CONQUEST: A Framework for Building Template-Based IQA Chatbots for Enterprise Knowledge Graphs

Caio Viktor S. Avila(✉), Wellington Franco, José Gilvan R. Maia, and Vania M. P. Vidal

Department of Computing, Federal University of Ceará, Campus do Pici, Fortaleza, CE, Brazil
caioviktor@alu.ufc.br

Abstract. The popularization of Enterprise Knowledge Graphs (EKGs) brings an opportunity to use Question Answering Systems to consult these sources using natural language. We present CONQUEST, a framework that automates much of the process of building chatbots for the Template-Based Interactive Question Answering task on EKGs. The framework automatically handles the processes of construction of the Natural Language Processing engine, construction of the question classification mechanism, definition of the system interaction flow, construction of the EKG query mechanism, and finally, the construction of the user interaction interface. CONQUEST uses a machine learning-based mechanism to classify input questions to known templates extracted from EKGs, utilizing the clarification dialog to resolve inconclusive classifications and request mandatory missing parameters. CONQUEST also evolves with question clarification: these cases define question patterns used as new examples for training.

Keywords: Interactive Question Answering · ChatBot · Linked Data · Knowledge Graph

1 Introduction

Linked Data technologies made it possible to merge data from many fields, origins, formats, and vocabularies into a unique, uniform, and semantically integrated representation [6], known as Enterprise Knowledge Graph (EKG) [8]. An EKG can be represented by a common vocabulary defined by a closed domain ontology in *OWL*, which allows multiple heterogeneous sources to be accessed simultaneously through queries written in *SPARQL* [9,11]. Competence Questions (CQs) are commonly used to guide the process of ontology construction for EKGs [17]: domain experts list a set of questions that they hope to be answerable, i.e., a CQ can be seen as templates of frequent queries to the EKG. However, creating SPARQL queries is difficult for most users, so natural and intuitive

E. Métais et al. (Eds.): NLDB 2020, LNCS 12089, pp. 60–72, 2020.
https://doi.org/10.1007/978-3-030-51310-8_6

consultation interfaces are of paramount importance in this case [12]. Template-Based Question Answering (TBQA) systems can be valuable within this context: a question Q in Natural Language (NL) is mapped into a well-known *SPARQL* query template Q', so TBQA executes Q' on the EKG in response to Q [4]. Each template contains "slots" to be filled with user-provided parameters, e.g., values for filters, properties, and classes suitable for answering Q. TBQA systems have the advantage of reducing the complex task of interpreting questions in NL to a more straightforward task of classification of intention, which is substantially cheaper than general Question Answering (QA). However, TBQA systems can run into some problems, such as (1) inconclusive template classification or (2) absence of mandatory query parameters in the question. User dialogue is usually employed to disambiguate intent and request parameters, thus generating Template-Based Interactive Question Answering (TBIQA) systems [13]. Conversational systems are popularly known as *chatbots*.

The process of building a TBIQA system can vary greatly depending on its domain, existing tools, and purpose [13]. In this paper, we propose the following standard workflow for the process of creating TBIQA systems on EKG: (1) construction of the templates of questions answerable by the system; (2) construction of the Natural Language Processing (NLP) engine; (3) construction of a question classification mechanism for mapping a question into a template; (4) definition of the system interaction flow; (5) construction of the EKG query mechanism; and (6) construction of the user interaction interface.

Thus, as the main contribution of this paper, we introduce *CONQUEST* (*Chatbot ONtology QUESTion*), a *framework* for creating *chatbots* for the TBIQA task on EKGs represented by a closed domain ontology. *CONQUEST* automates much of the proposed *workflow*, automatically handling steps 2–6. Thus, *CONQUEST* only delegates to the developer the task of building the templates of questions to be answered.

2 Related Work

In [1], the authors present an approach for the automatic generation of query templates supported by TBQA systems. The system has as input a set of pairs of questions in NL and their answers. The questions are then generalized to a template by mapping sets of questions to the same query. As an advantage, the approach allows the composition of patterns for the resolution of complex queries for which complete templates are not known. However, the method depends on the quality of the lexicon used for the highest coverage of templates, and there may be a need to extend the lexicon to specific domains. Besides, the system also does not allow the user to control the templates supported by the system. The authors do not discuss how the system can be made available to users, indicating that this must be addressed per each specific case.

In [3], the authors present a TBQA system over KGs that automatically generates supported questions based on the underlying KG. The process of construction of the questions is carried out based on a small set of query patterns

defined by the authors. The system then constructs the questions supported for each of the predefined patterns, generating variations of them. These questions are then stored in an index that is consulted at run time to identify the most likely question being asked by the user. In addition, the system allows the interactive construction of queries with auto-completing. As a disadvantage, the approach does not allow developers to control the questions supported by the system, which would make it challenging to implement QC support and relevant questions for specific applications.

Medibot [2] is a chatbot in Brazilian Portuguese on a KG in the domain of medicines. *Medibot* has two modes of operation, the first of which is a TBQA, where regular expressions are used to classify the template in which the user's question fits. The approach depends on the manual implementation of regular expressions and the code for querying and building responses, which makes it difficult to reuse and apply in chatbots with a large number of templates. Moreover, the implementation heavily depends on *Telegram* interface.

Many of the existing works in the area of TBQA focus on the automatic generation of templates. However, such approaches limit the developer control over the supported questions, but try to increase the question coverage, which is a positive aspect in the context of consultation on the Web. In business environments, it is expected that the discussions carried out will be limited to a specific set of queries for the performance activities of the company, so it is essential that this set is entirely and correctly covered. Consequently, *CONQUEST* ensures that the developer has full control over the collection of supported templates, ensuring the correctness of the queries that answer them. Besides, most systems do not address how the TBQA service might be made available to users, leaving the developer the task of customizing or creating systems access mechanisms from scratch. *CONQUEST* deals with this by reusing instant messaging services as an access channel to the chatbot, in addition to providing access to the service through a REST API accessible through HTTP requests, all from the execution of a single instance of the chatbot.

3 CONQUEST Framework

The *CONQUEST* framework is composed of the *CONQUEST Trainer* and *CONQUEST Chatbot* modules. The first is responsible for training the necessary components for the TBIQA chatbot being produced. The second is responsible for executing the chatbot, using the components trained to provide a TBIQA service. The source code of the framework can be found in the Github repository[1]. In this paper, the term *developer* will refer to the developer of the chatbot. The term *user* is referring to the end-user who issues questions to that chatbot.

The input given by the developer to the conquest framework is composed of the set of template questions answerable by the system, together with the EKG (ontology + instances) being consulted. The domain ontology provides the structure for the instances, allowing the identification of the type of an

[1] https://bit.ly/2JTE5I0.

instance or parameter value based on the context in which it appears (properties and relationships with which it is linked). A template question whose system is capable of answering is called Question Answering Item (QAI). Each QAI has its *slots* that will be filled with information from user questions, the so-called Context Variables (CVs). A QAI is formally defined as $QAI_{01} = ([QP_1, QP_2, ..., QP_n], SP, RP)$, where: QP_k is a Template (*Question Pattern*) in NL associated to a question, where $1 \leq k \leq n$; SP is a *SPARQL query Pattern*, a template that is employed to retrieve information from the KG; and RP is a *Response Pattern*, a template answer in NL shown to the user.

The following is an example of how the question "What is the maximum price for a given drug in a certain state?" would be represented as a QAI. Where it was given as input only the QP *"What is the maximum price for the medicine $medicine in $state?"*. This template can be provided as input to the system using a JSON file:

```
{"QPs": ["What is the maximum price for the medicine \$medicine in \$state?"],
"SP": "SELECT ?name (MAX(?priceAux) as ?price)  WHERE{
  ?s a <Medicine>;
    rdfs:label ?name;
    <price> ?appliedPrice.
  ?appliedPrice a <Price>;
    <state> $state;
    <value> ?priceAux.
  FILTER(REGEX(?name,$medicine,'i'))}",
"RP":{"header": "",
    "body": "The ?name has a maximum prince of ?price reais",
    "footer":""}}
```

3.1 CONQUEST Trainer

This module is executed only *offline* by the developer. First, two distinct indices are built during **Index Construction**: a class index and a property index. Each of these has information about the domain ontology schema being consulted and are of fundamental importance for the next workflow steps. These indexes have information about labels and definitions of classes and properties, as well as information about properties that relate classes.

The **Processing QAIs** step takes place after the index construction step. This step is divided into three processes. (1) **Consistency check:** for each QAI, all CVs and *Return Variables* (RVs) declared in the query SP are enumerated. Then, for each QP defined in this QAI, the framework checks whether the CVs quoted in that QP belongs to the CV set declared in SP. Likewise, it is checked whether the CVs and RVs quoted in the RP response pattern are contained in the set declared in SP; (2) ***Parsing* and semantic interpretation of a** *SPARQL query Pattern* (SP): The semantic *parsing* of a SP is performed while traversing the SPARQL query tree representation of SP that is generated by

the RDFLib[2] library. The CVs are retrieved during this traversal, together with their type (resource or literal), class, and if this is literal type, their properties, and classes owners. Further details about this complex process will be omitted for the sake of space constraints.

The type of a CV indicates whether it should be replaced by a URI that identifies a resource in the KG (if it is resource type) or a literal. If a CV is inferred to be resource type, then the class attribute will represent the class to which the resource replacing CV must be an instance. On the other hand, if a CV is inferred as being literal, then the class attribute will assume on of the following values: *xsd:string*; *xsd:double*; *xsd:integer*; *xsd:datetime*. In the case of a literal CV, it still has two additional attributes, its "owner property" and "owner class". In the example given, the CV $state has <state> as its "owner property" and <Price> as its "owner class". For the sake of convenience, throughout this article the pairs "owner property" and "owner class" will be regarded as a *string* of the form "Property@Class", which is referred to as "owner pair"; and (3) **Constructon of a vectorial representation (QV) for a QP:** Each QP is mapped into a "representative" vector, which will be called the *Question Vector* (QV). A QV is formed by the concatenation of two other vectors, being the first a *Sentence Vector* (VS) and the second a vector representing the kinds of CVs used in the QP, i.e., a *Context Vector* ($CVec$). Therefore, $QV = VS \oplus CVec$, where $\oplus$ is the concatenation operation over two vectors. The VS is built by resorting to NLP and *Word Embedding* techniques [14]. The first step in building VS from a QP is replacing the CVs markers with *Out of Vocabulary* (OOV) symbols. The second step consists on *string* normalization. The third and last step is computing the very VS vector, so we resort to the NLP *SpaCy* [5] for carrying out this computation. Since the VS vector is built solely based on the text from a QP, VS is considered to be the vector carrying *textual features*. $CVec$ is a vector representing the number of CVs (named entities required) to answer the question encoded by that vector. $CVec$ is a vector of $n+3$ dimensions, where n is the number of owner pairs ("Property@Class") for CVs literals *string*. The other three additional dimensions of $CVec$ refer to the CVs literals from *xsd:integer, xsd:double* and *xsd:datetime* classes. Thus, for each CV existing in QAI, the position of $CVec$ representing the CV type will be incremented by 1. Because of the use of information from the semantic interpretation from KG, $CVec$ is considered the vector representing the *semantic features* of the template.

Training the NER Module is the third step in training stage. The Named Entity Recognition [15] module is responsible for identifying potential candidates in a natural language sentence for CV values. These candidates are used to construct the $CVec$ vector for the given input question. Using NER allows possible values for CVs to be identified directly from the question, eliminating the need to request each CV individually during the consultation time. More specifically, in *CONQUEST*, the NER module is trained to recognize possible values for literal CVs. *CONQUEST* uses a simple regular expression mechanism for identifying entities of numeric types, such as *xsd:integer* and *xsd:double*. For the recogni-

[2] https://rdflib.readthedocs.io/en/stable/.

tion of data type entities (*xsd:datetime*), *CONQUEST* reuses the *dateparser* library [18]. For literals of the *xsd:string* class, *CONQUEST* classifies a candidate for its likely owner pair. This is done by querying terms in an *Apache Solr* [20] index. For each owner pair used in the QAI set (only for xsd:string literals *CV*s), its possible values contained in the KG are fetched. For example, if the owner pair "*ont:name@ont:Person*" is used for a *CV* of type *xsd:string*, then all possible values for the *ont:name* attribute of instances of class *ont:Person* will be retrieved. These retrieved values will be indexed as search keys for the owner pair "*ont:name@ont:Person*". Thus, if the name of a person in the KG is queried, then its owner pair will be returned.

Training the Question Classifier is a cornerstone for our architecture, been the fourth step in the training stage. Based on the promising results obtained recently in the field of Machine Learning (ML) [7] and aiming to address the problem of linguistic variability, we resort to classification ML models due to their high generalization capabilities and versatility. However, using such an approach brings with it a new challenge, the issue of small training sample size [21]. The system is expected to face this problem during the early stages of deploying a chatbot built by *CONQUEST*. To overcome this challenge, *CONQUEST* performs a semantic enrichment step over the input features by using *CVec* as part of the classifier input (*Semantic Features*). For classifier training, the set of *QV*s produced during the stage of processing QAIs is used as the training dataset, with the respective QAI of each *QV* as the output label of the classifier. The default ML model adopted in *CONQUEST* is the Gaussian Naïve Bayes (GaussianNB), which, coupled with the use of *semantic features*, performed as one of the best models tested, both in terms of rating hit rates and time needed for its training.

Saving the trained artifacts is the final step in training stage, where are saved the artefacts: (1) *Ontology Index* that contains ontology schema information so that it can be accessed directly and easily. This information is saved as the indices described previously; (2) the *QA Items* are used in the process of question interpretation, parameter checking and requesting, SPARQL query construction, and response construction; (3) the *NLP Model* is used for natural language processing, including *workflow* for text normalization and segmentation, *word embeddings*, and index (*Apache Solr*) used in NER; (4) and the *Classification Model* that effectively maps a NL question to a QAI.

3.2 CONQUEST Chatbot

An instance of a *CONQUEST Chatbot* is executed during the online stage. This instance accesses the trained artefacts stored in *Persistence* to provide the TBIQA service. Figure 1 depicts the architecture of a *CONQUEST Chatbot*, which is divided into three layers: *User Interface, CONQUEST Core, and Data*.

The **User Interface** layer aims to provide an intuitive and practical interface for users accessing the *chatbot*. To this end, this layer has a set of *APIs* for communicating with instant messenger services, i.e., the *Chat Messenger API*.

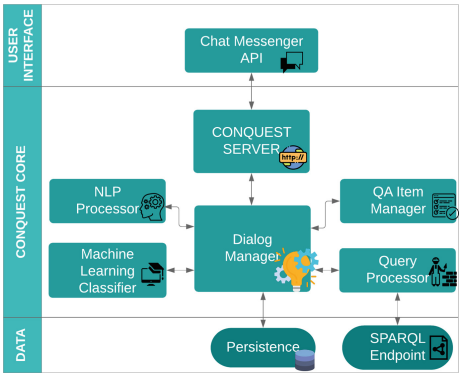

Fig. 1. Architecture for a *CONQUEST Chatbot*.

The **CONQUEST Core** is the main layer of the architecture since it is responsible for processing the questions and their answers. This layer consists of the following six components: (1) ***CONQUEST Server***, responsible for providing *chatbot* services through HTTP requests, acting on the boundary between the interface layer and the system core. This component gets HTTP requests as input, forwarding them to *Dialog Manager*, and finally returning the respective responses to the user. The *CONQUEST Server* can be accessed either through an IM service (e.g., Telegram), or directly via HTTP requests, thus being available in a wide range of channels simultaneously; (2) ***Dialog Manager*** is the central module regarding the execution of a *CONQUEST Chatbot*. The *Dialog Manager* is responsible for managing the request processing flow, exchanging information between components, and managing the dialog flow; (3) ***NLP Processor*** is responsible for taking a question Q in natural language and converting it to a vectorial representation QV. The following sequence of steps is performed for this purpose: (I) normalization and *tokenization* of Q; (II) Identification of named entities contained in the sentence by the NER component. The first type of entities looked for are the literals of the *xsd:string* class. To do this, the sliding window process of a n-gram [19] is performed over the *tokens* contained in Q. The starting value of n is equal to the number of *tokens* in Q, where the window slides from left to right, one *token* at a time, and decreasing in size by 1 each time it reaches the end of the *tokens* sequence. During this process, each *n-gram* is queried against the *Solr* index, and if it is contained, then it is removed from the sequence. Subsequently, entities like *xsd:datetime* and numeric types are sought as defined in Sect. 3.1; (III) Computation of SV vector for Q; (IV) Computation of the $CVec$ vector for Q, using the named entities found in step II; and finally (V) calculating the QV representation of Q; (4) ***Machine Learning Classifier*** receives the QV vector representation of Q as input and then returns the confidence classification level for each QAI; (5) ***QA Item Manager*** retrieves information about the classified QAI. This information is used for (a) determining the CVs needed for the question by filling this information automatically

or requesting it from users, (b) retrieving the SPARQL query template (SP) to be used, and (c) retrieving the response pattern (RP) to be generated; and (6) ***Query Processor*** receives as input a *template* SPARQL SP and its set of filled CVs. As a result, this module performs the actual assembling and execution of the query in *Endpoint SPARQL*; Finally, the query result is returned to the *Dialog Manager*, which generates the natural language response based on the RP template.

The third and last layer is the **Data Layer**, which is responsible for storing the *chatbot* knowledge, which refers to both learned artifacts during the training phase and the EKG being queried. This layer is divided into two components: (1) ***Persistence*** holds the knowledge obtained in the offline stage. This knowledge is retrieved by *Dialog Manager* and then distributed to the other *CONQUEST Core* modules so that they can perform their tasks. Moreover, *Persistence* is also used to store *Interaction State* that saves the current state of a user interaction during the *chatflow*. The state of the interaction consists of the current point of interaction following Fig. 2 and the information acquired so far (e.g., question given as input, classified QAI, values for CVs and other information for a coherent dialogue). This ensures that *chatbot* performs long interactions consistently; (2) ***SPARQL Endpoint***, which is external to the system, so it is accessed using HTTP requests to execute SPARQL queries. The current implementation resorts to the *SPARQL Wrapper* [16] library, which is responsible for handling requests and responses to this *endpoint*.

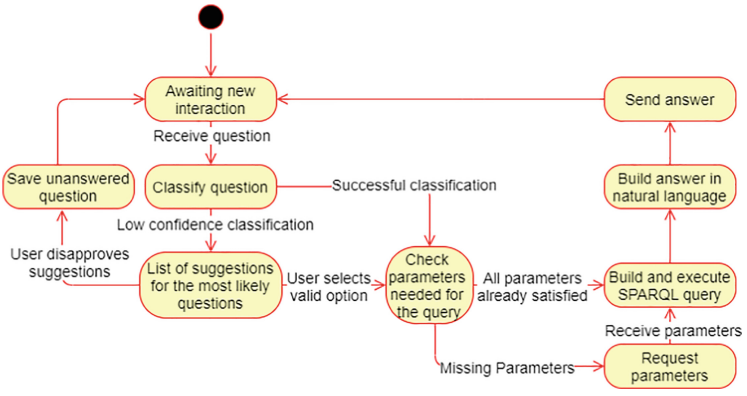

Fig. 2. *Chatflow* followed by a *CONQUEST Chatbot*.

CONQUEST Chatbot's Chatflow is depicted by Fig. 2 and can be summarized as: the *chatbot* receives the question in NL, classifying it for a QAI; if this classification is not possible, then the *chatbot* performs the disambiguation dialog; after a successful question classification, the *chatbot* checks to see if all CVs have been filled in, prompting them to the user otherwise; finally, the *chatbot* consults the EKG and returns the response to the user. In the case of

confirmation of the clarification dialog, the question given as input is added as a new Question Pattern (QP) to be considered in classifier training.

4 Results and Discussions

A qualitative assessment was carried out to assess the impact of using *CONQUEST*. For the sake of comparison, we re-implemented *MediBot* [2], a chatbot published recently that fits our requirements since it adopts an TBIQA perspective to operate over KG.

The **Template Construction** process was shown to be quite natural and required the developer to input only a few variations of the NL question. The JSON file containing the QAIs used and the data needed to deploy an instance of our implementation of *MediBot* on top of *CONQUEST* are publicly available[3]. The example is given in Sect. 3 is an example of how one of the QAIs could be written, and it will be considered in the discussions that follow in this section.

In **NLP Engine Construction** step, the developer should only select the language supported by his chatbot being produced. *CONQUEST* uses the *Spacy* library for NLP, which supports more than 53. However, support for each language is at a different stage. At the same time, the library achieves great results for English, the same cannot be said for Brazilian Portuguese (language supported by MediBot). Because of this, we used the 100-dimensional GloVe model produced by [10] as the *Word Embedding*. This model was loaded into *SpaCy*, thereby leveraging the entire processing pipeline of this library.

The **Template Classifier Construction** step is transparent to the developer, with CONQUEST already having a default classification model. Experiments were carried out to select the best model and to assess the impact of using semantic features on this task. For these experiments, the 8 query templates answered by *MediBot* presented by the authors were implemented. As a set of training and validation, 10 variations of the question in NL were used for each template, using cross-validation with parameter $CV = 5$. For the test set, 5 examples of variations for each template different from those used in the training/validation stage. The results presented are the average of the tests performed 10 times. The script with the experiments can be found at the link[4].

Table 1 summarizes the results of main trained models without and using the *Semantic Features* proposes in this work. In the first case, the best model was the Multilayer Perceptron (MLP) classifier with two hidden layers. This MLP model achieves a score of 0.926 on the F1 metric, which is considered a good result. However, the time required for model training took around 0.229 seconds, so this is one of the slowest models for training. Since the *chatbots* produced by *CONQUEST* use the questions given at runtime as new training examples, this results in constant growth of the *dataset*. Consequently, the cost for model training is of critical importance. The use of *Semantic Features* generally presents significant improvements in the evaluated models. In this case, it is important to

[3] https://bit.ly/2T9Pbhu.
[4] https://bit.ly/2I0WguG.

highlight the performance improvements of the *GaussianNB* model, which has achieved the best performance in all measured aspects. When comparing the best trained model with the use of *Semantic Features* against the best without them (MLP with two hidden layers), it is possible to see a slight improvement of about 5.075%, which can already be considered a promising result. When comparing the results under the light of the F1-score for the *GaussianNB* model without and with *Semantic Features*, it is possible to notice an improvement of about 38.014 %, which configures a great improvement overall. The real improvement comes from comparing the training time taken by the two best models, *MLP with two hidden layers* and *GaussianNB*: there is a 98.280% reduction in the required time to training, which means that the first model takes about 58 times longer in training than the second. The results of the selected model (GaussianNB with *Semantic Features*) in the test set was 0.979 for Precision, 0.975 for Recall and 0.974 for F1.

Table 1. Results of the model evaluation experiment.

Classifier	Without semantic features				With semantic features			
	Precision	Recall	F1	Time (secs)	Precision	Recall	F1	Time (secs)
GaussianNB	0.772	0.712	0.705	00.023068	**0.983**	**0.975**	**0.973**	**00.003952**
LogisticRegression	0.8	0.787	0.764	01.301347	0.958	0.937	0.933	00.040490
SVC linear	0.916	0.875	0.870	00.048965	**0.983**	**0.975**	**0.973**	00.007134
DecisionTreeClassifier	0.545	0.575	0.534	00.056715	0.858	0.875	0.860	00.007359
MLPClassifier 2 layers	**0.941**	**0.912**	**0.926**	00.229768	0.966	0.962	0.96	00.176730
Nearest Neighbor	0.707	0.675	0.657	**00.004416**	0.879	0.875	0.86	00.006486
GaussianNB + Logistic (Soft Voting)	0.772	0.712	0.705	00.218400	**0.983**	**0.975**	**0.973**	00.110563

CONQUEST's **Interaction Flow** frees the developer from dealing with the scheduling of the conversion flow using techniques such as state machines, conversation scripts, etc. In this example (Fig. 3), the user formulates the question in a manner considerably different from the known template. Consequently, the *chatbot* attempts to resolve user intent by displaying the QP template by replacing the CVs values found by NER in the original question. Having the suggestion confirmed by the user, a new example is added for this QAI after the CVs values are replaced by their corresponding identifiers (e.g., "buscopan" is replaced by $medicine). However, the *chatbot* realizes that the value for CV $state is still missing, thus using the inferred type of CV to make its request. Finally, after substituting the values of CVs in SP and executing it in *endpoint SPARQL*, then the *chatbot* returns the response following RP.

The **Query Engine Construction** step is fully automatic. The SPARQL query pattern (SP) passed in QAI is used to build the actual query to be executed on the EKG. CVs markers present in SP are filled with the parameters

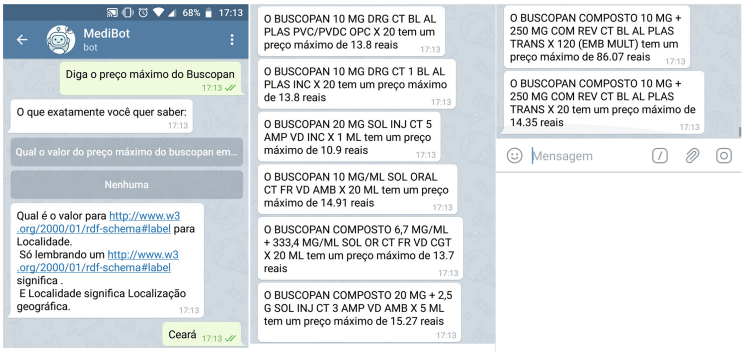

Fig. 3. Using the clarification dialog in *Telegram*.

passed by the user at query time. In the current example, $medicine and $state in *SP* are replaced by "Buscopan" and "Ceará", respectively. *CONQUEST* builds the final NL response by replacing the values of the output variables in the response pattern (*RP*) in QAI with the values returned by executing the query in the EKG. In the example, *?name* and *?price* in the "body" of *RP* are replaced by the values of the variables *?name* and *?price* for each item of the query response.

User Interface was tested with instant messaging application (*Telegram*) and directly via HTTP requests. In the first case, immediate reuse eliminates the need for the installation of new apps and adaptation by the final user. In the second case, external applications can be integrated into larger services, such as existing chatbots built with commercial environments (e.g., *Dialogflow, chatfuel, etc.*), where *CONQUEST* can provide only the specific TBIQA skill for a "larger" chatbot. Finally, *CONQUEST* allows the same instance of a chatbot to be shown in different channels running from the same code, which facilitates maintenance and service increment.

5 Conclusions

CONQUEST framework automates much of the process of building TBIQA *chatbots* on EKGs, where supported templates must be provided as input and dialogue are used to address the problems of inconclusive classification and the lack of parameters in the question. *CONQUEST* resorts to machine learning to acquire new ways in which the same question can be accomplished, which allows the chatbot to evolve with usage. Unlike other works in the field, *CONQUEST* allows complete control of the questions supported, which guarantees support for complex and specific needs, e.g., *Competency Questions*, and also addresses the problem of access to the built service, allowing support through multiple channels simultaneously. As future work, we plan to address the automatic generation of query templates to answer simple questions, so developers focus their efforts on complex and challenging templates.

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 (2017)
2. Avila, C.V., et al.: MediBot: an ontology based chatbot for Portuguese speakers drug's users. In: Proceedings of the 21st International Conference on Enterprise Information Systems. ICEIS, vol. 1, pp. 25–36. INSTICC, SciTePress (2019). https://doi.org/10.5220/0007656400250036
3. Biermann, L., Walter, S., Cimiano, P.: A guided template-based question answering system over knowledge graphs. In: Proceedings of the 21st International Conference on Knowledge Engineering and Knowledge Management (2018)
4. Diefenbach, D., Lopez, V., Singh, K., Maret, P.: Core techniques of question answering systems over knowledge bases: a survey. Knowl. Inf. Syst. **55**(3), 529–569 (2017). https://doi.org/10.1007/s10115-017-1100-y
5. Explosion AI: Industrial-strength natural language processing (2019). https://spacy.io
6. Frischmuth, P., et al.: Linked data in enterprise information integration. In: Semantic Web, pp. 1–17 (2012)
7. Géron, A.: Hands-On Machine Learning with Scikit-Learn and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems. O'Reilly Media, Inc., Sebastopol (2017)
8. Gomez-Perez, J.M., Pan, J.Z., Vetere, G., Wu, H.: Enterprise knowledge graph: an introduction. Exploiting Linked Data and Knowledge Graphs in Large Organisations, pp. 1–14. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-45654-6_1
9. Jin, G., Lü, F., Xiang, Z.: Enterprise information integration based on knowledge graph and semantic web technology. J. Southeast Univ. (Nat. Sci. Ed.) **44**(2), 250–255 (2014)
10. Hartmann, N., Fonseca, E., Shulby, C., Treviso, M., Rodrigues, J., Aluisio, S.: Portuguese word embeddings: evaluating on word analogies and natural language tasks. arXiv preprint arXiv:1708.06025 (2017)
11. Heath, T., Bizer, C.: Linked data: evolving the web into a global data space. Synth. Lect. Semant. Web Theory Technol. **1**(1), 1–136 (2011)
12. Kaufmann, E., Bernstein, A.: How useful are natural language interfaces to the semantic web for casual end-users? In: Aberer, K., et al. (eds.) ASWC/ISWC - 2007. LNCS, vol. 4825, pp. 281–294. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-76298-0_21
13. Konstantinova, N., Orasan, C.: Interactive question answering. In: Emerging Applications of Natural Language Processing: Concepts and New Research, pp. 149–169. IGI Global (2013)
14. Li, Y., Yang, T.: Word embedding for understanding natural language: a survey. In: Srinivasan, S. (ed.) Guide to Big Data Applications. SBD, vol. 26, pp. 83–104. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-53817-4_4
15. Nadeau, D., Sekine, S.: A survey of named entity recognition and classification. Lingvist. Investig. **30**(1), 3–26 (2007)
16. RDFLib: SPARQL Wrapper SPARQL endpoint interface to Python (2019). https://rdflib.github.io/sparqlwrapper/. Accessed 26 Nov 2019

17. Ren, Y., Parvizi, A., Mellish, C., Pan, J.Z., van Deemter, K., Stevens, R.: Towards competency question-driven ontology authoring. In: Presutti, V., d'Amato, C., Gandon, F., d'Aquin, M., Staab, S., Tordai, A. (eds.) ESWC 2014. LNCS, vol. 8465, pp. 752–767. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-07443-6_50
18. Scrapinghub: dateparser date parsing library designed to parse dates from HTML pages (2019). https://pypi.org/project/dateparser/. Accessed 25 Nov 2019
19. Shishtla, P.M., Pingali, P., Varma, V.: A character n-gram based approach for improved recall in Indian language NER. In: Proceedings of the IJCNLP-2008 Workshop on Named Entity Recognition for South and South East Asian Languages (2008)
20. Smiley, D., Pugh, D.E.: Apache Solr 3 Enterprise Search Server. Packt Publishing Ltd., Birmingham (2011)
21. Yang, P., Hwa Yang, Y., Zhou, B.B., Zomaya, A.Y.: A review of ensemble methods in bioinformatics. Curr. Bioinform. **5**(4), 296–308 (2010)