Answering Controlled Natural Language Questions on RDF Knowledge Bases

Giuseppe M. Mazzeo
University of California, Los Angeles
mazzeo@cs.ucla.edu

Carlo Zaniolo
University of California, Los Angeles
zaniolo@cs.ucla.edu

ABSTRACT

The fast growth in number, size and availability of RDF knowledge bases (KB) is creating a pressing need for research advances that will let people consult them without having to learn structured query languages, such as SPARQL, and the internal organization of the KBs. In this demo, we present our Question Answering (QA) system that accepts questions posed in a Controlled Natural Language. The questions entered by the users are annotated on the fly, and an ontology driven autocompletion system displays suggested patterns computed in real time from the partially completed sentence the person is typing. By following these patterns, users can enter only semantically correct questions which are unambiguously interpreted by the system. This approach assures high levels of usability and generality, which will be demonstrated by (i) the superior performance of our system on well-known QA benchmarks, (ii) letting attendees suggest their own test questions, and (iii) accessing an assortment of RDF KBs that, besides the encyclopedic DBpedia from Wikipedia, will include others on specialized domains, such as music and biology.

1. INTRODUCTION

The last few years have seen major efforts toward organizing as RDF knowledge bases (KBs) both general and specialized knowledge. In the first group, we find DBpedia [1] that encodes the encyclopedic knowledge extracted from Wikipedia, and in the second group we have the thousands of projects that cover more specialized domains [2]. While these KBs can be effectively queried through their SPARQL [3] endpoints, the great majority of web users are neither familiar with SPARQL nor with the internals of the KBs. Thus, the design of user-friendly interfaces that will grant access to the riches of RDFKBs to a broad spectrum of web users has emerged as a challenging research objective of great social interest

The importance of this topic has inspired a significant body of previous work, which includes the approaches described in [6, 9, 10] and several others that rely on user-friendly graphical interfaces.

While some of these approaches [6] allow users to enter complex queries through a web browser, Natural Language (NL) interfaces remain the solution of choice when the used devices do not support well a full browser, or when voice recognition is used instead of typing. Translating NL questions into formal language queries represents an old, challenging, and widely studied problem [7, 8, 11], for which a general solution has not been found yet. This is, in fact, a very complex problem, combining several non-trivial subproblems, such as parsing the syntactic structure of the question, mapping the phrases of the question to resources of the KB, and resolving ambiguities. The last problem is quite serious, because syntax often leaves much room for ambiguity, which cannot be resolved without much knowledge about the underlying application domain and understanding the context in which the question is asked.

In order to reduce the complexity of the problem, techniques replacing the 'full' natural language with a *controlled natural language* (CNL) have been proposed. A CNL system restricts the grammar that can be used to input questions, with the objective of making the language (i) 'formal' enough to be accurately interpreted by machines, but still (ii) 'natural' enough to be readily acquired by people as an idiomatic version of their NL. These systems are based on the idea that it is worth giving up the great flexibility and eloquence of the natural language in order to make the questions unambiguous to the machine that can thus produce answers of better accuracy and completeness.

In this demonstration session we will present our system for querying RDF data, called `CANaLI` (acronym for Context-Aware controlled Natural Language Interface). `CANaLI` has been applied to various QA testbeds [4], producing results of superior precision and recall. We will let `CANaLI` answer these testbed questions along with new questions suggested by the conference attendees, as needed to prove the usability and generality of the system. The attendees will thus be able to observe how `CANaLI` guides the users in typing questions, by allowing users to type only questions that are semantically correct w.r.t. the underlying KB, and syntactically correct w.r.t. the grammar of its CNL. Moreover, as soon as the user hesitates with typing, the system suggests correct completions she can select from. This allows people to self-learn `CANaLI` easily and quickly.

This short paper is organized as follows. Section 2 provides an overview of `CANaLI`, describing its basic operation, by means of some examples (Sec. 2.1 and 2.2), the index

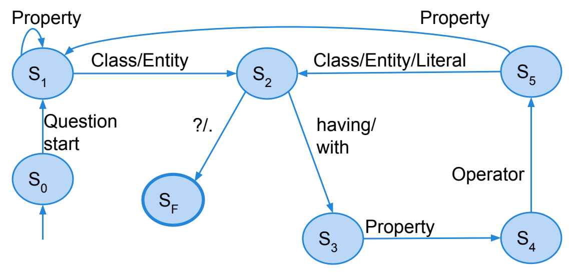

Figure 1: The main states and transitions of the automaton used by `CANaLI`

used to suggest valid tokens (Sec. 2.3), and the system architecture (Sec. 2.4). Experimental results are presented in Section 3. Finally, we describe the demonstration scenario in Section 4.

2. OVERVIEW OF CANALI

`CANaLI` is a system that enables users to enter questions in a controlled and guided way, as a sequence of tokens, that define:

- KB resources: entities, properties, and classes,
- operators (e.g., equal to, greater than, etc.),
- literals: numbers, strings, and dates,
- NL phrases, such as "having", that play a syntactic sugaring role.

Each token is represented by an NL phrase, consisting of one or more words from the application domain, since operators, variables or URLs used in SPARQL are not allowed. `CANaLI` operates on tokens in the style of finite state machines, with (currently) 12 states, including the initial and final state. Despite its simplicity, `CANaLI` is very general, since it can be used with arbitrary RDF KBs, and supports most of the common questions asked by users, including those contained in previous papers and various testbeds (see Section 3).

2.1 Answering Simple Questions

The operation of `CANaLI` can be explained with the help of the transition diagram in Figure 1, and a simple example[1]. For examples, say that the user wants to enter the question: "What is the capital of United States?". When the user starts typing a new question, `CANaLI`'s automaton is in the initial state (S_0), ready to accept tokens representing the question start. In this case, `CANaLI` sees "What is the" and it moves to the state S_1. At S_1, the system can accept a token representing an entity, a property, or a class. In our example, the user enters "capital", that is a property recognized by `CANaLI`. Thus, the system loops back to S_1, ready to accept as next input token another property, entity, or class. In our simple example the user enters "United States", that is an entity, and the system moves to S_2, after recognizing "United States" as an entity with "capital" as valid property. Thus, in order to be consistent with the semantics of the KB, our user must enter entities that have the property "capital", and the system will stop her from progressing any further if that is not the case. Of course, to reach this 'no progress' point the user must have ignored the suggestions that the system had previously generated as valid completions of the typed input. `CANaLI` shows completions under the input area: if the user selects any such completion its text is added to the input area. In S_2 the question mark can be accepted, which marks the end of the question, whereby `CANaLI` moves to the final state S_F and launches the actual query execution. Alternatively, the user can enter conditions, using tokens such as "having", which will be discussed later.

Let us now consider an example involving a chain of properties: "What is the population of the capital of United States?". In this case, at S_1, user inputs the property "population", whereby the system loops back to S_1. `CANaLI` now accepts "of capital" because the capitals have a population, and loops back to S_1, where "of United States" takes us to state S_2 where the question mark completes the processing of the input and launches the query.

Thus, the four basic states S_0, S_1, S_2, and S_F, support a large set of very simple questions asked by everyday users[2]. More complicated but nevertheless common questions are those adding constraints, i.e., query conditions. For instance, assume that the user wants to ask[3]: "What is the capital of countries having population greater than 100 millions?" After the input "What is the capital", has moved us to S_1, `CANaLI` accepts "countries", as a class that has "capital" as a valid property, and moves to S_2. In S_2, `CANaLI` accepts "having", and other uninterpreted connectives used as syntactic sugar, to move to S_3, where it will accept only a valid property. In this case, "population" can be accepted since countries have this property. However, this example illustrates the ambiguity that beset all NL interfaces, no matter how sophisticated their parser is. Indeed, this constraint is also applicable to "capital", since capitals have population too. Clearly every NL system would suffer from the same problem, and only a person who knows that currently no city has more than 100 millions people, might be able to suggest that the question is probably about countries rather than capitals. However, `CANaLI` finesses this inherently ambiguous situation by displaying all alternative interpretations whereby the user has to make a choice. Once the property "population" is accepted, and its context clarified, the automaton moves to the state S_4, that accepts an operator. Thus, the user can input "greater than". The automaton thus moves to state S_5, that accepts the right-hand side of the constraint. In general, the right hand side of a constraint can be an element of the KB or a literal. In our example, only a number can be accepted, since the right-hand side must be of the same type as the left-hand side, "population," which is numerical. Thus, the user enters 100 millions and the automaton moves back to S_2. From this state, the user can specify more constraints, or input the question mark, ending the question.

Examples of constraints using resources of the KB as right-hand side are the following: "Give me the country having capital equal to Washington."[4], "Give me the movies having

[1]Here, the system response is based on the context provided by the question typed so far and the underlying KB, rather than just the current state and last token as a finite state automaton would.

[2]Indeed, the most frequent web questions are definition questions (e.g., What is Ebola?), that are even simpler.

[3]This provides a good example of the broken but effective English now supported by `CANaLI`.

[4]Indeed, the complete automaton of `CANaLI` has also a transition from S_4 to S_2 that allows to implicitly assume the equality operator. This allows to accept questions such as "Give me the country having capital Washington."

director equal to a politician.", "Give me the cities having population greater than the population of Los Angeles.". In all the cases, the token accepted in S_5 is a token whose type is semantically coherent with the property previously accepted in S_3. However, while accepting an entity or a class moves the automaton to S_2, accepting a property (e.g., *population*) moves the automaton to S_1, where the element possessing the property must be specified (e.g., *Los Angeles*).

2.2 More Complex Questions

For the sake of presentation we have shown in Fig. 1 only the states that are most commonly used in queries. In reality `CANaLI` has five more states which support the additional patterns which are discussed next via illustrative examples:

- "Give me the cities having population greater than that of Los Angeles." The use of the pronoun *that* in place of the already used attribute *population*, makes the question more natural than the question where "population" is repeated. However, a special state for handling pronouns had to be added to `CANaLI`.
- "Give me the actors having birth place equal to their death place." The use of the possessive determiner implies that the properties *birth place* and *death place* are related to the same variable. A new state is needed here too, since there is no simple way to the rephrase the question using the grammar accepted by the basic automaton in Fig. 1.
- "Give me the actors having birth date greater than that of their spouse." This question combines the two situations described above.
- "Give me the country having the 2nd largest population". Questions like this require to sort the results by the value of the attribute accepted in a specific state and to set the offset and number of returned results according to the token accepted in another state, i.e., a token such as *the nth greatest* or *one of the nth greatest.*
- "Give me the drugs without specified side effects". This question requires negation. We remark that a token such as *without specified* can not be handled as the tokens like *having*, which defines a comparison between two operands.

2.3 How CANaLI suggests valid tokens

To achieve real-time response, `CANaLI` uses an index supported by Apache Lucene, which handles our tokens as if they were Lucene documents. Every acceptable token is associated with one or more phrases of the natural language. When the user types a string S, a query is performed on the Lucene index, to ensure that the returned tokens (i) have a phrase that matches S, (ii) have a type that is among the acceptable ones, according to the current automaton state and the previous token, and (iii) are semantically correct, according to the KB, as explained below.

To achieve (iii) above, besides indexing the elements of the KB by their label and type (i.e., entity, property, or class), we use two additional fields: *domain of*, and *range of.* The first is needed in cases such as "What is the population of": a token can follow if it is domain of "population" (e.g., "capital", "countries", "United States", etc.). The second is used in cases such as "...having capital equal to": a token can follow if it is range of "capital" (e.g., "birth place", "city", "Washington", etc.). In the case of properties, we also rely on the field *domain*, as needed for cases such as "What is the capital of countries having", that can be followed by a property having as domain the property "capital"[5] or the class "country" (e.g., "population", "language", etc.).

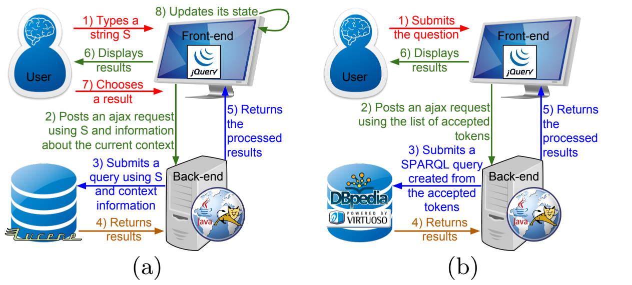
8) Updates its state
1) Types a string S
6) Displays results
7) Chooses a result
User
Front-end
jQuery
2) Posts an ajax request using S and information about the current context
5) Returns the processed results
3) Submits a query using S and context information
4) Returns results
Back-end
Java
Lucene
(a)
1) Submits the question
6) Displays results
User
Front-end
jQuery
2) Posts an ajax request using the list of accepted tokens
5) Returns the processed results
3) Submits a SPARQL query created from the accepted tokens
4) Returns results
Back-end
Java
DBpedia
POWERED BY VIRTUOSO
(b)

Figure 2: `CANaLI`'s architecture & work-flow guiding users in (a) typing questions, (b) retrieving answers.

We created the Lucene index using the elements of the 2014 DBpedia release, using all the entities, and all the properties and classes of the DBpedia KB (those in name space `http://dbpedia.org/ontology/`). Also some classes of the Yago KB and the 20k most frequent raw properties (those in name space `http://dbpedia.org/property/`) were indexed. Furthermore, for all the indexed properties having non-literal range, we created an inverted property, and indexed it. The time needed to create such index, by processing the ~100 millions triples of the English DBpedia, is ~25 minutes using a single machine with 32GB of RAM, starting from the raw files downloaded from the DBpedia website. The obtained Lucene index is ~1.1 GB large and can be easily stored in the main memory of a server, thus assuring a nearly instantaneous response to our search queries.

2.4 The Architecture of CANaLI

Figure 2 shows the architecture of `CANaLI` and its workflow in suggesting and accepting tokens (a), and computing the answers to the submitted question (b). `CANaLI` provides a web client, that uses an autocompleter implemented in *JavaScript*, using *jQuery* libraries. The client keeps track of the input tokens and the current state of the automaton, and when the user types a string S in the auto-completer, an Ajax request is sent to a web server, implemented in Java. The server uses the string S and the status of the automaton to query a Lucene index, that enables to quickly extract the results matching the string S and coherent with the current status of the automaton. Specifically, the suggested tokens must be syntactically coherent, according to the grammar of the language, and semantically consistent, according to the semantics defined by the KB. Completions are returned to the user, and refined as she types more input. Alternatively the user can select one of the suggested completions, and this selection is used to update the question text entered so far and to select the next state of the automaton. When the final state is reached, a request is submitted from the client to server, that uses the sequence of accepted tokens to create a SPARQL query, that is submitted to DBpedia, or the corresponding endpoints for the other KBs, and the results are shown to the user in a user-friendly sniplet format.

[5]Specifically, the range of the property "capital" must be domain for the property "population"

	Proc.	Right	Part.	F proc.	F glob.
CANaLI	**46**	**44**	1	**0.98**	**0.92**
Xser	42	26	7	0.73	0.63
QAnswer	37	9	4	0.40	0.30
APEQ	26	8	5	0.44	0.23
SemGraphQA	31	7	3	0.31	0.20
YodaQA	33	8	2	0.26	0.18

Figure 3: Results on QALD-5 benchmark - Total number of questions: 49.

3. EXPERIMENTAL EVALUATION

A popular set of benchmarks was used to measure the performance of QA systems, i.e., the QALD (Question Answering over Linked Data) benchmarks [4]. The benchmarks consist of sets of NL questions, each associated with a gold standard query in SPARQL, representing the translation of the question. The accuracy of the systems is measured by comparing their results with those obtained by the gold standard queries. We assessed the performances of CANaLI on these benchmarks and the result obtained on each query are presented in [5].

Figure 3 summarizes the results obtained on QALD-5, that consists of 49 questions (the results obtained on the previous benchmarks are equivalent), reporting the results obtained by CANaLI, together with the official results of the participating systems. The columns in the table represent the number of processed questions ("Proc."), the number of questions answered with F-measure equal to 1 ("Right"), the number of questions answered with F-measure strictly between 0 and 1 ("Part."), the average F-measure achieved over the processed questions ("F proc."), and the F-measure over the whole set of questions ("F glob."), assuming 0 as F measure for the non-processed questions.

CANaLI allows to process 46 questions. The 3 questions that could not be processed require two currently unsupported features: (i) sorting by an aggregate function (e.g., "Which musician wrote the most books?"), and (ii) using arithmetic ("What is the height difference between Mount Everest and K2?"). CANaLI provided a completely wrong answer to one question, namely, "Who is the heaviest player of the Chicago Bulls?". The question was input in CANaLI by using the property *team*, while the gold standard query used the UNION of both *team* and *draftTeam*, and the correct result was a player associated to *Chicago Bulls* through the latter property. Therefore, CANaLI missed the correct answer. CANaLI provided a partially wrong answer to another question, "Which programming languages were influenced by Perl?", whose gold standard query used the union of two properties in its constraints (*influence* and *influenced by*). Since CANaLI does not support the UNION operator, only one property was used (*influenced by*), thus missing some of the correct results. Finally, with 44 right answers and a higher F-measure on both the processed and the whole questions, CANaLI proved to be superior to the other systems. Clearly, restrictions imposed by a CNL make an interface like CANaLI a bit less user friendly than full NL interfaces. However, considering that, besides Xser [12], the accuracy of the other full NL systems is far from being acceptable, we believe that an accurate answer is worth a bit extra effort spent in rephrasing the question.

4. DEMONSTRATING CANALI

In this demonstration session, we will exhibit the power, usability and flexibility of CANaLI, by starting from simple questions and moving to more complex ones. The attendees will see how CANaLI guides users in typing questions by allowing to type questions that are only semantically correct w.r.t. the underlying KB: as soon as the typist hesitates or halts even momentarily, the system comes to the rescue by suggesting a list of correct completions the user can select from. This enables people to self-learn CANaLI easily and quickly. In fact, in our demo, after asking the attendee for new questions, we will invite them to enter their questions directly into CANaLI. We will then demonstrate the QA effectiveness of the system by testing its superior precision and recall on complex questions taken from published testbeds that have thwarted the efforts of other QA systems. Finally, we will explain briefly the working of the system, and how the SPARQL queries are generated. This will also allow us to clarify the reasons for the flexibility and generality of the approach, whereby we will show CANaLI in action on several KB, including MusicBrainz and biomedical KBs, and discuss our current work-in-progress to extend it to support temporal questions on the archived history of Wikipedia/DBpedia.

Acknowledgements

The authors wish to thank Maurizio Atzori, Shi Gao, and Matteo Interlandi for very helpful discussions. This research was supported in part by a 2015 Google Faculty Research Award.

5. REFERENCES

[1] http://wiki.dbpedia.org/.

[2] http://linkeddatacatalog.dws.informatik.uni-mannheim.de/.

[3] http://www.w3.org/TR/sparql11-overview/.

[4] http://greententacle.techfak.uni-bielefeld.de/~cunger/qald/.

[5] http://yellowstone.cs.ucla.edu/canali/.

[6] M. Atzori and C. Zaniolo. Swipe: searching wikipedia by example. In *Proceedings of the 21st World Wide Web Conference*, 2012.

[7] B. F. Green, Jr., A. K. Wolf, C. Chomsky, and K. Laughery. Baseball: An automatic question-answerer. In *Papers Presented at the May 9-11, 1961, Western Joint IRE-AIEE-ACM Computer Conference*, 1961.

[8] P. Gupta and V. Gupta. A survey of text question answering techniques. *International Journal of Computer Applications*, 53(4):1–8, 2012.

[9] R. Hahn, C. Bizer, C. Sahnwaldt, C. Herta, S. Robinson, M. Bürgle, H. Düwiger, and U. Scheel. Faceted wikipedia search. In *Business Information Systems, 13th International Conference*, 2010.

[10] L. Han, T. Finin, and A. Joshi. Schema-free structured querying of dbpedia data. In *Proceedings of the 21st ACM International Conference on Information and Knowledge Management*, 2012.

[11] S. R. Petrick. On natural language based computer systems. *IBM J. Res. Dev.*, 20(4):314–325, July 1976.

[12] K. Xu, Y. Feng, and D. Zhao. Answering natural language questions via phrasal semantic parsing. In *Working Notes for CLEF 2014 Conference*, 2014.