NL$_2$CM: A Natural Language Interface to Crowd Mining

Yael Amsterdamer, Anna Kukliansky and Tova Milo
Tel Aviv University, Tel Aviv, Israel
{yaelamst,annaitin,milo}@post.tau.ac.il

ABSTRACT

The joint processing of *general data*, which can refer to objective data such as geographical locations, with *individual data*, which is related to the habits and opinions of individuals, is required in many real-life scenarios. For this purpose, *crowd mining* platforms combine searching knowledge bases for general data, with mining the crowd for individual, unrecorded data. Existing such platforms require queries to be stated in a formal language. To bridge the gap between naïve users, who are not familiar with formal query languages, and crowd mining platforms, we develop NL$_2$CM, a prototype system which translates natural language (NL) questions into well-formed crowd mining queries.

The mix of general and individual information needs raises unique challenges. In particular, the different types of needs must be identified and translated into separate query parts. To account for these challenges, we develop new, dedicated modules and embed them within the modular and easily extensible architecture of NL$_2$CM. Some of the modules interact with the user during the translation process to resolve uncertainties and complete missing data. We demonstrate NL$_2$CM by translating questions of the audience, in different domains, into OASSIS-QL, a crowd mining query language which is based on SPARQL.

1. INTRODUCTION

A useful distinction between information needs in real-life tasks classifies such needs into two types: *general information* needs, which involve data not tied to a particular person, such as the locations of places or opening hours; and *individual knowledge* needs, which concerns individual people, e.g., their actions and opinions. The first type of data can often be fetched from standard databases or knowledge bases, whereas obtaining individual data about the members of a certain population often requires posing questions to (a sample of) the relevant people. As an example, based on a real question posted in a travel-related forum, consider a group of travelers who booked a hotel in Buffalo, NY. The group members may be interested in the answer to the question "What are the most interesting places near Forest Hotel, Buffalo, we should visit in the fall?" Answering this question requires processing mixed data: the sights in Buffalo and their proximity to Forest Hotel is a general data that can be found, e.g., in a geographical ontology such as LinkedGeoData;[1] the interestingness of places is an individual opinion; and which places one should visit in the fall is an individual recommendation. As another example, consider a dietician wishing to study the culinary preferences in some population, focusing on food dishes rich in fiber. While nutritional facts can be found in a (general) knowledge base, the eating habits of people are individual information.

The example tasks mentioned above could be performed using standard tools such as web search or forums. However, these tools have shortcomings in processing mixed general and individual knowledge: search engines are very efficient nowadays, but cannot ask people to provide individual data that is not already recorded in the search engine's repository, when such data is required; web forums allow fetching individual data from web users, but do not search relevant general data in knowledge bases; and both tools may incur post-processing efforts to the user, in manually analyzing their text-based output, filtering irrelevant results, aggregating answers and identifying consensus, etc. This calls for a *hybrid and automated approach*, which involved searching a general knowledge base, asking and analyzing the answers of people, and jointly processing the results.

Such an alternative approach is studied in our recent work about *crowd mining* [1, 2]. In [2], a novel query language was defined, OASSIS-QL, whose syntax splits individual information needs from general ones. This allows OASSIS-QL queries to be evaluated by an efficient, crowd-powered query engine, such that general query parts are evaluated against a knowledge base (ontology) and the individual parts are evaluated with the help of the crowd of web users.

While existing crowd mining tools such as OASSIS-QL are a major step towards processing mixed general and individual information needs, one cannot expect naïve users (like the travelers in our first example above) to write complex queries in a formal language. To overcome this, we introduce NL$_2$CM. This prototype system allows users to express their requests in natural language (NL), identifies and isolates specific general and individual information needs, and automatically translates these needs into well-formed queries. We exemplify the operation of this framework for OASSIS-QL, which is evaluated over RDF-modeled data. However, the

[1] LinkedGeoData. http://linkedgeodata.org/

SIGMOD'15, May 31–June 4, 2015, Melbourne, Victoria, Australia.

http://dx.doi.org/10.1145/2723372.2735370.

principles we develop and the modular architecture of NL$_2$CM, may cater for other crowd mining platforms that support query languages over, e.g., relational or XML data.

The problem of developing NL interfaces to query engines has been studied in previous work for queries over general, recorded relational/XML/RDF data (e.g., [5, 6, 7, 12]). However, the mix of individual and general information needs in the scenarios we consider add new challenges to the problem. For instance, individual and general information needs may be mixed in an intricate manner in an NL question. Until now, no existing tools could account for distinguishing the different types of needs, which is necessary for the translation process. Existing NL tools can identify only individual expressions of sentiments and opinions, but do not account, e.g., for individual habits. Moreover, naïve approaches, such as checking which parts of the query do not match to the knowledge base, cannot facilitate this task since most knowledge bases are incomplete.

Our solution to these challenges consists of (i) a modular architecture, reusing existing NL tools where possible; (ii) a newly-developed module for distinguishing individual information needs from general ones; and (iii) seamless integration of individual and general query parts into a well-formed query. The translation process can be outlined as follows. First, off-the-shelf NL tools are instrumented for parsing the free text and converting it into well-defined data structures, which represent the semantic roles of text parts. These structures then undergo decomposition into their general and individual parts by our new module, which employs means of declarative pattern matching and dedicated vocabularies. Each part is separately processed, and in particular, an existing *General Query Generator* is used to process the general query parts, whereas the individual parts are processed by new modules. The processed query parts are integrated to form the final output query. NL$_2$CM implements these ideas and further *interacts with the user* in order to deal with uncertainties and missing details (e.g., "Buffalo" is ambiguous) in the user request.

We next provide a brief overview of the demonstration of NL$_2$CM and of related work. Section 2 explains the design and the techniques used in the translation framework. The implementation details of these ideas, along with the interaction of the user with the system are explained in Section 3. An elaborate demonstration scenario is described in Section 4.

Demonstration Overview. We will demonstrate the operation of NL$_2$CM in three stages, to gradually introduce it to the audience members. First, we will demonstrate the translation of real-life NL requests collected from web forums into OASSIS-QL queries, and observe the correspondences between different query parts and parts of the original NL sentence. Second, we will invite the audience members to contribute questions on any topic that interests them. These NL requests will also be translated into OASSIS-QL, and we will execute some of the queries via the OASSIS query engine, to gain further intuition about the semantics of the generated queries and their results. Last, we will demonstrate the interaction of NL$_2$CM with the user in cases when either rephrasing the question or adding information is required. In parallel, we will gain an insight into the system operation through its administrator mode screen. See Section 4 for full details.

```
SELECT VARIABLES
WHERE
    {$x instanceOf Place.
     $x near       Forest_Hotel,_Buffalo,_NY}
SATISFYING
    {$x  hasLabel "interesting"}
    ORDER BY DESC(SUPPORT)
    LIMIT 5
   AND
    {[]  visit $x.
     []  in   Fall}
    WITH SUPPORT THRESHOLD = 0.1
```

Figure 1: Sample OASSIS-QL Query, $\mathcal{Q}$

Related Work. A recent line of work studies the use of crowdsourcing platforms for performing different data processing tasks with the help of the crowd (e.g., [3, 4, 9, 13, 14]). The idea of *ontology-based crowd mining* was originally introduced in [1], and extended to support the evaluation of user-specified declarative queries in [2], by the OASSIS system. NL$_2$CM forms an important enabling tool for crowd mining systems like OASSIS, since it makes them accessible to the public, and allows any user to formulate even complex queries in an intuitive manner. Previous work includes a line of NL interfaces to declarative systems, e.g., [5, 6, 7, 12], many of which employ interactive request refinement as is done in NL$_2$CM. In particular, FREyA [5] translates NL into SPAQRL, and interacts with the user to resolve unrecognized or ambiguous terms. We employ it for processing general information needs, as described in Section 3. However, to our knowledge, no prior work can account for NL requests concerning individual data.

2. TECHNICAL BACKGROUND

The process of automated query translation can be viewed as converting between two types of data representations, namely, the source and target languages. In our setting, we use NL parsing tools to generate a standard representation of the source NL text (see below). This representation allows processing the text to detect and isolate the general and individual query parts, and eventually convert them to the representation of the target query language.

Next, we review the design and techniques used in the NL$_2$CM translation framework. As explained in the Introduction, this framework is modular and can be adapted to support different query languages. To make the discussion concrete, we describe here the translation of NL to OASSIS-QL, which is also used in our demonstration scenario. We will mention throughout the section which parts of the solutions are specific to OASSIS-QL, and changes that can be made to support other languages. For background, we start by briefly explaining the syntax and semantics of OASSIS-QL, a full description of which can be found in [2]. Then, we describe the modular architecture of NL$_2$CM, and the interaction between the framework modules.

2.1 Query Language Overview

As mentioned in the Introduction, OASSIS-QL queries are evaluated against an ontology of general knowledge as well as the individual knowledge of the crowd. An RDF model is used to represent both types of knowledge, and thus, the syntax of OASSIS-QL is defined as an extension of SPARQL, the RDF query language, to crowd mining.

An OASSIS-QL query has three parts: (i) a SELECT clause, which defines the output of the query; (ii) a WHERE clause,

which is evaluated against an ontology; and (iii) a SATISFYING clause, which defines the data patterns to be mined from the crowd. The returned output is composed of data patterns that capture the *relevant* and *significant* habits and opinions of the crowd. We briefly review below the OASSIS-QL syntax, using the example question "What are the most interesting places near Forest Hotel, Buffalo, we should visit in the fall?". Figure 1 displays the query $\mathcal{Q}$, which is the translation of this NL question into OASSIS-QL.

The SELECT clause (line 1) specifies that the output should be *significant variable bindings*, i.e., bindings to the query variables which yield significant data patterns. In this example, $\mathcal{Q}$ contains a single variable $x, and its bindings to places in Buffalo that match the query, e.g., the Delaware Park and Buffalo Zoo may be returned. The language also allows projecting the results over a subset of the variables.

The WHERE clause (lines 2-4) defines a SPARQL-like selection query on the ontology. It consists of triples of the form entity-relation-entity, which correspond to the RDF ontology triple structure. The ontology triples represent general knowledge *facts*, e.g., the triple {Delaware_Park instanceOf Place} signifies that Delaware Park is a name of a place. The triples in the query WHERE clause may further contain variables (e.g., $x), and are used to perform a selection over the ontology facts. This returns all variable bindings such that the resulting set of triples is contained in the ontology.

The SATISFYING clause (lines 5-12), to which we apply the previously found variable bindings, defines the data patterns (fact-sets) to be mined from the crowd. Each pattern appears in a separate subclause in curly brackets (lines 6 and 10-11). Consider, e.g., the fact-set {[] visit $x. [] in Fall}, where [] stands, intuitively, for "anything". Combined with the binding $x $\mapsto$ Delaware_Park, this fact-set corresponds to a habit of visiting Delaware Park in the fall. The system can then ask crowd members how frequently they engage in this habit, if at all. Intuitively, the *support* of a data pattern captures a habit frequency or a level of agreement to a statement, aggregated from the answers of several crowd members. In OASSIS-QL, one can select the patterns with the k-highest(lowest) support values, using ORDER and LIMIT (lines 7-8). Alternatively, one can set a minimal support threshold for the patterns (line 12). Note that neither the limit nor the support threshold values are specified in our example NL request. In this common scenario, the system can either use default values that are pre-configured at the system administrator level, or interact with the user to obtain them (See Section 4.1).

2.2 NL Parsing Modules

We now proceed to describe the modules of the translation framework. To parse the input user request, two standard NL tools are used: *POS (Part-Of-Speech) tagger* and *dependency graph parser*. A Part-of-Speech (POS) tagger assigns a linguistic category to every meaningful unit in a given text [8]. E.g., in the running example question, "Buffalo" is a *proper noun* and "interesting" is an *adjective*. A dependency graph parser produces a directed acyclic graph (typically, a tree), whose nodes correspond to meaningful units and its edges are labeled by semantic relationships between these units [8]. E.g., an edge labeled "subject" may connect a verb node with its grammatical subject.

The dependency graph along with the POS tags serve for higher-level data processing by the subsequent modules.

2.3 Individual Expression Detection

In order to distinguish the individual and general parts of the parsed NL request, we use an *Individual eXpression (IX) Detector*. This module identifies and extracts the individual parts from the dependency graph.

As mentioned in the Introduction, no existing tools can account for IX detection. Some previous work considers identifying expressions of sentiment or subjectivity in texts (e.g., [10]), but these expressions are only a subset of individual expressions. For instance, they do not capture individual habits such as where people visit, what they cook, etc. We thus develop a dedicated solution for IX detection, based on a general technique in NL processing, namely, pattern matching. This technique is simple and transparent (in comparison with, e.g., machine learning), and by defining the IX detection patterns in a declarative manner, allows a system administrator to easily manage, change or add the predefined set of patterns. By our analysis of user requests with individual parts, we have identified different IX patterns and classified them into the three following types.

- **Lexical individuality** stems from a term in the text. E.g., "interesting" conveys an individual opinion.
- **Participant individuality** stems from participants or agents in the text that are relative to the person addressed by the request. E.g., "you" in "Where do *you* visit in Buffalo?" is an individual participant.
- **Syntactic individuality** stems from the sentence syntax. For example, in "Obama *should* visit Buffalo", the verb auxiliary "should" denotes the speaker's opinion.

Every IX detection pattern of the above types matches a *connected subgraph* within the dependency graph (which corresponds to a set of semantically related text units). We define IX detection patterns in a SPARQL-like syntax, in terms of the POS tags; the dependency graph edges; and dedicated vocabularies. For lack of space, we will not explain here the detection pattern syntax but give an intuition about it via an example: the following simple pattern detects IXs with a verb that has an individual subject, a particular case of an individual participant.

```
1 $x subject $y
2 filter(POS($x) = "verb" && $y in V_participant)
```

Intuitively, the query above selects a verb ($x) which is connected by a subject-labeled edge to its subject ($y). The filter statement enforces that $x is a grammatical verb by its POS tag, and that $y is in the vocabulary V_participant, dedicated to individual participants.

More generally, individuality in NL can be very complex, and almost every expression can appear in a (non-)individual context. Thus, the IX detection patterns and vocabularies should be designed to capture expressions which appear in individual contexts with *high probability*.

2.4 General Query Generator

The different query parts should now be converted into the basic building blocks of the query. As mentioned in the Introduction, since the problem of translating NL to queries over general data has been studied in previous work (e.g., [5, 6, 7, 12]), it is preferable to reuse an existing tool for this task. Thus, we embed in the translation framework an off-the-shelf General Query Generator for processing the general parts of the query, i.e., the parts of the dependency graph

which were not identified as IXs. In the particular case of translation to OASSIS-QL, the syntax of the general query parts is based on SPARQL, and thus a SPARQL Query Generator such as [5] can be used. From the Generator's output, we can extract the SPARQL triples that will form the WHERE part of the target OASSIS-QL query. The crux in reusing an existing Query Generator lies in having it translate *only the general request parts* rather than the full request. The way to resolve this issue may depend on the choice of Query Generator. The particular solution used in our implementation is described in Section 3.

2.5 Individual Query Parts Creation

While the general parts of the query are translated by a General Query Generator, translating the IXs is done by a new module. Unlike the Query Generator, this module cannot rely on aligning the request parts to the ontology, since these parts do not correspond to an ontology. Instead, a mapping is defined from certain grammatical patterns within the IXs into query parts, according to the syntax of the query language. In the case of OASSIS-QL they are mapped to query triples. For example, consider the IX "places we should visit", which corresponds to a subgraph where "we" and "places" are, respectively, the subject and object of "visit". A mapping would then generate the triple {[] visit $x}, where $x corresponds to "places"; and [] corresponds to "we", meaning that this individual participant is projected out, which is necessary for aggregating the answers of different crowd members about the same habit.[2]

2.6 Query Composition

The last module of the architecture combines the generated individual query parts with the general query parts, created by the Query Generator. In the case of OASSIS-QL, forming the query mainly means creating the subclauses of the SATISFYING clause, aligning the variables of the WHERE and SATISFYING clause, and creating the SELECT clause. Intuitively, the subclause creation phase ensures that every subclause defines a data pattern to be mined from the crowd that corresponds to a single event or property. For example, the visit of Buffalo, in our running example, is described as occurring in the fall and thus their corresponding triples are put in the same subclause (lines 10-11 in Figure 1). For each subclause, either a support threshold or a top/bottom-k support selection is defined. The alignment of variables is done such that every reference to a particular term in the original sentence is represented by an occurrence of the same variable. By default, the generated SELECT clause is simple and does not project out any variables (as in Figure 1), which means that the query returns bindings to all the variables. In Section 4.1 we discuss interacting with the user to determine which variables should be projected out.

3. IMPLEMENTATION DETAILS

To complete the picture, we now provide the details of the implementation of NL$_2$CM. The main system modules are depicted top-down in Figure 2 according to the formerly described architecture. In this Figure, the modules and data repositories painted black are new.

[2]"should" does not appear in the query, since it is implied by the fact the SATISFYING clause handles individual data.

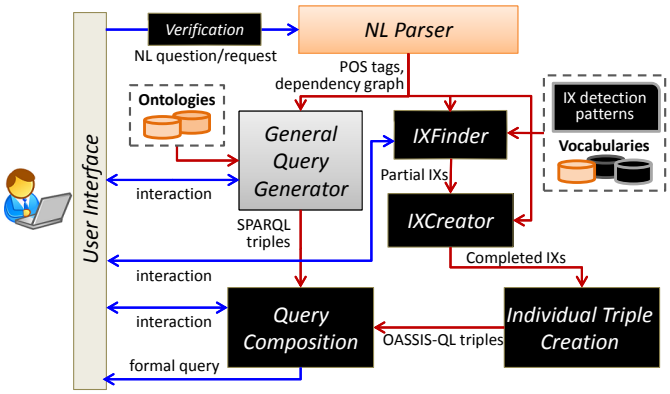

Figure 2: System Architecture

NL$_2$CM is implemented in Java 7, and its graphical user interface is web-based, and written in PHP 5.3 using jQuery 1.x. Users can write their questions/requests as free text in a text field (see Figure 3). The text then undergoes a basic verification process, which checks for certain types of questions/requests that are not supported by the system. In our case, one common example is *descriptive questions* such as "How to...?" "Why...?" "For what purpose...?", whose answer semantics is not supported by OASSIS-QL. In the case that NL$_2$CM detects an unsupported question, it displays an adequate warning to the user along with a link to an explanation and tips how to rephrase the question.

The questions that pass the verification step are sent to the NL parsing modules. In NL$_2$CM, we use the Stanford Parser [8] to obtain both the dependency graph representation and POS tags of the input request, that are passed on the subsequent modules (See Section 2.2).

Our newly developed IX Detector is split in Figure 2 into two components: the *IXFinder* uses vocabularies and a set of predefined patterns in order to find IXs within the dependency graph, as described in Section 2.3. We use a dedicated vocabulary for each type of IX pattern: for *lexical* individuality, we use the Opinion Lexicon [3], which contains expressions of sentiment or subjectivity. For the other types we use vocabularies of our own making (hence, some of the vocabularies in Figure 2 are painted black). The second module, the *IXCreator*, is responsible for completing the subgraphs representing the IXs. For example, if some verb is found to have an individual subject, this component further retrieves other parts belonging to the same semantic unit, e.g., the verb's objects.

The Query Generator is responsible for translating the general query parts into SPARQL triples. For that, NL$_2$CM employs the FREyA system [5] as a black-box module. Since FREyA only accepts full sentences as an input, NL$_2$CM feeds it with the dependency graph and POS tags of the original user request, *including the IXs*. Some of the IXs may be wrongly matched by FREyA to the ontology and translated into general query triples. To overcome this problem, the Query Composition module later deletes generated SPARQL triples that correspond to detected IXs.

The Individual Triple Creation module receives the IXs, and converts them, in this case, into OASSIS-QL triples, as described in Section 2.5. These triples are then composed

[3]http://www.cs.uic.edu/ liub/FBS/sentiment-analysis.html

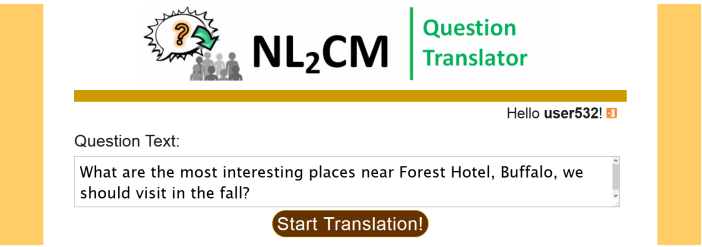

Figure 3: Enter NL question (UI screenshot)

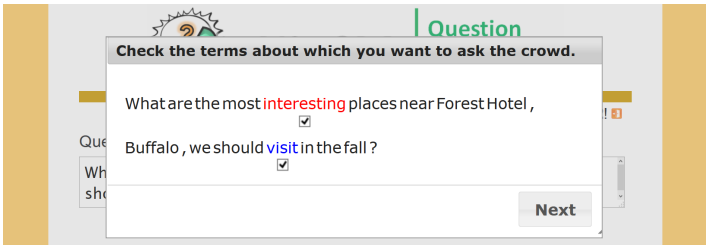

Figure 4: Verify the IXs (UI screenshot)

with the generated SPARQL triples into a well-formed query by the Query Creation module, as described in Section 2.6.

The UI of NL2CM allows manually editing the output query (Figure 6). For convenience, the design of NL2CM allows connecting it directly to OASSIS, the OASSIS-QL evaluation platform. This further enables the user to submit the query via the NL2CM UI to be executed with the crowd, track the progress of the evaluation process through OASSIS's UI, etc.

4. DEMONSTRATION

The usage of NL2CM includes optional interaction with the users. Next, we describe this interaction, and then give the full details about the interactive demonstration scenario.

4.1 User Interaction

Our user studies show that the quality of our developed translation is high for real user questions even without interacting with the user. Yet, user interaction may still be useful (i) for manually adjusting certain query parameters instead of using default values; and (ii) in cases of input questions that are ambiguous or vague. Thus, as depicted in Figure 2, some of the translation framework modules may interact with the user via the UI of NL2CM. The optional points of interaction are exemplified in Figures 3-6, for the running example question. To minimize the user effort, the system may be configured to always skip certain interaction points, or skip them when there is no uncertainty.

Figure 4 depicts the first possible point of interaction with the user in NL2CM, by the IX Detection module. An IX detection pattern can be marked as "uncertain", in which case the IX Detector asks the user to verify IXs detected by this pattern. For that, the uncertain individual parts are highlighted (each by a different color), and the user can check the parts about which the crowd should be asked. (For the sake of the example, in Figure 4 we have marked all the IX detection patterns as uncertain.)

The General Query Generator may also interact with the user to align ambiguous NL terms with the ontology concepts. This kind of interaction is already employed by FREyA, and we have thus plugged it into the UI of NL2CM. For example, FREyA can ask the user whether "Buffalo" refers to Buffalo, NY, USA, to Buffalo, IL, USA, or to other entities in the ontology that are named Buffalo. The response of the user is recorded and serves to improve the ranking of optional entities in subsequent user interactions with the system [5].

Next, the Query Composition module may interact with the user to complete missing details that are required for composing the query. As noted in Section 2.1, user questions rarely contain explicit values which can be used in the LIMIT or THRESHOLD expressions of an OASSIS-QL query.[4] In other cases, the system may either use default values or ask the user to specify the missing values. In Figure 5, the user is asked to specify the value of k for the top-k query over interesting places, which sets the value of the LIMIT expression in the query $\mathcal{Q}$ in Figure 1, line 8. The user is also asked to select the minimal frequency of the "visit in the fall" event, which is translated to a threshold between 0 and 1, and can be plugged into the THRESHOLD expression in line 12 of $\mathcal{Q}$.

Last, by default, the SELECT clause does not project out any variables. In the running example question this makes sense, since the only variable in $\mathcal{Q}$ corresponds to "places", and the relevant places are indeed the answer to the original question. However, consider the variation "What are the most interesting places ... we should visit *with a tour guide*?". The user may or may not want to get, along with each place, the name of the tour guide that recommends it. Moreover, if we replace "a tour guide" with "locals", the user is probably not interested in specific names of locals with whom to visit Buffalo. Thus, the system can ask the users for which terms, out of the terms that correspond to query variables, they want to receive instances (variable bindings). After this last interaction with the user, the final query is composed and displayed (Figure 6).

4.2 Full Demonstration Scenario

To introduce NL2CM to the conference participants, we will demonstrate its translation of NL questions to OASSIS-QL, its interaction with the users, and the execution of the resulting queries by the OASSIS engine. The demonstration will be split into three parts: (i) translating real-life NL questions collected from web forums; (ii) translating NL questions from the audience members; and (iii) observing the feedback of the system about questions that could not be directly translated. We will use three monitors, for the NL2CM interactive UI, for the OASSIS crowd UI, and for the NL2CM administrator mode, which will provide the audience a peek "under the hood" of the system. The system will use the publicly available general data ontologies LinkedGeoData and DBPedia.[5]

For the first step of the demonstration, we have collected a set of example questions that were originally posted on the question-and answer platforms Yahoo! Answers [11]. These questions concern various topics, including travel ("Which hotel in Vegas has the best thrill ride?"), shopping ("What type of digital camera should I buy?"), health ("Is chocolate milk good for kids?"), and others. We will ask the audience

[4]Note, however, that similar values are likely to be required by other query languages.

[5]http://dbpedia.org/About

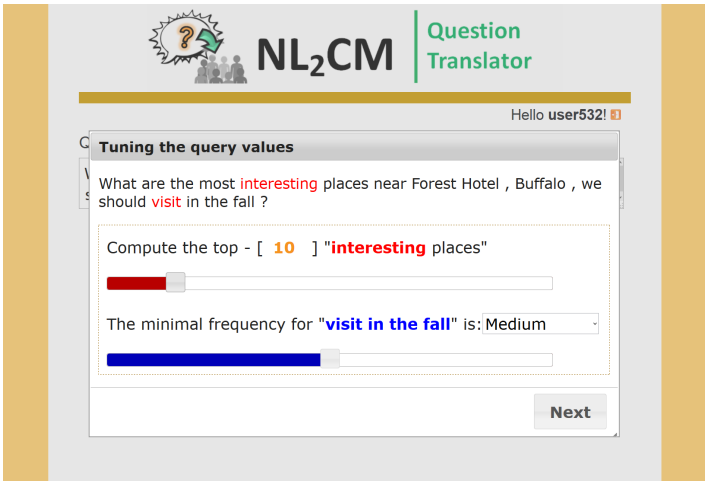

Figure 5: Select the `LIMIT` value (UI screenshot)

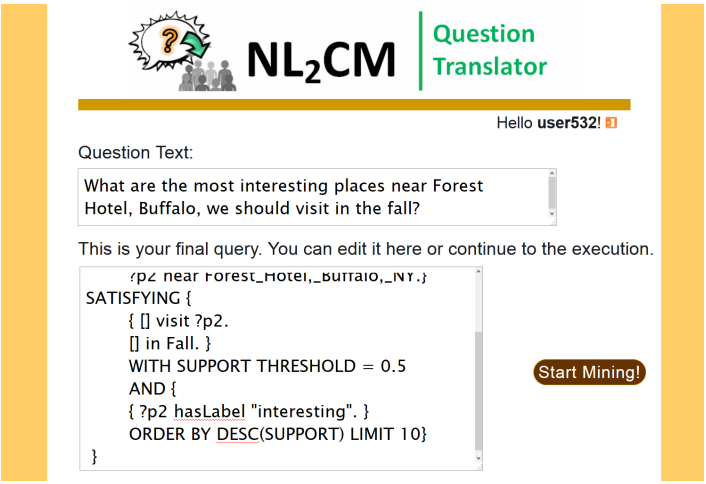

Figure 6: Final query (UI screenshot)

to choose a topic of interest, and then feed a sample question about this topic to NL$_2$CM. Then, we will demonstrate the translation process and inspect the generated query; this will give us a better understanding of the query structure and the mapping performed by system. To gain further intuition about the query semantics, we will instruct the system to feed some of the generated queries into `OASSIS` and, via the second monitor, view the tasks that are generated to the crowd by the query engine.

In the second step, we will invite volunteers to write questions of their own and feed them to the system. If a question does not pass the verification step, the system will point out to the user where the difficulty occurred, as well as provide tips on how to rephrase the question, where possible. Otherwise, our volunteer users will be able to interact with the system, verify the detected IX or provide additional information that is necessary for composing the query (as shown for the running example question in Figures 3-6), and will finally observe the resulting `OASSIS-QL` queries.

To complete the picture, in the case that all the questions in the previous stage passed the verification, we will give a few examples for real-life questions from Yahoo! Answers that do not pass the verification. We will explain what is the reason that certain types of question are not supported, and show the tips NL$_2$CM provides for rephrasing them. For instance, "How should I store coffee?" is a descriptive question which is not supported, but the similar question "*At what container* should I store coffee?" is supported (and for this example, may capture the original user's intention just as well).

While the volunteer users interact with NL$_2$CM, we will also turn the attention of the audience to the administrator mode monitor, which will display the intermediate outputs passed between the NL$_2$CM modules. We will be able to observe the structure and content of these outputs, and gain additional intuition about the operation of the system.

Acknowledgements. This work has been partially funded by the European Research Council under the FP7, ERC grant MoDaS, agreement 291071 and by the Israel Ministry of Science.

5. REFERENCES

[1] A. Amarilli, Y. Amsterdamer, and T. Milo. On the complexity of mining itemsets from the crowd using taxonomies. In *ICDT*, 2014.

[2] Y. Amsterdamer, S. B. Davidson, T. Milo, S. Novgorodov, and A. Somech. OASSIS: query driven crowd mining. In *SIGMOD*, 2014.

[3] A. Bozzon, M. Brambilla, S. Ceri, and A. Mauri. Reactive crowdsourcing. In *WWW*, 2013.

[4] V. Crescenzi, P. Merialdo, and D. Qiu. A framework for learning web wrappers from the crowd. In *WWW*, 2013.

[5] D. Damljanovic, M. Agatonovic, H. Cunningham, and K. Bontcheva. Improving habitability of NL interfaces for querying ontologies with feedback and clarification dialogues. *J. Web Sem.*, 19, 2013.

[6] F. Li and H. V. Jagadish. Constructing an interactive natural language interface for relational databases. *PVLDB*, 8(1), 2014.

[7] Y. Li, H. Yang, and H. V. Jagadish. NaLIX: A generic natural language search environment for XML data. *ACM Trans. DB Syst.*, 32(4), 2007.

[8] M. D. Marneffe, B. Maccartney, and C. D. Manning. Generating typed dependency parses from phrase structure parses. In *LREC*, 2006.

[9] A. D. Sarma, A. Parameswaran, H. Garcia-Molina, and A. Halevy. Crowd-powered find algorithms. In *ICDE*, 2014.

[10] Stanford Sentiment Analysis tool. `http://nlp.stanford.edu/sentiment`.

[11] Yahoo! webscope dataset ydata-yanswers-all-questions-v1_0. `http://research.yahoo.com/Academic_Relations`.

[12] M. Yahya, K. Berberich, S. Elbassuoni, M. Ramanath, V. Tresp, and G. Weikum. Deep answers for naturally asked questions on the web of data. In *WWW*, 2012.

[13] C. Zhang, Z. Zhao, L. Chen, H. V. Jagadish, and C. Cao. CrowdMatcher: crowd-assisted schema matching. In *SIGMOD*, 2014.

[14] L. Zhang, D. V. Kalashnikov, and S. Mehrotra. Context-assisted face clustering framework with human-in-the-loop. *IJMIR*, 3(2), 2014.