divided into two categories: Semantic parsing methods and End-to-end approaches. 62 semantic parsing question answering systems from 2010 till 2015 are analyzed Hoffner et al. [14]. They discuss the main challenges in question answering systems as well as the common solutions. Chakraborty et al. [22] provide a more recent overview of neural networks based question answering.

Although end-to-end QA system achieved state-of-the-art results, they mostly focus on simple/compound question, and either neglect other types [22] or use simple pattern matching techniques to address *Ordinal* or *Filter* types. Hence, we mostly study semantic parsing methods.

Walter et al. [29] introduce BELA - a QA system that consists of a 5-step pipeline: question parsing, template generation, string similarity computation, synonym-finding, and semantic similarity computation. The main idea of the system is that the system decides, which steps should be applied, depending on the complexity of the input question. The system is evaluated on QALD-2[4], however it is not capable of answering questions that require sorting or filtering of the results.

Unger et al. [28] propose TBSL, a QA system that parses the input question to extract syntactic information from the question using predefined lexicons, then it uses this information to generate a logical expression similar to the question. Using this expression, the system chooses the candidate query templates. Finally, TBNSL attempts to fill in the empty slots in the candidate templates through the entities and relations mentioned in the given question. Moreover, it uses ranking techniques to select the best candidate query. Considering that the query templates are manually created based on the dataset at hand, it is considered to be over-fitted for the dataset.

CASIA [26] and SINA [24] are two more examples of QA systems based on a pipeline architecture. The pipeline of such systems includes tasks such as question processing, entity and relation recognition and disambiguation, and SPARQL query generation. These systems are benchmarked on the QALD-3[5] challenge dataset. Similarly, there is no support for questions where complicated features such as *Filter*, *Ordinal*, etc. are required.

Hakimov et al. [12] develop a QA system that uses a semantic approach based on Zettlemoyer et al. [34]. They investigate the use of handcrafted lexicons to minimize the lexical gap between the vocabulary used in natural language questions and the one of the training data. The system is benchmarked on QALD-4[6], however, the systems has no support for *Ordinal* or *Filter* questions.

POMELO [13] is another pipelined QA system, which resembles the architecture of CASIA [26] and SINA [24]. In addition to the pre-processing steps in the pipeline, POMELO scans the question for certain terms such as numbers, `mean`,

[4] `http://qald.aksw.org/index.php?x=task1&q=2`
[5] `http://qald.aksw.org/index.php?x=task1&q=3`
[6] `http://qald.aksw.org/index.php?x=task1&q=4`

`higher`, etc. in order to construct the SPARQL query. This step helps POMELO to support more query types than CASIA [26] and SINA [24]. However, it fails to handle compound questions. Moreover, its support for *Filter* and *Ordinal* types is limited due to the fact that it is based on a hand-crafted list of patterns.

The AskNow approach as described in [11] is a QA framework by M. Dubey et al. that takes a natural language question as its input, then transforms the question into an intermediate logical form called *Normalized Query Structure*, which later will be changed into a SPARQL query. AskNow defines three types of queries: Simple, complex and compound. As a result, it is able to support compound questions. Nevertheless, the support for *Filter* and *Ordinal* is limited to the pre-defined patterns. Much like the system proposed by Unger et al. [28], Abujabal et al. [1] use a similar approach with the main difference that the system is able to learn SPARQL templates from question-query pairs. Given a question, it tries to match the question to an empty candidate template(s) that corresponds to the given question. In addition, it benefits from ranking methodologies for selecting the best candidate queries.

The aforementioned QA systems are either based on templates/patterns or use and ad-hoc methods to support complex queries. While, we base our solution on extending a well structured, modular, standalone SPARQL query generator that is capable of generating target SPARQL queries for input questions, provided the entities and relations mentioned in the question.

3 Approach

Given a question in natural language and the correct linked items (entities and relations), SQG [33] goes into the details of generating a SPARQL query that corresponds to the input question. By using this generated SPARQL query and augmenting it with necessary constraints, we are able to obtain a SPARQL query that supports new, previously unsupported, types.

In order to extend SQG [33] to support the two new types, we model these types as extra constraints that need to be applied on the list of all possible answers. For the *Ordinal* class, to get the correct answer for the example question **Q1:**`"What is the most populated city in Italy?"`, we first need to get a list of all the cities in Italy, then sort them in descending order with respect to the population of each city and then return the top city as the most populated city in Italy. The same idea applies to the type *Filter*, where the list of possible answers should conform to a certain constraint. For example, given the question **Q2:**`"What are the cities with more than a million population in Egypt?"`, we need to get all the cities in Egypt and only return those with the population more than a million as the answer. This unified view of modeling the new types as constraints enables us to extend SQG by adding an extra layer over the existing architecture.

To support the aforementioned types, we divide the overall task into three sub-tasks. First, we need to classify the given questions in order to recognize those questions that belong to the new types. Second, we parse the given question

to extract special keywords that would help us to select a KG property, which would act as the constraint for the intended SPARQL query. The last task is to set any parameters needed for the SPARQL query in order to capture the intention of the given question.

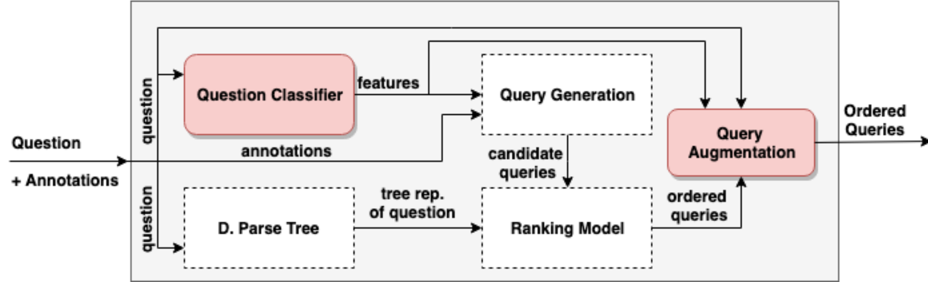

Fig. 1. Proposed ExSQG Architecture. Components highlighted in red are modified/added components

Figure 1 shows the architecture of ExSQG. It extends SQG [33] with two new components – *Question Classifier* and *Query Augmentation*. The new question classifier replaces the original question classifier from the SQG [33] as it does not support the new question types. The original question classifier is built as a flat classifier using Naive Bayes and SVM and supports only List, Boolean and Count questions.

In SQG [33], the ranking model was the last step in the query generation pipeline. However, in the ExSQG architecture, the query augmentation component resides at the end of the pipeline. The augmentation component is responsible for complementing the SPARQL query, which is selected by the ranking model, by adding the necessary constraints and parameters in order to generate the final query that corresponds to the input question.

Intuitively each question is of List, Boolean or Count type. However it may belong either to *Ordinal* or *Filter*, or both. We call the first three categories PRIMARY CLASSES and *Ordinal* and *Filter* secondary classes. Accordingly, we build a hierarchical question classifier, which consists of a multi-class classifier for primary classes and a binary classifier for each of the secondary classes. Figure 2 shows the architecture of the *Questions Classifier*. When a question is passed through the classifier module, it is first classified by the primary classifier to find out its primary class. Given the primary class, it passes through all the secondary binary classifiers to check if the question belongs to one or more of the secondary classes. As shown in Figure 3, both **Q1** and **Q2** are identified as *List* by the primary classifier, however, **Q1** is classified as *Ordinal* as the secondary class, while the second class of **Q2** is established as *Filter*.

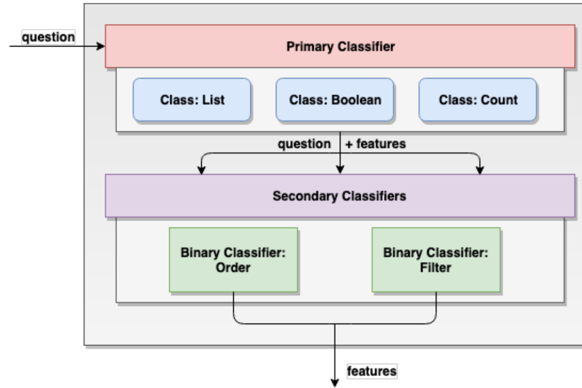

Fig. 2. Architecture of the Hierarchical Question Classifier

After the question is classified, it passes through the rest of the pipeline. If the question is classified to have only a primary class and no further secondary classes, then the query is returned by the ranking model as the result of SQG [33]. On the other hand, when the question is classified to be one of the secondary classes, it passes through the query augmentation component with its corresponding SPARQL query chosen by the ranking model.

The first task of the query augmentation is to select a KG property that acts as the constraint in the SPARQL query. First, the natural language question is cleaned by removing stop words and any entity mentions. The result of this process is called a *base-form* and is used in the *Parameters Settings* step. By parsing the base-form according to the class of the question provided by the question classifier, we are able to further clean it, which would result in having single or multiple words. This sequence of words is called *keyword* or *keywords*. For example, the base-form for the **Q1** is `"most populated city"` and the keywords are `"most populated"`.

In parallel with the keyword extraction task, the SPARQL query provided by the ranking model is used to capture the list of KG relations in the one-hop distance of the subgraph containing the answer. Empirically, by analyzing Filter and Order questions and their corresponding gold SPARQL queries. We found that the relations used as constraints are always in the one-hop space distance from the subgraph that contains the answer. Thus, we operate under the assumption that the KG property that acts as the constraint is contained within this list. These extracted relations are then filtered retaining only those, which are comparable (e.g. Numbers, Dates, etc.). For instance, the candidate relations for **Q1** are `dbo:areaTotal, dbo:Country, dbo:populationTotal`, etc. .

In order to select the correct KG relation from the list of possible relations, we capture the semantic closeness of the keywords and each of those relations by computing the cosine similarity between their word embeddings. The KG relation and keywords, which form the closest pair, are selected as the final KG

relation, which acts as the constraint in the final SPARQL query. Note that since both the keywords and KG relations might consist of more than a single word, we use Word Mover Distance [15] to measure the similarity between the keywords and the KG relations. For example, from the list of candidate relations for **Q1**, `dbo:populationTotal` is the most similar one in comparison to the keywords `most populated`. It's worth noting that before checking the similarity between the KG relations and the keywords. The KG relation is transformed into a correct English form, from `populationTotal` into `population total`. This is done by simply splitting the KG relation at each capital letter, since they are always written in a camel case form.

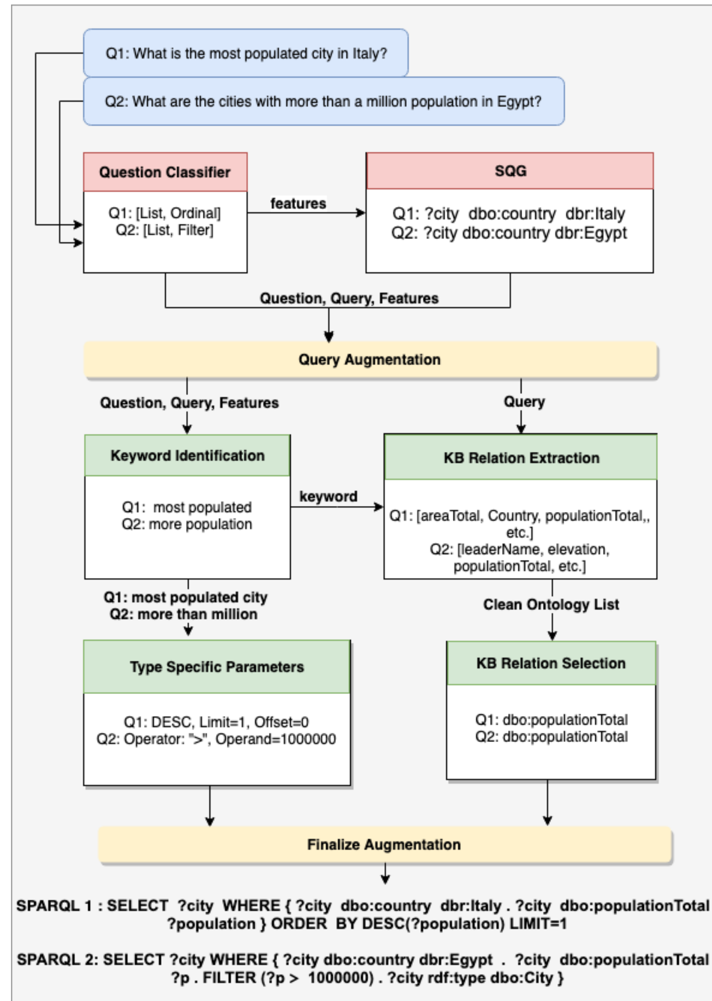

Fig. 3. ExSQG pipeline for Ordinal and Filter examples

The final step is to set any parameters for the given query. This parameter setting depends on the type of the query. For queries of type *Ordinal*, there are three parameters to be considered; the direction of sort, offset and limit. In order to set the direction of sort, we train a classifier that predicts the sorting direction given the keywords. On the other hand, the offset is set by parsing the base-form provided by the components responsible for the keyword extraction. If the base-form contains an ordinal mention (e.g first, second, third, etc.), it is used to set the offset in the SPARQL query. Otherwise, the offset is set to zero. The last parameter in the *Ordinal* queries is the limit. To set the limit of the query, we use Part Of Speech(POS) tags to check if the keyword or keywords refer to a singular or plural noun to set the limit accordingly. For our running example question **Q1**, the limit would be set to one as the keywords `most populated` refers to `city`, which is singular. Otherwise, it is set to negative one, which means all possible answers.

If the query belongs to the *Filter* class, there is only one parameter to be set, which is the comparison operator (e.g. less than, more than, same as, etc.). In order to be able to set the correct operator, we train a classifier that predicts the operator given the keyword. The keywords are prepared by running the keyword identification component on the training sets. The classifier is trained on such keywords and their corresponding operator extracted from each SPARQL query. For instance, the operator *greater* with operand `1000000` would be extracted for the example question **Q2**.

Finally, after the KG property is selected and the values of the parameters are set, these results are used to augment the SPARQL query provided by the ranking model. This augmentation is done as follows i) first, we syntactically parse the query returned from the ranking model ; ii) we prepare the SPARQL equivalent for any of the parameters and/or constraints; iii) we append these additions to the query returned by the ranking model.

Figure 3 illustrates the flow of ExSQG with the example questions **Q1** and **Q2**. It shows each component in the pipeline with its inputs and its output when the system is given an *Ordinal* or *Filter* question.

4 Empirical Study

In this section, we introduce the datasets used in this work and provide statistical information about them. In addition, we present the results of ExSQG on the benchmarking datasets.

4.1 Datasets

Q/A datasets commonly contain triples of i) natural language question, ii) the equivalent formal query, and iii) the answer set. Since many of the Q/A datasets only contain *List* questions with no extra features such as *Ordinal* and *Filter*, we hand-picked the ones that include such questions from multiple datasets, so that we could build a robust and general query generator. In order to have a

unified dataset, we aim to collect the datasets with the same underlying KG. Among others, DBpedia [2] is an ongoing community-based knowledge base that is in a constant process of development and we would use the datasets, which are based on DBpedia.

First, we use LC-QuAD[27], which contains 5,000 manually crafted questions and their corresponding SPARQL query. Although LC-QuAD does not contain any questions that belong to the new types, we include it in order to provide performance comparison with the baseline system (SQG). Second, we use all the datasets from the QALD [7] challenge (QALD 1-9). As these datasets where part of a Q/A campaign over multiple years, many of the questions are used more than once in these datasets (out of more than 5,000 questions in these datasets, only 1400 are unique). However, these datasets are particularly important since they contain all of the types of questions, and they are carefully designed to challenge different aspect of the QA systems. The last dataset we use is DBNQA [21], which is a template-based dataset containing about 800,000 automatically created question and SPARQL query pairs. This dataset is especially useful, since it provides a vast number of questions from the *Filter* and *Ordinal* types.

Although DBNQA contains the target question types, it is generated using a set of pre-defined templates. Thus, if we train the classifiers on DBNQA, it would be biased towards the underlying templates. On the other hand, considering the number of unique question/query pairs in the QALD 1-9 challenge, it is not sufficient to train the classifiers. Therefore, we combine training and testing sets from all the available datasets.

The idea behind these combinations is to compare the performance of the models trained on each of the combinations with each other. These combinations are as follows:

- LC-QuAD: Using only LC-QuAD
- LC-QuAD + QALD: Combined data from both datasets
- LC-Quad + QALD + DBNQA: Combined data from all the datasets

Since DBNQA has over 800,000 questions-query pairs, while LC-QuAD and QALD contain about 10,000 questions combined, we do not include DBNQA entirely but rather use a subset of the dataset in order to avoid the classifiers' overfitting over questions from DBNQA. We used random different subsets with different sizes that varied between **1%**, **5%**, **10%**, and **25%** from the available questions in the dataset.

Using these multiple subsets gives us a better idea when the model gives the best performance, while decreasing the chance of overfitting over DBNQA. The combined datasets are named as follows:

- LC-QuAD + QALD + 1% DBNQA: Combined 1
- LC-QuAD + QALD + 5% DBNQA: Combined 5
- LC-QuAD + QALD + 10% DBNQA: Combined 10
- LC-QuAD + QALD + 25% DBNQA: Combined 25

[7] http://qald.aksw.org/

For the secondary classifiers, we prepare the training and testing sets using all the available data from all the available datasets. Since the amount of the available data for the secondary classes is not as much as the data available for the primary classes.

Table 2. Datasets Statistics

Dataset	# of Questions	Unique Questions	List	Boolean	Count	Ordinal	Filter
QALD (1-9)	5,237	1,396	1,056	98	79	94	75
LC-QuAD	5,000	4,998	3,967	368	658	0	0
DBNQA	894,499	871,166	688,689	76,835	98,372	3,893	1,797

Table 2 shows the total number of question and query pairs per dataset. In addition, it shows the total number of questions available for each type per dataset.

4.2 Experiment Settings

For the training process for any of the aforementioned classifiers, we prepared a train/test set from all the available data. We split each dataset as 70% for the training set and 30% for the test set. Furthermore, we use 10-fold cross-validation during the evaluation process. In addition, we use *scikit-learn*[8] implementations for all the classifiers used.

Moreover, for the cleaning process of questions, we use Spacey[9] and NLTK[10]. Finally, to prepare the embedding matrix, which contains the vector representation for all the words in our vocabulary, we use the pre-trained word vectors by Global Vectors for Word Representation (GloVe)[11] [23].

4.3 Evaluation Metrics

Since the proposed system architecture consists of a pipeline of components, in order to evaluate the performance of such a system, we first evaluate the performance of each component individually. Then we assess the overall performance of the system.

We evaluate the performance of the classifiers trained in terms of *accuracy*. In addition, we use *precision*, *recall*, and *F1-score* to measure the performance of the KG property selection component.

[8] https://scikit-learn.org/stable/
[9] https://spacy.io/
[10] https://www.nltk.org/
[11] https://nlp.stanford.edu/projects/glove/

4.4 Empirical Results

The selection process of the best classifier consists of two parts. First, we select the best classifier with the best set of features. Then, we experiment with the best performing classifier with the best set of features against different datasets with various sizes.

Table 3. Accuracy for the question classifier under different features

Feature	NB	SVM	MaxEnt
1-gram	91.0%	96.7%	98.5%
(1+2)-grams	95.3%	96.9%	98.9%
(1+2+3)-grams	95.7%	96.7%	98.9%
+TF-IDF	94.5%	92.4%	99.0%
+Normalized Numbers	95.7%	96.9%	99.0%
+POS	95.9%	96.4%	99.1%
First N-words N=3	93.6%	94.2%	96.2%
First Last N-words N=3	93.3%	95.3%	97.4%

Table 3 shows the accuracy of the question classifier under a different set of features. This experiment is done on the **combined dataset 5**. In order to select the best set of features, we show the accuracy of the classifier at each row for the current feature, combined with the best set of features selected so far. As the table shows, we end up using the *MaxEnt* classifier as it out-performed the other classifiers.

Table 4. MaxEnt Classifier Performance against multiple datasets of different sizes

Dataset	MaxEnt
LC-QuAD	90.1%
LC-QuAD + QALD	89.7%
Combined_1	95.9%
Combined_5	99.3%
Combined_10	99.5%

Table 4 shows the performance of the classifier when it is trained on different datasets. In this experiment, we use the **combined dataset 25** as the test for all the classifiers. We can see from the table that the performance of the question classifier increases with the size of the dataset. However, this increase could also be due to the classifiers overfitting over questions from DBNQA.

For the following experiments, we mainly focus on the QALD datasets to show the performance of the system as they are very popular and used a lot in

benchmarking QA over KG systems [9]. Thus, we are able to have a reference point to compare our approach with other systems.

Table 5. Accuracy of the question classifier on QALD (4, 5, 6, 7)

Dataset	No. Questions	Accuracy
QALD-4	67	51 (76%)
QALD-5	33	28 (84%)
QALD-6	99	87 (87%)
QALD-7	30	25 (83%)

Table 5 shows the accuracy of the hierarchical question classifier on QALD (4, 5, 6, 7). It also shows the total number of questions available per dataset. The accuracy of the proposed question classifier in Table 5 is less than the reported accuracy for the question classifier for SQG [33], because of the complex nature of the questions that belong to secondary classes.

Table 6. Performance of Ordinal Questions Pipeline

Dataset	Precision	Recall	F1
QALD-4	0.40	0.33	0.36
QALD-5	0.83	0.83	0.66
QALD-6	0.80	0.66	0.72
QALD-7	0.33	0.50	0.40

Table 6 shows the precision, recall, and F1 score for ExSQG for questions of type *Ordinal*. A generated SPARQL query is considered correct if it yields the same answer as the target SPARQL query, this means that the system is able to correctly classify the question and successfully generate the correct SPARQL query. The performance of the ExSQG is lower than the performance on QALD-5, and 6 for two reasons. First, by inspecting the questions that lead to an incorrect answer, we found out that the number of miss-classified questions from QALD-4 is higher than those of QALD-5, and 6. Second, most of the questions that belong to the *Ordinal* class from QALD-4 were generally more complex than those that belonged to QALD-6. Not in terms of linked items, rather in the queries that correspond to the question and the constraints used in such queries. For example, some query constraints are not simply KG relations but a count over such relations.

Table 7 shows the precision, recall, and F1 score of ExSQG for questions of type *Filter*. It also shows that the ExSQG system does not provide the same performance as it does for questions of the type *Ordinal*. This is due to the fact that there are much fewer questions of the type *Filter* that we support in the datasets

Table 7. Performance of Filter Questions Pipeline

Dataset	Precision	Recall	F1
QALD-4	0.11	1.00	0.20
QALD-6	0.14	0.33	0.20

than questions of type *Ordinal*. The current system is able to correctly generate the SPARQL for questions that require filtration over the value of a KB Relation (e.g. *"Cities in Germany with area larger than 30000 KM"*), or questions that compare two KB resources over a certain KB relation (e.g. *"Does Game of Thrones have more episodes than Breaking Bad"*). In the first question the constraint is the `dbo:areaTotal` and in the second one – `dbo:numberOfEpisodes`. On the other hand, questions that require a string matching filter query, date matching, or filtration based on a count are not yet supported. Therefore, any miss-classification or incorrect query generation would significantly impact the overall performance. The results for QALD-5,7 are not shown as well in this table, because there were only 3 filter questions and our system was not successful to correctly predict and answer them.

Table 8. Absolute increase percentage in performance between the SQG [33] and ExSQG

Dataset	No. of Questions	Performance Increase
QALD 4	67	8.0%
QALD 5	33	18.0%
QALD 6	99	5.0%
QALD 7	30	3.0%

Table 8 shows the absolute difference in performance between SQG [33] and the ExSQG. For this experiment, we assume an ideal scenario for the question classifier for both systems – SQG [33] and ExSQG. We also assume that we always get an intermediate SPARQL query from the ranking model for questions that belong to the new types. These conditions are assumed to mitigate any error propagation from SQG [33] and to be able to measure the performance of the ExSQG on questions that belong to the new types. The variation of the performance of ExSQG on QALD (4, 5, 6 and7) as shown in Table 8 is due to the fact that there is only a limited number of questions that belong to secondary classes in these datasets. However, there are more questions that have secondary classes in QALD-5 in comparison to the other datasets.

5 Conclusions

Encouraged by the existing efforts on query generation in the QA community, we presented ExSQG as an extension to an available query generator component (SQG [33]) in order to support *Filter* and *Ordinal* questions. We provided a hierarchical architecture for a question classifier, which yields high accuracy in different benchmarking datasets. Furthermore, ExSQG augments the query using identified keywords from the question and match them to the linked items from the KG based on word embedding techniques. Finally, we empirically showed that ExSQG achieves state-of-the-art accuracy on the benchmarking datasets.

Considering the upcoming Q/A datasets such as LC-QuAD 2.0 [10], which not only contains *Ordinal* and *Filter* in about 17% out of 50k questions, but also includes new types such as *aggregation*, which appears in more than 4% of all questions, we aim to expand our work to also cover aggregation.

6 Acknowledgments

This research was supported by the European Union H2020 project CLEOPATRA (ITN, GA. 812997) as well as by the German Federal Ministry of Education and Research (BMBF) funding for the project SOLIDE (no. 13N14456).

References

1. Abujabal, A., Yahya, M., Riedewald, M., Weikum, G.: Automated template generation for question answering over knowledge graphs. In: Proceedings of the 26th international conference on world wide web. pp. 1191–1200. International World Wide Web Conferences Steering Committee (2017)
2. Auer, S., Bizer, C., Kobilarov, G., Lehmann, J., Cyganiak, R., Ives, Z.: Dbpedia: A nucleus for a web of open data. In: The semantic web, pp. 722–735. Springer (2007)
3. Berant, J., Chou, A., Frostig, R., Liang, P.: Semantic parsing on freebase from question-answer pairs. In: Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing. pp. 1533–1544 (2013)
4. Berant, J., Liang, P.: Semantic parsing via paraphrasing. In: Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers). pp. 1415–1425 (2014)
5. Bollacker, K., Evans, C., Paritosh, P., Sturge, T., Taylor, J.: Freebase: a collaboratively created graph database for structuring human knowledge. In: Proceedings of the 2008 ACM SIGMOD international conference on Management of data. pp. 1247–1250. AcM (2008)
6. Bordes, A., Chopra, S., Weston, J.: Question answering with subgraph embeddings. arXiv preprint arXiv:1406.3676 (2014)
7. Bordes, A., Usunier, N., Chopra, S., Weston, J.: Large-scale simple question answering with memory networks. arXiv preprint arXiv:1506.02075 (2015)
8. Diefenbach, D., Both, A., Singh, K., Maret, P.: Towards a question answering system over the semantic web. Semantic Web (Preprint), 1–19 (2018)

9. Diefenbach, D., Lopez, V., Singh, K., Maret, P.: Core techniques of question answering systems over knowledge bases: a survey. Knowledge and Information systems **55**(3), 529–569 (2018)
10. Dubey, M., Banerjee, D., Abdelkawi, A., Lehmann, J.: Lc-quad 2.0: A large dataset for complex question answering over wikidata and dbpedia. In: Proceedings of the 18th International Semantic Web Conference (ISWC). Springer (2019)
11. Dubey, M., Dasgupta, S., Sharma, A., Höffner, K., Lehmann, J.: Asknow: A framework for natural language query formalization in sparql. In: International Semantic Web Conference. pp. 300–316. Springer (2016)
12. Hakimov, S., Unger, C., Walter, S., Cimiano, P.: Applying semantic parsing to question answering over linked data: Addressing the lexical gap. In: International Conference on Applications of Natural Language to Information Systems. pp. 103–109. Springer (2015)
13. Hamon, T., Grabar, N., Mougin, F., Thiessard, F.: Description of the pomelo system for the task 2 of qald-2014. In: CLEF (Working Notes). pp. 1212–1223 (2014)
14. Höffner, K., Walter, S., Marx, E., Usbeck, R., Lehmann, J., Ngonga Ngomo, A.C.: Survey on challenges of question answering in the semantic web. Semantic Web **8**(6), 895–920 (2017)
15. Kusner, M., Sun, Y., Kolkin, N., Weinberger, K.: From word embeddings to document distances. In: International Conference on Machine Learning. pp. 957–966 (2015)
16. Lehmann, J., Isele, R., Jakob, M., Jentzsch, A., Kontokostas, D., Mendes, P., Hellmann, S., Morsey, M., van Kleef, P., Auer, S., Bizer, C.: DBpedia - a large-scale, multilingual knowledge base extracted from wikipedia. Semantic Web Journal **6**(2), 167–195 (2015)
17. Lindberg, D.A., Humphreys, B.L., McCray, A.T.: The unified medical language system. Yearbook of Medical Informatics **2**(01), 41–51 (1993)
18. Lukovnikov, D., Fischer, A., Lehmann, J., Auer, S.: Neural network-based question answering over knowledge graphs on word and character level. In: Proceedings of the 26th international conference on World Wide Web. pp. 1211–1220. International World Wide Web Conferences Steering Committee (2017)
19. Maheshwari, G., Trivedi, P., Lukovnikov, D., Chakraborty, N., Fischer, A., Lehmann, J.: Learning to rank query graphs for complex question answering over knowledge graphs. In: International Semantic Web Conference. Springer (2019)
20. Miller, G.A.: Wordnet: a lexical database for english. Communications of the ACM **38**(11), 39–41 (1995)
21. Ngomo, N.: 9th challenge on question answering over linked data (qald-9). language **7**, 1
22. Nilesh Chakraborty, Denis Lukovnikov, G.M.P.T.J.L.A.F.: Introduction to neural network based approaches for question answering over knowledge graphs (2019)
23. Pennington, J., Socher, R., Manning, C.: Glove: Global vectors for word representation. In: Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP). pp. 1532–1543 (2014)
24. Shekarpour, S., Marx, E., Ngomo, A.C.N., Auer, S.: Sina: Semantic interpretation of user queries for question answering on interlinked data. Web Semantics: Science, Services and Agents on the World Wide Web **30**, 39–51 (2015)
25. Singh, K., Radhakrishna, A.S., Both, A., Shekarpour, S., Lytra, I., Usbeck, R., Vyas, A., Khikmatullaev, A., Punjani, D., Lange, C., et al.: Why reinvent the wheel: Let's build question answering systems together. In: Proceedings of the 2018

World Wide Web Conference on World Wide Web. pp. 1247–1256. International World Wide Web Conferences Steering Committee (2018)
26. SZ, H., et al.: Casia@ v2: A mln-based question answering system over linked data (2014)
27. Trivedi, P., Maheshwari, G., Dubey, M., Lehmann, J.: Lc-quad: A corpus for complex question answering over knowledge graphs. In: International Semantic Web Conference. pp. 210–218. Springer (2017)
28. Unger, C., Bühmann, L., Lehmann, J., Ngonga Ngomo, A.C., Gerber, D., Cimiano, P.: Template-based question answering over rdf data. In: Proceedings of the 21st international conference on World Wide Web. pp. 639–648. ACM (2012)
29. Walter, S., Unger, C., Cimiano, P., Bär, D.: Evaluation of a layered approach to question answering over linked data. In: International Semantic Web Conference. pp. 362–374. Springer (2012)
30. Wick, M.: GeoNames. GeoNames (2006)
31. Yih, S.W.t., Chang, M.W., He, X., Gao, J.: Semantic parsing via staged query graph generation: Question answering with knowledge base (2015)
32. Yin, W., Yu, M., Xiang, B., Zhou, B., Schütze, H.: Simple question answering by attentive convolutional neural network. arXiv preprint arXiv:1606.03391 (2016)
33. Zafar, H., Napolitano, G., Lehmann, J.: Formal query generation for question answering over knowledge bases. In: European Semantic Web Conference. pp. 714–728. Springer (2018)
34. Zettlemoyer, L.S., Collins, M.: Learning to map sentences to logical form: Structured classification with probabilistic categorial grammars. arXiv preprint arXiv:1207.1420 (2012)