Machine Learning of SPARQL Templates for Question Answering over LinkedSpending *(Discussion Paper)*

Roberto Cocco[1], Maurizio Atzori[1], and Carlo Zaniolo[2]

[1] Department of Math/CS (DMI), University of Cagliari (Italy)
robertococco@outlook.com atzori@unica.it
https://dmi.unica.it/atzori
[2] CS Department, University of California, Los Angeles (USA)
zaniolo@cs.ucla.edu
http://web.cs.ucla.edu/~zaniolo/

Abstract. We present a Question Answering system aimed to answer natural language questions over open RDF spending data provided by LinkedSpeding. We propose an original machine-learning approach to learn generalized SPARQL templates from an existing training set of *(NL question, SPARQL query)* pairs. In our approach the generalized SPARQL templates are fed to an instance-based classifier that associates a given user-provided question to an existing pair, that is used to answer the user question. We employ an external tagger, delegating the Named-Entity Recognition (NER) task to a service developed for the domain we want to question. The problem is particularly challenging due to the small training set size available, counting only 100 questions/SPARQL queries.

We illustrate the results of our new approach using data provided by the Question Answering over Linked Data challenge (QALD-6) task 3, showing that it can provide a correct answer to 14 of the 50 questions of the test set. These results are then compared to existing systems, including QA^3, our previous work where templates were provided by an expert instead of being generated automatically from a training set.

Keywords: Question Answering · SPARQL · Semantic Web · Machine Learning

1 Introduction

The recent years saw a steady growth of structured data made available to the public, with governments giving their contribution by publishing information about public expenses. At the same pace the need grows to make this data available to non-technical users. In this paper, we propose a tool capable of answering a question posed in natural language by a user with no experience of RDF datasets and SPARQL queries. This is a follow-up to our previous template-based question-answering system called QA^3 [1], with the main difference being the way templates are obtained. In fact, while QA^3 requires the templates to be generated by an experienced user, in this paper we present a system that is able to learn new templates from datasets fed to an intelligent template generator. Our system works by processing a dataset containing *(question, SPARQL query)* pairs in input. Specific references contained in each question and its associated SPARQL query are automatically linked by our system, and treated as "variables" that may vary in different user questions. In other words, both questions and SPARQL queries are generalized into templates that can be filled in with different values w.r.t. the instances available in the dataset.

At answering time, the user question is subject to a similar generalization process, whereby it is matched to a list of generalized templates containing the same tags and ordered by the Jaccard Index (also known as *intersection over union*).
The best matching templates, and their associated queries, are then filled in with the data so obtained by tagging the user question. Due to the kind of approach used, in some cases more than a query per template is reconstructed, so we show the user the template question filled in with the highest Jaccard Index and ask the user if this has the same meaning as the original one. This allows our system to exploit user's feedback to further refine the initial training set. Furthermore, the system keeps presenting the user with questions until he/she returns a positive response or no more questions are left to show. This results in a very flexible system, requiring the user to input only non-technical data as long as an initial small training set of question/answer pairs.

RDF. The Resource Description Framework[3] is a set of specifications used to represent graph data. It provides us with a general method to decompose knowledge into triples, composed of a *subject*, a *predicate* and an *object*.

Linked Spending. With more and more governments providing spending data to the public, Linked Spending[4] took on the task of making open spending data available via RDF datacubes, a W3C standard [2]. It now provides more than 2 million planned or carried out financial transactions.

Question answering and QALD. Question answering systems focus on correctly answering questions posed in natural language, instead of retrieving information on the basis of user-generated keywords as traditional search engines do [4] or SQL-like query languages like SPARQL. This problem can be approached in many different ways, but the main challenges can be considered: *(i)* Extracting the relevant information from the questions and mapping it to the datacube, *(ii)* Dealing with terms ambiguity, *(iii)* Working with more than one dataset, *(iv)* Data quality and heterogeneity. In this paper, we leverage

[3] `https://www.xml.com/pub/a/2001/01/24/rdf.html`
[4] `http://linkedspending.aksw.org/`

the award-winning results of QA^3, and focus on automating (through machine learning) the process of generating templates, which previously required a try-and-error approach by a human expert.

Question Answering over Linked Data[5] (QALD) is a series of evaluation campaigns that provides an up-to-date benchmark for assessing and comparing question answering systems that query RDF-based Linked Data. Over the years, QALD has generated a series of training/test sets. The relevant one for this work is Task 3 of QALD6, containing a training set of 100 instances of *(question, SPARQL query)* pairs for LinkedSpending, and another 50 pairs as test set.

Template-based approaches. Template-based approaches to question answering work by constructing templates (or pseudo queries) from a linguistic analysis of the input questions [4]. These templates are neutral, as they contain no reference to the dataset, and they stand in the middle between the natural language question and a query. Since these templates often reflect the linguistic structure of the questions, structural variations must be included, exponentially increasing the number of possible queries to build.

For an extensive review of existing work, please refer to [1].

2 Our Approach

The main three components that compose this template-based approach are the following:

– a *tagger*, that handles the Named-Entity Recognition task, and is provided by QA^3 [1];
– a *template generator*, that handles the induction of templates of both questions and SPARQL queries by processing a given training set of pairs;
– a *template matcher*, that first performs the ranking of templates for the question posed by the user, and then fills-in the best-matching template.

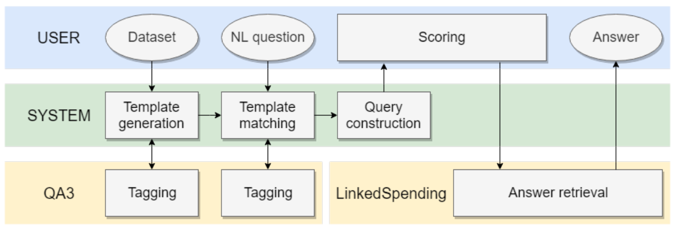

Fig. 1. Our System flow, including tools provided by QA^3 (lower layer) and inputs by the user (upper layer)

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

As shown in Figure 1, a dataset of *(NL question, SPARQL query)* pairs is fed to the *generator*, which, through the *tagger*, outputs a new dataset composed of a list of query templates paired with their respective questions' template. When asked a NL question, the system tags it while the *matcher* extracts similar templates from the templates dataset. From these templates, initial queries and questions can be reconstructed if needed. The scoring of templates w.r.t. a given question is done by the system using semantic similarity, and the system presents the user with successive reconstructed questions until it receives a positive answer, or the list of questions is exhausted. Finally the query, paired with the question chosen by the user, is used to retrieve the answer from the LinkedSpending endpoint[6]. In the following we provide some additional information on each module of the system.

Tagger. The tagger gets the response from a NER service and makes it usable by the system. It sends the end-user question to a service specific for the domain that is being questioned by the user. The service's response usually contains a reference to a datacube, the list of entities found on said datacube, along with the chunk from the question that matched the entity and, optionally, the entity's type.

Generator. The generator creates a template by processing a NL question, a SPARQL query, or both, and removing all the references to the input datacube. It initially gets the list of entities found on the input datacube and the datacube found by the *tagger*. If found in the query, the datacube reference is replaced with the tag (placeholder) `<DATASET>`. The prefixes are replaced by their expanded form, and aliases and the from clause are removed. Next, the expressions and the supported types (numbers, years) found both on the question and the query are replaced with tags depending on the type of data they represent (e.g. `<PROP>` for properties, `<VALUE>` for values, `<YEAR>` for years, `<NUM>` for numbers ecc). Finally, the variables get streamlined to a simple naming convention (`varA`, `varB`, ...), and predicates with no matching found in the datacube are replaced by blank nodes. During the filling of a template, the process is basically inverted. The generator still gets from the *tagger* a list of expressions, which in this case were derived from the English question asked by the end-user. Then the template's tags are replaced by the obtained results, based on their types.

Matcher. The matcher, given a question template, returns a list of compatible query templates. This is done by querying a dataset processed by the generator for the tags in the question template. This list, ordered by the Jaccard Index (calculated between the question template, and the question templates associated to the queries in the dataset), will only contain query templates with an equal or lesser number of tags.

[6] `http://linkedspending.aksw.org/sparql`

3 Running Example

In this section, we detail the various steps performed by our prototype and how they work, using a running example associated with the following natural language question: *What was the total Wandsworth spending in 2013 from the housing department?*.

3.1 Data Preprocessing

Data preprocessing is done by feeding a dataset containing *(NL question, SPARQL query)* pairs to the *generator*. The result is a dataset containing a list of query templates associated with a list of question templates and a list of tags. Take a look at Table 1 for an example of such a pair. Pairs that produce the same query template will be joined into an individual entity that references both question templates.

Table 1. An Example of question/SPARQL pair

Question	**SPARQL query**
"What was the total Wandsworth spending in 2013 from the housing department?"	`select sum(xsd:decimal(?amount)) {` `?obs qb:dataSet ls:wwspending_2013.` `?obs lso:wwspending_2013-Department "housing".` `?obs lso:wwspending_2013-amount ?amount. }`
Question Template	**SPARQL query Template**
"What was the `<AGGRA>` `<DATASET>` from the `<VALUE1>` department?"	`select <AGGRA>(xsd:decimal(?varA)) {` `?obs qb:dataSet <DATASET> .` `?obs [] <VALUE1> .` `?obs <MEASURE> ?varA . }`

3.2 Question analysis & data matching

These tasks are done by the system's tagger with the aid of a domain-specific NER service, in our case the one provided by the QA^3 tagger. The tagger has the following two goals: (i) finding the correct dataset and (ii) matching question terms with the dataset terminology (NER). Missing the correct dataset or failing to find the correct terms will lead to an incorrect answer, so the NER service's accuracy needs to be very high for our system to work properly.

QA^3 proposed an approach based on choosing "the dataset that better *covers* the question, that is, the one that minimizes the portion of the question not referring to elements in the dataset" [1]. The question-to-dataset terms matching is done by first creating an in-memory index of all literals (labels, comments, and values) for each dataset, and normalizing the textual elements that are keys in the indexes and the questions by removing the stop words.
The response from the NER service usually consists of a dataset and a list of entities, as seen in Figure 2.
We then feed the question and the entities found by the tagger to the *generator* to obtain a template (e.g. *"What was the <AGGRA> <DATASET> from the <VALUE1> department?"*).

```
Dataset: wandsworthspending_2013

Chunk:  What was the total
Chunk:  Wandsworth spending in 2013
        S:  ls:/instance/wandsworthspending_2013
        P:  <http://www.w3.org/2000/01/rdf-schema#label>
        O:  "Wandsworth spending (2013)"
Chunk:  from the
Chunk:  housing
        S:  ls:/instance/observation-wandsworthspending_2013-...
        P:  lso:wandsworthspending_2013-Department
        O: "housing"
Chunk:  department
        S:  lso:wandsworthspending_2013-Department
        P:  <http://purl.org/dc/terms/identifier>
        O:  "Department"
```

Fig. 2. Response's structure

3.3 Query Construction

Inverting the data preprocessing process, the query construction takes a template as input, and returns a query. In this process, the templates are filled-in with the data obtained by the tagger from the NL question. In those cases where the same type of tag appears more than once, or we're working with a template with less tags than the NL question, the system creates a query for every combination, in order to cover different structural variation (e.g., the question *"Which class achieved the highest revenue for the Town of Cary, North Carolina?"* processed as *"Which <PROP0> achieved the <AGGRa> revenue for the <VALUE1>, <VALUE2>?"*, will result in the queries shown in Table 2). This outputs a total of $n!$ queries, where n is the number of times the tag appears in the template, unless templates have fewer tags than the NL question. In this second situation, $n!/m$ templates are returned with m denoting the difference between the number of NL question's tags and the number of template's tags.

Table 2. Example of a query's variations

Query template	`select <AGGRa> (?varA) {?obs <SUB1> ?varA }`
Variation A	`select max (?varA) {?obs lso:town_of_cary ?varA }`
Variation B	`select max (?varA) {?obs lso:north_carolina ?varA }`

3.4 Scoring

The scoring process takes place in two different steps. In the *matcher*, we order the list by the Jaccard Index, for which we can also specify a threshold value. If none of the questions in the list has a value over the threshold, the system widens the search to include templates with fewer tags than the ones on the question. This is done in order to avoid matching unrelated templates, a situation that could occur when more entities are tagged than those that are actually needed. After the query construction, since both the *matcher* and the *generator* can return more than a query, our system ranks them by the Jaccard Index (higher scores first) and presents them one-by-one to the user, until the system gets a positive feedback (Figure 3), in which case the system will proceed to the next step. Otherwise it will terminate refusing to provide an answer.

```
Ask a question:
> Which class achieved the highest revenue for the Town of Cary?

Did you meant "Which class earned the most for the Town of
Cary?" [y, N]
> y

Answer: ['https://openspending.org/town_of_cary_revenues/Class/1']
```

Fig. 3. User feedback exploitation (self-learning)

3.5 Answer retrieval & presentation

The reconstructed query associated to the question picked by the user is executed over the LinkedSpending datacube [3]. The answer obtained by running the SPARQL query computed from the SPARQL query template is run against the LinkedSpending endpoint. Unless the answer is empty, the results are shown to the user as the final answer to the processed question, as in Figure 3.

4 Experiments

We used a simple command line interface, i.e. the one shown in Figure 3, where the user poses a question to the system and is then presented with the questions found in the scoring process. The system executes the query associated with the question chosen by the user. We assume that on the scoring step the user always picks the correct question from the proposed ones if available, or a wrong one in case no correct questions are shown. We think this is the expected scenario whenever the user is practically using the prototype, that is, not in an adversarial worst-case setting. If each proposed question is associated with a query that returns an empty answer, we count the user question as not processed.

QALD-6 Dataset. We tested our system with the datasets of the QALD-6 challenge for statistical question answering over RDF datacubes. It consists in a training dataset of 100 questions, and a test dataset of 50. For each question, both datasets contain the correct answer and other metadata.

Results. The results are obtained by feeding the training dataset to the system generator, and then asking the questions from the training and test datasets respectively. The system scored a total of 14 answers on the test dataset, processing 30 out of 50 questions (precision 47% over processed questions). On the training dataset, it managed to process 59 out of 100 questions, and correctly answered to 41 (69% over processed questions). These results are reported in Table 3. Although precision and recall are lower than our award-winning QA^3 system, we believe these are great results considering that no human expert contributed to this result. In fact, measures are reasonable when compared against existing approaches despite our approach being completely automatic. That is, the system will support new queries as long as at least one example is provided in the training set or the user provide a positive feedback. We believe this represents a significant improvement of the state-of-the-art in unsupervised QA on knowledge bases.

Table 3. Results on training and test datasets, obtained by feeding the training dataset to the system

Dataset	N	Processed	Correctly answered
Training	100	59	41
Test	50	30	14

Table 4. Comparison against other QA systems, as reported by the QALD-6 independent competition

System	Processed	Recall	Precision	F-1
This approach	**30**	**0.41**	**0.47**	**0.44**
QA^3	44	0.62	0.59	0.60
CubeQA	49	0.41	0.49	0.45
SPARKLIS (expert user)	50	0.94	0.96	0.95
SPARKLIS (beginner user)	50	0.76	0.88	0.82

References

1. Atzori, M., Mazzeo, G.M., Zaniolo, C.: QA^3: a Natural Language Approach to Question Answering over RDF Data Cubes. Semantic Web **10**(3), 587–604 (2019). https://doi.org/10.3233/SW-180328, `https://doi.org/10.3233/SW-180328`
2. Cyganiak, R., Reynolds, D.: The RDF Data Cube Vocabulary (W3C Recommendation). `https://www.w3.org/TR/vocab-data-cube/` (Jan 2014)
3. Höffner, K., Martin, M., Lehmann, J.: Linkedspending: Openspending becomes linked open data. Semantic Web **7**(1), 95–104 (2016). https://doi.org/10.3233/SW-150172, `http://dx.doi.org/10.3233/SW-150172`
4. Unger, C., Freitas, A., Cimiano, P.: An introduction to question answering over linked data. In: Reasoning Web. Reasoning on the Web in the Big Data Era - 10th International Summer School 2014, Athens, Greece, September 8-13, 2014. Proceedings. pp. 100–140 (2014)