Formal Query Generation for Question Answering over Knowledge Bases

Hamid Zafar[1], Giulio Napolitano[2], Jens Lehmann[1,2]

[1] Computer Science Institute, University of Bonn, Germany;
hzafarta@cs.uni-bonn.de, jens.lehmann@cs.uni-bonn.de
[2] Fraunhofer IAIS, Germany; giulio.napolitano@iais.fraunhofer.de,
jens.lehmann@iais.fraunhofer.de

Abstract. Question answering (QA) systems often consist of several components such as Named Entity Disambiguation (NED), Relation Extraction (RE), and Query Generation (QG). In this paper, we focus on the QG process of a QA pipeline on a large-scale Knowledge Base (KB), with noisy annotations and complex sentence structures. We therefore propose SQG, a SPARQL Query Generator with modular architecture, enabling easy integration with other components for the construction of a fully functional QA pipeline. SQG can be used on large open-domain KBs and handle noisy inputs by discovering a minimal subgraph based on uncertain inputs, that it receives from the NED and RE components. This ability allows SQG to consider a set of candidate entities/relations, as opposed to the most probable ones, which leads to a significant boost in the performance of the QG component. The captured subgraph covers multiple candidate walks, which correspond to SPARQL queries. To enhance the accuracy, we present a ranking model based on Tree-LSTM that takes into account the syntactical structure of the question and the tree representation of the candidate queries to find the one representing the correct intention behind the question. SQG outperforms the baseline systems and achieves a macro F1-measure of 75% on the LC-QuAD dataset.

1 Introduction

Extensive progress has been made in recent years by systems using Knowledge Graphs (KGs) as their source of information. As the complexity of such systems may be considerable, it is common practice to segment the whole task into various subtasks usually performed sequentially, including Named Entity Disambiguation (NED), Relation Extraction (RE) and Query Generation (QG) among others [1]. This segmentation, however, rarely corresponds to true modularity in the architecture of implemented systems,which results in the general inability by the wider community to successfully and efficiently build upon the efforts of past achievements. To tackle this problem, researchers introduced QA modular frameworks for reusable components such as OKBQA [2,3] but little attention has been given to query generation. For instance, OKBQA includes 24

reusable QA components of which only one is a query generator. Nonetheless, the increasing complexity of questions leads to the following challenges for the query generation task [4,1]:

1. Coping with large-scale knowledge bases: Due the very large size of existing open-domain knowledge bases such as DBpedia [5] and Freebase [6], special consideration is required.
2. Question type identification: For instance the question might be a boolean one, thus the query construction should be carried out accordingly to generate desired answer.
3. Managing noisy annotations: The capability to handle a set of annotations including several incorrect ones might increase the chance of QG to construct the correct query.
4. Support for more complex question which requires specific query features such as aggregation, sort as well as comparison.
5. Syntactic ambiguity of the input question: For example, "Who is the father of X?" might be interpreted as "X is the father of who?" if the syntactical structure of the question is ignored.

To the best of our knowledge, none of the existing works can deal with all these challenges. Thereby, We present SPARQL Query Generator (SQG), a modular query builder for QA pipelines which goes beyond the state-of-the-art. SQG employs a ranking mechanism of candidate queries based on Tree-LSTM similarity and is able to process noisy input. We also considered scalability aspects, which enables us to evaluate SGQ on a large Q/A dataset based on DBpedia [5].

2 Related Work

Diefenbach et al. [1] studied more than 25 QA systems and discussed the different techniques these used for each component, including the query generation component. They show that in most QA systems the query generation component is highly mixed with components performing other tasks, such as segmentation or relation extraction. Furthermore, their work mainly focused the analysis on the overall performance of the QA systems, as only a few systems and publications provided deep analysis of the performance of their query generation component. For instance, CASIA [7], AskNow [8] and Sina [9] each have a SPARQL generation module, but do not provide an evaluation on it. [4] is a very comprehensive survey of question answering over knowledge graphs, analyzing 72 publications and describing 62 question answering systems for RDF knowledge graphs. In particular, they argue that it may not be beneficial that a wide range of research articles and prototypes repeatedly attempt to solve the same challenges: better performance may be obtained by providing sophisticated and mature solutions for individual challenges. An alternative approach to QA pipelines is end-to-end systems such as [10], which directly construct SPARQL queries from training data. However, those are currently mostly restricted to simple questions with

a single relation and entity in which case query generation is straightforward. While those systems are an interesting area of research and may be extensible to more complex questions, they are unlikely to completely replace QA pipelines due to the large amount of training data required.

To the best of our knowledge, there is a very limited number of working query builders in the question answering community which can be used independently and out of the box. Recent studies [3,2] explored the possibility of using independent QA components to form a fully functional QA system. Singh et al. [3] introduced Frankenstein, a QA framework which can dynamically choose QA components to create a complete QA pipeline, depending on the features of the input question. The framework includes two query building components: Sina and NLIWOD QB. NLIWOD is a simple, template-based QB, which does not include any kind of query ranking. Given correct inputs, NLIWOD compares the pattern based on a number of input entity and relation to build the output query. Sina [9] was originally a monolithic QA system, which did not provide a reusable query builder. However, Singh et al. [3] decoupled the query building component to make it reusable. Its approach consists in the generation of minimal sets of triple patterns containing all resources identified in the question and select the most probable pattern, minimizing the number of triples and of free variables. Another approach not far from ours is presented by Abujabal et al. [11]. In particular, they also rely on ranking methodologies for choosing among candidate queries. However they rely on question-query templates, previously learned by distant supervision, which are ranked on several features by a preference function learned via random forest classification. By contrast, our ranking approach is based on the similarity between candidate walks in a minimal subgraph and the question utterance.

3 Approach

We formally describe knowledge graphs and the query generation problem as a subproblem for question answering over knowledge graphs. Within the scope of this paper, we define a *knowledge graph* as a labelled directed multi-graph: a tuple $K = (E, R, T)$ where E is a set called entities, R is a set of relation labels and $T \subseteq V \times R \times V$ is a set of ordered triples. The definition captures the basic structure of, e.g., RDF datasets and property graphs.

We define *query generation* as follows: We assume that a question string s and a knowledge graph K are given. In previous steps of the QA pipeline, entity and relation linking have already been performed, i.e. we have a set of mappings M from substrings (utterances) of s to elements of E and R, respectively. The task of query generation is to use s, D and M to generate a SPARQL query.

This definition completely decouples the query generation task from the NED and RE tasks. However, in a realistic setting, the NED and RE modules produce a list of candidates per utterance in the question. As a result, we relax the constraint on the utterance annotation to have a list of candidates per each utterance. Formally, we define the task of *advanced query generation* based on

the definition of query generation, where each substring of s is mapped to a non-empty subset of E or R respectively, i.e. there are several candidates for entities and relations. For instance, Figure 1 illustrates that for the question "What are some artists on the show whose opening theme is Send It On?", the annotation for the "artists" might be *dbo:Artist* or *dbo:artists* and so on. In Section 4.2, we show that considering multiple candidates leads to a better performance, in comparison to the case where only the top candidate is considered by the query generator.

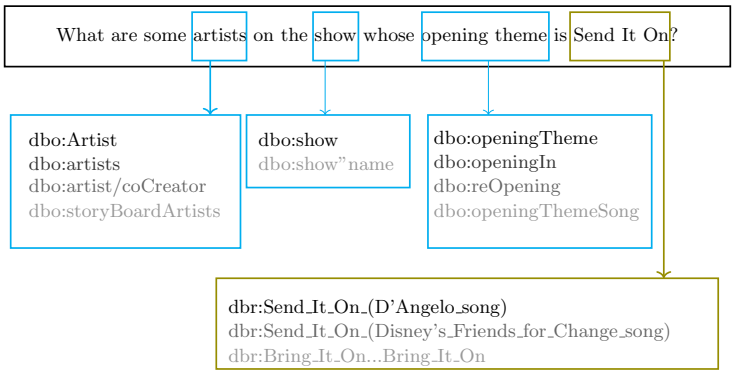

Fig. 1: A sample question annotated with output from NED and RE components. There is a list of candidates for each spotted utterance in the question ranked based on their confidence score.

Our hypothesis is that the formal interpretation of the question is a walk w in the KG which only contains the target entities E and relations R of the input question s plus the answer nodes. The definition of a (valid) walk is as follows:

Definition 1 (Walk). *A walk in a knowledge graph $K = (E, R, T)$ is a sequence of edges along the nodes they connect: $W = (e_0, r_0, e_1, r_1, e_2, \ldots, e_{k-1}, r_{k-1}, e_k)$ with $(e_i, r_i, e_{i+1}) \in T$ for $0 \leq i \leq k-1$.*

Definition 2 (Valid Walk). *A walk W is valid with respect to a set of entities E' and relations R', if and only if it contains all of them, i.e. :*

$$\forall e \in E' : e \in W \text{ and } \forall r \in R' : r \in W$$

A node e with $e \in W$ and $e \notin E'$ is *unbound*. An *unbound node* is an abstract node which is used to connect the other nodes in the walk. This node, however, can be related later to one or a set of specific nodes in the KG.

We carry out the task of capturing the valid walks in two steps: First, we establish a type for the question (e.g. boolean or count). Depending on the type, a number of valid walks are extracted from the KG, however, most of them may be an incorrect mapping of the input question, in the sense that they do not capture

the correct intention behind the question. Thus, we sort the candidate walks with respect to their similarity to the input question. The overall architecture of SQG is shown in Figure 2 and in the following sections we discuss each step in more detail.

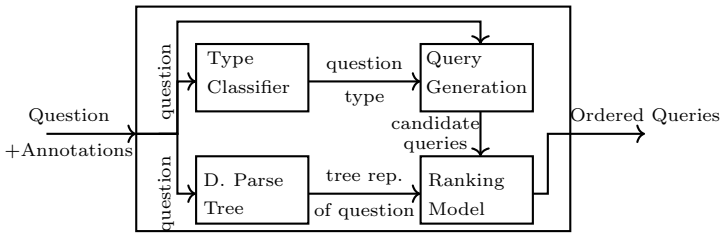

Fig. 2: The Architecture of SQG

3.1 Query Generation

In order to find candidate walks in the KG, we need to start from a linked entity $e \in E$ and traverse the KG. Given the size of the existing KGs such as DBpedia [12] or Freebase [6], however, it is very time-consuming to enumerate all valid walks. Thus, we restrict to the subgraph consisting of all the linked entities and relations. In this subgraph, we can then enumerate the candidate walks, which can be directly mapped to SPARQL queries. Yet, the type of question is a crucial feature that has to be identified in order to create the candidate queries with correct structure from the valid walks.

Capture the Subgraph We start with an empty subgraph, which is populated with the linked entities E as its nodes. Then, we augment the nodes with edges that correspond to the linked relations R, if such connections exist in the KG. This is illustrated in Figure 3, if only the solid lines are considered. In this step, we consider every possible direct way of connecting the entity(s) and the relation(s) to enrich the subgraph: a relation might connect two existing nodes in the subgraph, or it may connect an entity to a new unbound node. Thus, this subgraph might contain several valid walks but the correct one, since the intention of the question might require to include nodes in the two-hop distance. For instance, in Figure 3, the answer node ("$unbound_1$") is in the two-hop distance from the entity "dbr:Send_it_on" and is not included in the current subgraph. Consequently, we need to expand the subgraph with the set of candidate relations R. Depending on the question, such expansion might correspond to a very large portion of the underlying KG, which may not be even useful to create the final list of candidate walks. Instead of expanding the subgraph, we expand the existing edges of the subgraph with the set of candidate relations excluding the

relation that the existing edge represents. As a result, the search space in the underlying KG is greatly reduced (see Figure 3, where the dashed lines represent the edge expansions). Algorithm 1 summarize the process of capturing the subgraph.

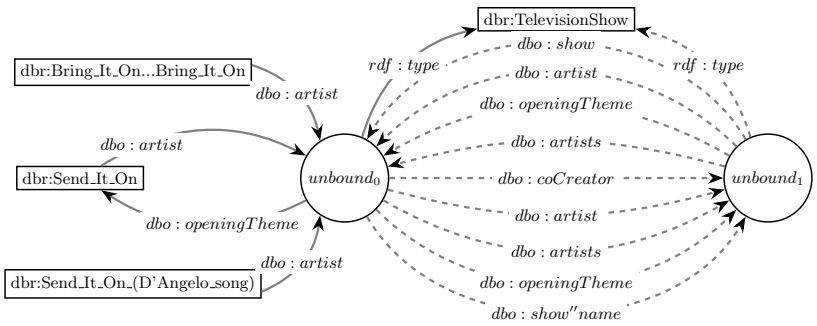

Fig. 3: The captured subgraph for the given question, annotated with candidate entities and relations; solid lines are the one that are in one hop distance; dashed line means more than one hop distance; circles represent unbound nodes, rectangles are the linked entities and edges are the relations in the KG.

Enumerate Candidate walks At this point, we have a subgraph that covers all the entities and the relations in the question. We consider every unbound node as a potential answer node, therefore we look for valid walks according to Definition 2. For our example, Figure 4 reveals four valid walks. In case there is only one valid walk in the subgraph, we map the walk to a SPARQL query and report it as the corresponding query for the given question. Note that further augmentations might be required to support different types of questions, such as those requiring counts or returning boolean values. If there is more than one valid walk, then ranking (as described below) needs to be performed in order to find the most similar query to the input question.

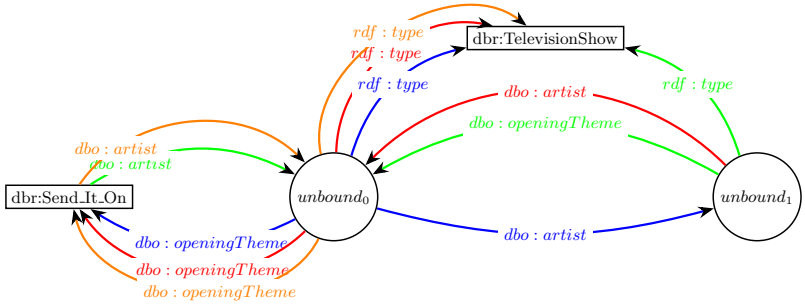

Fig. 4: Four candidate walks are found which are shown in different colors

```
Data: E', R', K
Result: G: Minimal covering subgraph
Initialize G as an empty graph;
Add ∀e ∈ E' to G as nodes;
foreach e ∈ E', r ∈ R' do
    if (e, r, ?) ∈ K then
        add (e, r, ?) to G ;
    else if (?, r, e) ∈ K then
        add (?, r, e) to G ;
end
foreach (e1, r, e2) ∈ G do
    foreach r' ∈ R', r' ≠ r do
        if (e2, r', ?) ∈ K then
            add (e2, r', ?) to G ;
        else if (?, r', e2) ∈ K then
            add (?, r', e2) to G ;
        else if (e1, r', ?) ∈ K then
            add (e1, r', ?) to G ;
        else if (?, r', e1) ∈ K then
            add (?, r', e1) to G ;
    end
end
```

Algorithm 1: Capture the subgraph

Question Type classification SQG supports simple and compound questions. In order to support questions such as boolean and count questions, we first need to identify the type of question. In some works, predefined patterns were used for similar purposes [13,8,14]. However, we train an *SVM* and *Naive Bayes* model to classify the questions into boolean, count or list questions based on their *TF-IDF* representation. Although TF-IDF is a simple representation, the results are shown in Section 4.2 reveal that it is strong enough for the purpose.

Given the class of the question, the query generator will format the query accordingly. In the case of a count query, for instance, the query generator adds the corresponding function to the output variable of the SPARQL query.

3.2 Query Ranking

There might be more than one valid walk, due to the uncertainty in the linked entities/relations and complexity of the KGs. As a result, we need a way to rank them with respect to the intention of the input question. Yit et al. [13] proposed a model based on convolutional neural networks to represent the input questions and the core-chains in a latent semantic space and compute the cosine similarity. Although the order of the words is captured to some extent in the model, the overall structure of the input question and candidate core-chains is not taken into account. Considering the fact that the walks consist of many shared entities/relations, our hypothesis for the ranking model is that the structure of the

walks is a distinctive feature to distinguish the similarity between the candidate walks and the input question. For instance, four walks are generated for our running example, which have distinct structures (see Figure 7). Therefore, the desired model should be able to incorporate the structure of the input. Tai et al. [15] presented Tree-LSTM, an enhanced version of the vanilla LSTM, which considers the tree representation of the input rather than just the sequence of input tokens. They applied the model to the semantic relatedness of natural language sentences and sentiment classification. We present a ranking model based on Tree-LSTM. It considers the tree representation of the candidate walks with respect to the syntactical structure of the question in order to compute their similarity. In the following, we shortly introduce LSTM and Tree-LSTM, subsequently we discuss the final model to fulfill the task.

Preliminaries As opposed to the vanilla neural networks, where the inputs at different steps are assumed to be independent, Recurrent Neural Networks (RNN) have a hidden state h_t at each step t, which depends not only on the current input but also the previous hidden state h_{t-1}. This formulation enables RNNs to memorize the previous computations for an arbitrarily long sequence of inputs. In practice, however, it has been pointed out that the memory functionality of the RNNs is limited to a few steps back [16]. As a remedy to this issue, RNNs with Long-Short Term Memory (LSTM) units were introduced [17]. They are empowered by a memory cell which is able to keep the state for a longer period of time.

Tree-LSTM RNNs and LSTMs consume the input in a sequential manner. However, when the structure of the input is not simply sequential, special treatment is required. Tai et al. altered the architecture of LSTM to support tree-structured input [15]. Tree-LSTMs aim to incorporate information that rests in the child nodes, whilst LSTM support only sequential propagation. Tree-LSTM units take into account the state of its child nodes to compute its internal state and the output. This architecture enables Tree-LSTMs to easily incorporate the tree structure of our input.

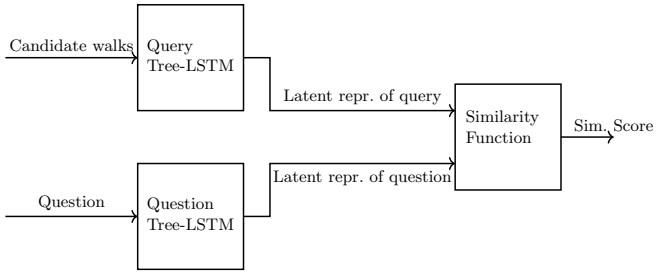

Fig. 5: Ranking Model Architecture

Ranking Model Figure 5 shows the architecture of the ranking model, in which two Tree-LSTMs are used to map the input walks and question into a latent

vectorized representation. We will later apply a similarity function to rank the candidate queries based on the similarity score. In the preparation phase for the input question to the *Question Tree-LSTM*, we substitute the surface mentions of the entities in the question with a placeholder After that, the dependency parse tree is created (see Figure 6). Furthermore, the *Query Tree-LSTM* receives the tree representation of the candidate walks. Figure 7 depicts the candidate walks of the running example. While all of the candidate walks are valid, only 7a provides the correct interpretation of the input question.

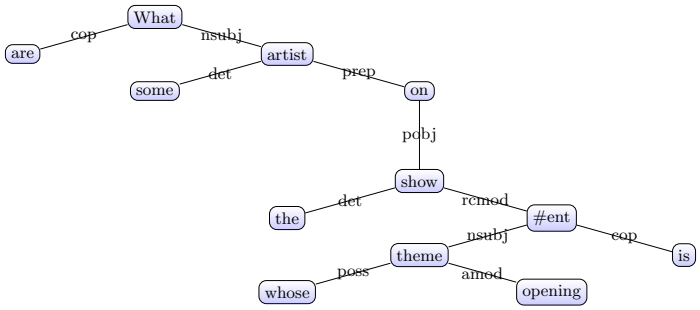

Fig. 6: Dependency parse tree of question: "What are some artists on the show whose opening theme is Send It On?"

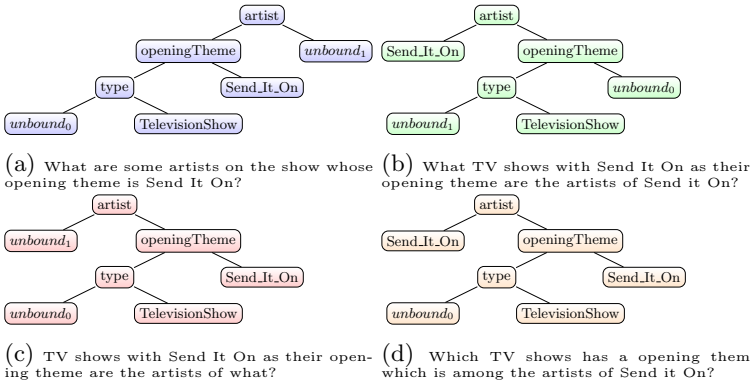

Fig. 7: Tree representation of the candidate walks along with their NL meaning, using the same colors as Figure 4

4 Empirical Study

In this section, we discuss the benchmark dataset and compare the result of SQG with two baseline systems. We further provide an extensive analysis of SQG on its different sub-modules such as an evaluation of question type classification, query generation, and ranking model.

SQG is implemented using Python/pytorch. Additionally, we use Glove word embedding [18] as the word representation in the ranking model. The code of SQG is published on our GitHub page `https://github.com/AskNowQA/query_generation`.

4.1 Datasets

We use the *LC-QuAD* [19] dataset in our experiments. The dataset consists of 5,000 question-answer pairs that cover different complexity and types of questions, such as simple and compound, boolean and count. Each pair is also annotated with the corresponding SPARQL query, target entities, and relations. The annotations are compatible with DBpedia [5] (version 2016-04).

4.2 Performance Evaluation

We measure the performance of SQG in terms of precision, recall and F1-measure on a subset of LC-QuAD containing 3,200 questions and it is the same subset as in Singh et al. [3] which reported the performance of Sina QB and NLIWOD QB using the same metrics. Table 1 demonstrates that SQG significantly outperforms the baseline systems.

Table 1: The comparison of SQG with existing works

Approach	Precision	Recall	F1-measure
Sina*	0.23	0.25	0.24
NLIWOD*	0.48	0.49	0.48
SQG	0.76	0.74	0.75

* Result of the baseline systems are taken from [3]

There are three shortcomings in the baseline systems. First, they require the correct entity/relation as input as NLIWOD does not support multiple candidates and Sina fails when there are more than three candidates. Second, the query augmentation ability is limited in both systems, and third, they lack a ranking mechanism, which is required to reorder the list of candidate queries with respect to the input question. We have addressed these problems in our proposed approach and, in the next section, we first evaluate the question type

classification in SQG. Afterwards, we define different scenarios, in which the input of the query generator varies from only target entity/relations to the list of candidates per utterance. Finally, we examine the performance of our Tree-LSTM as the ranking mechanism.

Question Type Classification First, we assess the accuracy of two different classifiers. Note that the results are independent of the entity/relation linking module, as they take no additional input from it. We perform a 10-fold cross validation on 50% of the dataset to train the model and find the optimal parameter values. We then evaluate the classifiers using the optimal parameter values on the test set. The performance results of both classifiers, in terms of precision, recall and F1-measure are shown in Table 2. A large amount of diverse training data ensures that the models, albeit simple, perform satisfactorily. Avoiding a manually crafted set of patterns enables SQG to be more applicable in different settings without further manual intervention.

Table 2: The accuracy of question type classifiers on the LC-QuAD Dataset

Model	Precision	Recall	F1-measure
Naive Bayes	0.92	0.92	0.91
SVM	0.99	0.99	0.99

Query Generator Evaluation In this section, we introduce three scenarios to evaluate the query generator as well as the ranking model in various settings:

Top-1 correct We supply only the correct target linked entity/relation to provide an upper-bound estimation of our performance.

Top-5 EARL+correct We consider a more realistic setting where a list of 5-candidates per entity/relation from EARL [20], a tool for NER and RE, is provided. In order to evaluate SQG independently of the performance of the linker system, we insert the correct target linked entity/relation if it does not already exist on the list. The purpose of this scenario is to assess the robustness of SQG when the input annotations include several incorrect links as well as the correct ones.

Top-5 EARL We use the output of EARL [20] to evaluate the QG component in a fully functional QA system.

The evaluation results of query generation for all three scenarios are summarized in Table 3.

In *Top-1 correct*, the query generator failed once to generate any candidate query, while in six other cases none of the candidate query(s) returns the desired answer. Note that, in this scenario, the ratio of generated queries per question is close to one, because only the true target entities/relations were given to the query generator. Consequently, the number of valid walks is very low.

In the second row of Table 3, which corresponds to *Top-5 EARL+correct*, we observe that the query generator is able to process the noisy inputs and cover 98% of the questions. Furthermore, the average number of generated query per question has increased to 2.25. In the next section, we study how this increase in the average number of generated query affects the ranking model.

The performance in *Top-5 EARL* has dropped dramatically in comparison to the first two scenarios. The main reason is that for 85% of the questions, EARL provided partially correct annotations, which means that either there are utterance(s) in the question which are annotated with an incorrect set of entities/relations, or there are utterance(s) in the question which should not be annotated, while EARL incorrectly annotates them with a set of entities/relations.

However, if we consider only the questions where EARL manages to include all the correct target links to SQG, the coverage is again 98%, which is consistent with the result of the artificial injection experiment from *Top-5 EARL+correct*.

Table 3: Evaluation metrics of Query Generator

Scenario	Incorrect(%)	No walk (%)	Covered (%)	Avg. #query
Top-1 correct	0.01	0.0	0.99	1.18
Top-5 EARL+correct	0.02	0.0	0.98	2.25
Top-5 EARL	0.12	0.72	0.16	3.28

Ranking Model Evaluation Before we analyze the ranking model, we examine the dataset that is prepared for it. We employ the Stanford Parser [21] to generate the dependency parse tree of the input questions and the generated candidate queries from the output of the query generator to create the dataset for the ranking model. We split the dataset into 70%/20%/10% for the train set, development set, and test set, respectively. Table 4 provides some insights on the dataset, on which the ranking model is evaluated.

In the *Top-1 correct* scenario, the number of generated queries per question is 1.18, because there are not many possible ways to connect the linked entity/relations. Moreover, the first row of Table 4 demonstrates that the number of incorrect items in the dataset is not proportional to the number of correct items. As a result, the ranking model is given an unbalanced dataset, since the dataset mostly contains positive examples. In *Top-5 EARL+correct*, the number of generated queries is higher, which leads to not only more training data for the ranking model but also more diverse data such that we would not face the same problem as in the previous scenario. The second row of Table 4 shows that the distribution of incorrect and correct data is almost equal, which leads to an increase in the performance of the ranking model, as it provides the model enough data on both classes to learn the set of parameters that perform well. In *Top-5 EARL*, the distribution of correct and incorrect instances in the datasets is not balanced, though the average number of generated queries is higher than

in the other scenarios. Note that in this scenario, the total number of generated queries is far lower than in the other scenarios, because the 72% of cases no walk is generated. The peak in the number of incorrect instances is caused by wrong annotations, which were provided by the NED and RE component.

Table 4: The distribution of incorrect and correct data per scenario

Scenario	Size	Train		Dev		Test	
		Correct	Incorrect	Correct	Incorrect	Correct	Incorrect
Top-1 correct	5,930	0.85	0.15	0.84	0.16	0.87	0.13
Top-5 EARL+correct	11,257	0.46	0.54	0.46	0.53	0.48	0.52
Top-5 EARL	4,519	0.20	0.80	0.19	0.81	0.22	0.78

We have used two similarity functions in the ranking model: Cosine similarity and a neural network-based function analogous the approach presented in [15]. The neural network computes the distance and the angle between the two latent representations of the query. However, the neural network-based approach had superior performance compared to the cosine similarity function. Thus, in the following, we provide the result of the ranking model using the neural network as the similarity function. Note that for every scenario, the parameters of the similarity network was tuned on the development set.

In the *Top-1 correct*, despite the unbalanced distribution of the dataset, the ranking model achieves an F1 measure of 74%. This, however, does not accurately reflect the performance results for the ranking model as the average number of generated queries in this scenario is 1.18, which means that mostly there is only one candidate query per question.

As the second row of the Table 5 reveals, the F1-measure increases from 75% in *Top-1 correct* to 84% in *Top-5 EARL+correct*. In contrast to the Top-1 correct, the average number of generated queries in *Top-5 EARL+correct* is almost double to 2.25 and the distribution of correct/incorrect example is almost balanced. This results show that the model performs better in comparison to *Top-1 correct* when it is given a bigger and balanced dataset.

A micro F1-measure of 74% was achieved the *Top-5 EARL*. Further analysis reveals that there are two main factors for the drop in the performance: first, the input dataset is highly unbalanced towards incorrect instances; Additionally, the size of the is considerably smaller than the one in *Top-5 EARL+correct*.

5 Conclusions and future works

We discussed the challenges of query generation in QA systems and introduced SQG, a two-step SPARQL query generator as a reusable component that can be easily integrated into QA pipelines. In the first step, a set of candidate queries are generated, which is followed by a ranking step that arranges the candidate queries based on their structural similarity with respect to the dependency parse tree of the input question. We provide a detailed analysis of the effect of different factors

Table 5: Micro accuracy of the ranking model using the top ranked item

Scenario	Precision (%)	Recall (%)	F1 (%)
Top-1 correct	0.77	0.74	0.75
Top-5 EARL+correct	0.84	0.84	0.84
Top-5 EARL	0.73	0.75	0.74

on the performance of QG component such as question type detection, noisy input and ranking the candidate queries. In our experiments, SQG outperformed current query generation approaches.

In future work, we plan to add support for more question types, which requires union, sorting, comparison or other aggregation functions such as min and max. Also, we will investigate the accuracy of SQG on other QA datasets that might shed light on unseen weaknesses.

Acknowledgments

This research was supported in part by an EU H2020 grant provided for the HOBBIT project (GA no. 688227) as well as by German Federal Ministry of Education and Research (BMBF) funding for the project SOLIDE (no. 13N14456).

We would also like to thank our colleagues Kuldeep Singh and Mohnish Dubey for providing us additional data and their helpful and inspiring comments.

References

1. D. Diefenbach, V. Lopez, K. Singh, and P. Maret, "Core techniques of question answering systems over knowledge bases: a survey," *Knowledge and Information systems*, pp. 1–41, 2017.
2. J.-D. Kim, C. Unger, A.-C. N. Ngomo, A. Freitas, Y.-g. Hahm, J. Kim, S. Nam, G.-H. Choi, J.-u. Kim, R. Usbeck, *et al.*, "Okbqa framework for collaboration on developing natural language question answering systems," 2017.
3. K. Singh, A. S. Radhakrishna, A. Both, S. Shekarpour, I. Lytra, R. Usbeck, A. Vyas, A. Khikmatullaev, D. Punjani, C. Lange, M. E. Vidal, J. Lehmann, and S. Auer, "Why reinvent the wheel–lets build question answering systems together," in *The Web Conference (WWW 2018), to appear*, 2018.
4. K. Höffner, S. Walter, E. Marx, R. Usbeck, J. Lehmann, and A.-C. Ngonga Ngomo, "Survey on challenges of question answering in the semantic web," *Semantic Web*, vol. 8, no. 6, pp. 895–920, 2017.
5. S. Auer, C. Bizer, G. Kobilarov, J. Lehmann, R. Cyganiak, and Z. Ives, "Dbpedia: A nucleus for a web of open data," *The semantic web*, pp. 722–735, 2007.
6. K. Bollacker, C. Evans, P. Paritosh, T. Sturge, and J. Taylor, "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.

7. S. He, Y. Zhang, K. Liu, and J. Zhao, "Casia@ v2: A mln-based question answering system over linked data.," 2014.
8. M. Dubey, S. Dasgupta, A. Sharma, K. Höffner, and J. Lehmann, "Asknow: A framework for natural language query formalization in sparql," in *International Semantic Web Conference*, pp. 300–316, Springer, 2016.
9. S. Shekarpour, E. Marx, A.-C. N. Ngomo, and S. Auer, "Sina: Semantic interpretation of user queries for question answering on interlinked data," *Web Semantics: Science, Services and Agents on the World Wide Web*, vol. 30, pp. 39–51, 2015.
10. D. Lukovnikov, A. Fischer, J. Lehmann, and S. Auer, "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.
11. A. Abujabal, M. Yahya, M. Riedewald, and G. Weikum, "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.
12. J. Lehmann, R. Isele, M. Jakob, A. Jentzsch, D. Kontokostas, P. Mendes, S. Hellmann, M. Morsey, P. van Kleef, S. Auer, and C. Bizer, "DBpedia - a large-scale, multilingual knowledge base extracted from wikipedia," *Semantic Web Journal*, vol. 6, no. 2, pp. 167–195, 2015.
13. S. W.-t. Yih, M.-W. Chang, X. He, and J. Gao, "Semantic parsing via staged query graph generation: Question answering with knowledge base," 2015.
14. V. Lopez, M. Fernández, E. Motta, and N. Stieler, "Poweraqua: Supporting users in querying and exploring the semantic web," *Semantic Web*, vol. 3, no. 3, pp. 249–265, 2012.
15. K. S. Tai, R. Socher, and C. D. Manning, "Improved semantic representations from tree-structured long short-term memory networks," in *ACL 2015, July 26-31, 2015, Beijing, China, Volume 1: Long Papers*, pp. 1556–1566, 2015.
16. Y. Bengio, P. Simard, and P. Frasconi, "Learning long-term dependencies with gradient descent is difficult," *IEEE transactions on neural networks*, vol. 5, no. 2, pp. 157–166, 1994.
17. S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.
18. J. Pennington, R. Socher, and C. Manning, "Glove: Global vectors for word representation," in *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, pp. 1532–1543, 2014.
19. P. Trivedi and M. Dubey, "A corpus for complex question answering over knowledge graphs," in *16th International Semantic Web Conference*, 2017.
20. M. Dubey, D. Banerjee, D. Chaudhuri, and J. Lehmann, "Earl: Joint entity and relation linking for question answering over knowledge graphs,"
21. D. Chen and C. Manning, "A fast and accurate dependency parser using neural networks," in *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, pp. 740–750, 2014.