Automatic Generation and Reranking of SQL-Derived Answers to NL Questions

Alessandra Giordani and Alessandro Moschitti

Department of Computer Science and Engineering,
University of Trento, Italy

Abstract. In this paper, given a relational database, we automatically translate a natural language question into an SQL query retrieving the correct answer. We exploit the structure of the DB to generate a set of candidate SQL queries, which we rerank with a SVM-ranker based on tree kernels. In particular we use linguistic dependencies in the natural language question and the DB metadata to build a set of plausible SELECT, WHERE and FROM clauses enriched with meaningful joins. Then, we combine all the clauses to get the set of all possible SQL queries, producing candidate queries to answer the question. This approach can be recursively applied to deal with complex questions, requiring nested queries. We sort the candidates in terms of scores of correctness using a weighting scheme applied to the query generation rules. Then, we use a SVM ranker trained with structural kernels to reorder the list of question and query pairs, where both members are represented as syntactic trees. The f-measure of our model on standard benchmarks is in line with the best models (85% on the first question), which use external and expensive hand-crafted resources such as the semantic interpretation. Moreover, we can provide a set of candidate answers with a Recall of the answer of about 92% and 96% on the first 2 and 5 candidates, respectively.

1 Introduction

In the last decade, a variety of approaches have been developed to automatically convert natural language questions into machine-readable instructions. In the area of databases, question answering systems are supposed to answer a natural language question by executing a SQL query. This is obviously a complex task as systems have to deal with the lexical gap between natural language expressions and database structure. In this paper, we will demonstrate that it is possible to fill such gap by relying on (i) the informative metadata embedded in all real databases, (ii) natural language processing methods, e.g., syntactic parsing, and (iii) advanced machine learning to build kernel-based rerankers.

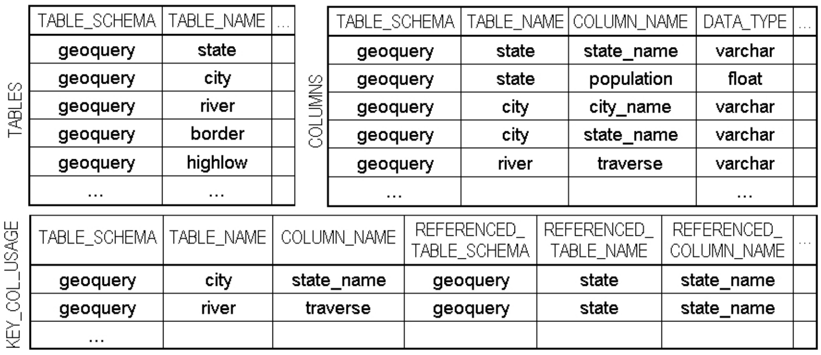
When designing a database, domain experts are requested to organize entities and relationships naming tables and columns in a meaningful way (i.e. *state_name* or *capital* instead of *table_1* or *table_2*). Moreover the database schema also specifies constraints and data types. This metadata is stored in an underlying database that contains tables of each database. The latter, in turn, contain columns referring to table names and column names. Such logic organization is referred to as *catalog*, and in SQL systems it is stored in a database called INFORMATION_SCHEMA (IS for brevity). A fragment sample is shown in Figure 1. IS can be inspected as a normal database, posing SQL queries to obtain useful fields to build a new SQL query.

Instead of using tailored dictionaries, we can enrich our knowledge based on the metadata added by the domain expert, when designing the database. For example, an answer for the question "*Which rivers run through New York*" can be found in the GeoQuery corpus (whose structure is stored in IS as shown in Figure 1).

While we have a simple matching for the word *rivers* with table *river* and *column river_name*, there isn't a direct mapping between the word *run* in the question and any of the columns in the metadata. However, the disambiguation of the term *run* can be easily performed by looking at the less semantically distant metadata entry, i.e., *traverse*. This matching is re-confirmed when investigating on all possible interpretations of *New York* in this database (i.e. city_name, state_name, etc.), by the existing reference between column *traverse* in table river and column *state_name* in table state.

However, a link between both words *New* and *York* is not so easy, since there is no evidence of relatedness between the two words in the metadata: this means that the whole database should be looked up for their stems. Words can be matched with lots of values (e.g., "New York" both as city and as state name, but also with "New Jersey"), as shown by Figure 2. We can generate all possible (even ambiguous) queries exploiting related metadata information (i.e. primary and foreign keys, constraints, datatypes, etc.) and select the most plausible one using a re-ranker.

Last but no least, we deal with complex natural language (NL) questions, containing subordinates, conjunctions and negations and nested SQL queries. In particular, we designed a mapping algorithm that matches dependencies between NL components and SQL structure that allows to build a set of possible queries that answers a given question.



TABLES

TABLE_SCHEMA	TABLE_NAME	...
geoquery	state	
geoquery	city	
geoquery	river	
geoquery	border	
geoquery	highlow	
...	...	

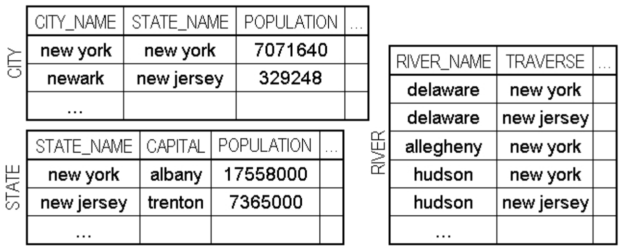
COLUMNS

TABLE_SCHEMA	TABLE_NAME	COLUMN_NAME	DATA_TYPE	...
geoquery	state	state_name	varchar	
geoquery	state	population	float	
geoquery	city	city_name	varchar	
geoquery	city	state_name	varchar	
geoquery	river	traverse	varchar	
...			...	

KEY_COL_USAGE

TABLE_SCHEMA	TABLE_NAME	COLUMN_NAME	REFERENCED_ TABLE_SCHEMA	REFERENCED_ TABLE_NAME	REFERENCED_ COLUMN_NAME	...
geoquery	city	state_name	geoquery	state	state_name	
geoquery	river	traverse	geoquery	state	state_name	
...						

Fig. 1. A DBMS catalog containing GEOQUERY and SAKILA



CITY

CITY_NAME	STATE_NAME	POPULATION	...
new york	new york	7071640	
newark	new jersey	329248	
...			

STATE

STATE_NAME	CAPITAL	POPULATION	...
new york	albany	17558000	
new jersey	trenton	7365000	
...			

RIVER

RIVER_NAME	TRAVERSE	...
delaware	new york	
delaware	new jersey	
allegheny	new york	
hudson	new york	
hudson	new jersey	
...		

Fig. 2. GeoQuery database fragment

Section 2 gives a formal description of the problem while Section 3 describes the basic steps of our algorithm used to build clause. Section 4 shows how we prune and weigh queries in their possible combinations to generate an ordered set of meaningful queries among which we find the answer. Section 5 describes tree kernels our kernel-based rerankers. Section 6 discusses the results obtained using a reranking algorithm, while Section 7 draws some conclusions.

2 The Problem

We will begin by introducing the notion of typed dependencies and how to obtain a collapsed list of dependencies starting from an NL sentence. Then we will introduce the subset of Structured Query Language that our system can deal with and, in order to formalize the problem, we will recall the notation of corresponding operations in relational algebra.

2.1 NL Questions and Dependencies List

To represent the textual relationships of the NL sentence we use typed dependency relations. The Stanford Dependencies representation [8] provides a simple and consistent description of the binary grammar relations existing between a governor and a dependent. As shown in the example below, each dependency is written as *abbreviated_relation_name* (governor, dependent). The governor and the dependent are words in the sentence associated with a number indicating the position of the word in the sentence.

In particular we refer to collapsed representation, where dependencies involving prepositions, conjuncts, as well as information about the referent of relative clauses are collapsed to get direct dependencies between content words.

For example, the Stanford Dependencies Collapsed (SDC) representation for the question, q_1: "*What are the capitals of the states that border the most populated state?*" is the following:

$SDC_{q_1} = attr$(are-2, what-1), $root$(ROOT-0, are-2),
det(capitals-4, the-3), $nsubj$(are-2, capitals-4),
$nsubj$(border-9, states-7), $rcmod$(states-7, border-9),
det(states-13, the-10), $advmod$(populated-12, most-11),
$amod$(state-13, populated-12), $dobj$(borders-9, state-13)

The current representation contains approximately 53 grammatical relations but for our purposes we only use the following: adverbial and adjectival modifier, agent, complement, object, subject, relative clause modifier, prepositional modifier, and root.

2.2 SQL Queries and Relational Algebra

The general SQL query with which our system can deal has the following form:

$$\text{SELECT } COLUMN \text{ FROM } TABLE \text{ [WHERE } CONDITION\text{]} \tag{1}$$

The query is interpreted starting from the relation in the FROM clause, selecting tuples that satisfy the condition indicated in the WHERE clause (optional) and then projecting the attribute in the SELECT clause.

In relational algebra, selection and projection are performed by σ and π operators respectively. The meaning of the SQL query above is the same as that of the relational expression:

$$\pi_{COLUMN}\,(\sigma_{CONDITION}(TABLE)) \tag{2}$$

It is worth noting that while relational algebra formally applies to sets of tuples (i.e. relations), in a DBMS relations are bags so it may contain duplicate tuples [3]. For our purposes the fact of having duplicates in the result adds nois; this is why we always delete multiple copies of a tuple by using the keyword DISTINCT in the *COLUMN* field. In our QA task we expect that questions can be answered with a single result set (e.g. we can deal with "*Cities in Texas*" and "*Populations in Texas*" but not with the combined query "*Cities and their population in Texas*"). That is, even if in general *COLUMN* could be a - possibly empty - list of attributes, in our system it just contains one attribute. We can apply to this attribute aggregation operators that summarize it by means of SUM, AVG, MIN, MAX and COUNT, always combined with DISTINCT keyword (e.g. `SELECT COUNT(DISTINCT state.state_name)`).

Instead, *CONDITION* is a logical expression where basic conditions, in the form e_L OP e_R, with OP=$\{<, >$,LIKE,IN$\}$, are combined with AND, OR, NOT operators. While e_L is always in the form `table.column`, e_R could be:

– numerical value (e.g. `city.population > 15000`) or
– string value (e.g. `city.state_name LIKE "Texas"`) or
– nested query (e.g. `city.city_name IN (SELECT state. capital FROM state)`

An example of a complex WHERE condition could be the following:
`city.population > 15000 AND city.city_name NOT IN (SELECT state.capital FROM state)) AND NOT city.state_name LIKE "Texas"` (i.e. "*major non-capital cities excluding texas*").

The meaning of *TABLE* is more straightforward, since it should contain table name(s) to which the other two clauses refer. This clause could just be a single relation or a join operation, which selectively pairs tuples of two relations. We only deal with theta-joins where we take the Cartesian product of two relations and exclusively select those tuples that satisfy a condition C. The notation for theta-joins of relations R and S based on condition C is $\overset{R \bowtie S}{C}$. We use the SQL keyword ON to keep this condition C separated from the other WHERE conditions since it reflects a database requirement and shouldn't match to anything of the NL question. (e.g. `city JOIN state ON city.city_name = state.capital`).

The complexity of generated queries is fairly high indeed, since we can deal with questions that require nesting, aggregation and negation in addition to basic projection, selection and joining (e.g. "*How many states have major non-capital cities excluding Texas*").

2.3 Problem Definition

The question answering task of finding an SQL query that retrieves an answer for a given NL question reduces to the following problem.

Given a question q represented by means of one typed dependency collapsed list SDC_q, generate the three sets of clauses $\mathcal{S}, \mathcal{F}, \mathcal{W}$ (argument of SELECT, FROM and WHERE, respectively) such that:

$$\exists s \in \mathcal{S}, \exists f \in \mathcal{F}, \exists w \in \mathcal{W} \text{ s.t. } \pi_s\left(\sigma_w(f)\right) \text{ answers q} \tag{3}$$

The query *answer* $\pi_s\left(\sigma_w(f)\right)$ is chosen among the set of all possible queries $\mathcal{A} = \{$SELECT $s\times$ FROM $f\times$ WHERE $w\}$ in a way that maximizes the probability of generating a result set answering question q.

3 Building Clauses Sets

In order to generate all possible queries for a question q we need to find their possible SELECT, FROM and WHERE clauses ($\mathcal{S}, \mathcal{F}$ and $\mathcal{W}$). We start from a dependency list SDC_q and (a) prune and stem its components, (b) add synonyms, (c) create the set of stems used to build $\mathcal{S}$ and $\mathcal{W}$ and (d) keep only dependencies possibly used in the recursive step to generate nested queries. Building the set $\mathcal{F}$ from $\mathcal{S}$ and $\mathcal{W}$ is straightforward.

We are now going to briefly discuss some examples to introduce the objective of individual steps and clarify how the entire process is carried out. The first question we take into account is the simplest one: "*What is the capital of Texas?*". Its answer can be retrieved executing the query: `SELECT capital FROM state WHERE state.state_name='Texas'`. We can see that they share only two stems, *capital* and *Texas*. The key of categorizing stems (Section 3.2) is to recognize that the first stem will be used in $\mathcal{S}$ and the second one in $\mathcal{W}$. In particular, since the word *Texas* is not a value in the *IS*, it is used as a r-value in the WHERE expression, while the l-value is derived from the column name under where it appears (Section 3.4).

The fact of being respectively projection and selection oriented can be inferred looking at their grammar relations, i.e. inspecting the dependency list (e.g. root of the sentence together subject dependent are typically used for projections). This list needs to be preprocessed (section 3.1) to take into account only relevant relations between the *stems* of the question. Let us consider for example the question: "*What is the capital of the most populous state?*" and its associated answering query `SELECT capital FROM state WHERE population = (SELECT max(population) FROM state)`.

The matching words are *capital* and *state*, while stemming also allows to find a mapping through *popul*. We can note that this stem is used both in the l-value and in the r-value of the WHERE expression. In fact, this query requires nesting and indeed the categorizing algorithm needs to be recursive. This stem is classified both as a selection oriented stem for the outer query, and as a projection oriented one for the inner query (note that it requires aggregation, handled when generating the SELECT clause set).

Finally we will introduce one last example to clarify Section 3.5. While with the other examples it is straightforward to compile the FROM clause, since the other clauses refer to the same table, when we deal with columns belonging to different tables things get complicated. Take question "*What are the capitals states bordering Texas?*") and its associated query `SELECT capital FROM ... WHERE border = 'Texas'`. How can we fill in the dots in the FROM clause? Fields *capital* and *border* belong respectively to tables *state* and *border_info*. Form the database catalog, we learn that these two tables are connected via the foreign key *state_name* and so the final $\mathcal{F}$ will include the following join: `state JOIN border_info on state.state_name = border_info.state_name`.

3.1 Optimizing the Dependency List

As introduced in Section 2.1, we don't need all grammatical relations provided in output by the Stanford Dependency parser. For this reason before preprocessing the list of dependencies we need to prune the useless ones and remove from *gov*ernors and *dep*endents the appended number (indicating the position of the word in question q). Then, *gov*s and *dep*s are reduced to stems (using the Porter stemmer[1]).

In order to disambiguate the sense of the stems that do not appear in metadata but could match with it, we create a list of synonyms using off-the-shelf resources (like Wordnet and similarity measures) combined with our internal knowledge (represented by database constraints). Using this list we can substitute certain stems with their stemmed synonyms.

The resulting SDC_q is optimized to be processed by the next step. An example showing $SDC_{q_1}^{opt}$ with respect to the original SDC_{q_1} introduced in Section 2.1 can be found in Table 1.

3.2 Categorizing Stems

Before building $\mathcal{S}$ and $\mathcal{W}$ sets we need to identify those stems that are projection and/or selection oriented. Those stems will be added respectively to Π and/or

[1] `http://tartarus.org/martin/PorterStemmer/`

Table 1. Categorizing stems into projection and/or selection oriented sets

(1)*root*(ROOT, are),	$\Pi = \{\text{capital, state}\}$
(2)*nsubj*(are, capital),	$\Sigma = \{\text{are}\} \Rightarrow \Sigma = \phi$
(3)*prep_of*(capital, state),	
(4)*nsubj*(border, state),	$\Pi' = \{\text{state, border}\}$
(5)*rcmod*(state, border),	$\Sigma' = \{\text{border, state}\}$
(6)*advmod*(populat, most),	
(7)*amod*(state, populat),	$\Pi'' = \{\text{most, populat, state}\}$
(8)*dobj*(border, state)	$\Sigma'' = \phi$

Σ categories according to the following rules. For each grammatical relation *rel*(*gov*,*dep*) in SDC_q^{opt}:

1. If it is *ROOT*, *dep* is the key to populate $\mathcal{W}$ so add it to Σ and remove the relation from SDC_q^{opt}. This stem can be an auxiliary verb, e.g., *is, are, has, have* and so on. It is useless to build the arguments of the queries but it could be used transitively to add other stems[2].
2. If it starts with *nsubj*, check if $gov \in \Sigma$. If not (because there isn't any *ROOT* relation) add *gov* to Σ. Then add *dep* to Π and remove *rel* from SDC_q^{opt}, otherwise keep it, since it could be a subject related to a subordinate (we will need it in the recursive steps).
3. If it starts with *prep* or it ends with *obj*, we used it to create conditions (possibly involving nesting):
   - check if $gov \in \Pi$. If not (because no *ROOT* or *nsubj* relations were found so far) add *gov* to Π.
   - Then add *dep* to Σ if there is not any *table.column* like [3] *gov.dep*. Otherwise, also add *dep* to Π and remove *rel* from SDC_q^{opt}.
4. If it ends with *mod*, it implies that *dep* is a modificator of *gov*, so they should be paired together: if $gov \in \Sigma$ add *dep* to Σ and if $gov \in \Pi$ add *dep* to Π and remove *rel* from SDC_q^{opt}. This should be done only if *dep* is not a superlative (i.e. doesn't end with -st). The non-removed relations will be taken into account in the recursive step, adding both *dep* and *gov* to Π.
5. If none of the above rules can be applied, iterate the algorthm recursively building Π' and Σ', Π'' and Σ'' and so on, until SDC_q^{opt} is empty.

In order to show how these steps are used to build projection and/or selection oriented sets from which we generate $\mathcal{S}$ and $\mathcal{W}$, let us consider the list of optimized dependencies $SDC_{q_1}^{opt}$ in Table 1.

[2] Stems of 3 or less characters would introduce too much noise in retrieving matching strings, so they will be eliminated in an additional step 6. Useful words like *in, of, not, or, and* are embedded in relation abbreviations when collapsing dependencies.

[3] We query metadata seeking for something similar to *gov* as a table and to *dep* as a column, i.e. we search for table names using π_{table_name} $(\sigma_{table\_name \cong dep \wedge column\_name \cong gov}(IS.Columns))$. For brevity we use the symbol $s_1 \cong s_2$ for s_2 substring of s_1, i.e. s_1 LIKE "%s_2%".

At the first iteration we use $ROOT$ to add *are* to Σ, then we also exploit it to add *capital* and include *state* to Π as soon as we check that there is an occurrence `state.capital` in IS. At this point these three relations have been deleted from $SDC^{opt}_{q_1}$ obtaining $SDC^{opt\prime}_{q_1}$ used in the next iteration. Note that since *are* is a short stem, it should be deleted from Σ.

At the second iteration (first recursion step) we don't have a $ROOT$ relation so we use *nsubj* to add add *border* to Σ' and *state* to Π'. Since with *rcmod* we find an occurrence `border.state_name` in IS, *border* is added also to Π. At this point, seeking through the end of the list we discard *dobj* because even if *border* $\in \Pi'$ we do not find `state.border` in IS, so these other three relations are deleted from $SDC^{opt\prime}_{q_1}$ obtaining $SDC^{opt\prime\prime}_{q_1}$ for the last iteration.

In the third iteration we have $SDC^{opt\prime\prime}_{q_1}$ composed by two *mod* relations, so we add all stems to Π'' and delete their associated relations from the list.

3.3 Building the SELECT Clauses Set

Once we have identified the set Π of projection-oriented stems, we can use it to search in metadata all the fields that could match with them. The generation process for S is described by the following generative grammar.

$$\begin{aligned}
&\mathcal{S} \rightarrow \text{AGGR '(' FIELD ')'} \mid \text{FIELD}\\
&\text{AGGR} \rightarrow \text{max} \mid \text{min} \mid \text{sum} \mid \text{count} \mid \text{avg}\\
&\text{FIELD} \rightarrow \text{TAB.COL}\\
&\text{TAB} \in \bigcup^{x\in\Pi} \pi_{table\_name}(\sigma_{table\_name \cong x}(\text{IS.Tables}))\\
&\text{COL} \in \bigcup^{x\in\Pi} \pi_{column\_name}(\sigma_{column\_name \cong x}(\text{IS.Columns}))
\end{aligned}$$

With each element of $\mathcal{S}$, we also associate a weight w_i, calculated according to the procedure described in Section 4.3 (we will discuss it later). For example, considering the IS scheme in Figure 1, the SELECT clauses originated from Π of Table 1 are shown in Fig. 3. Note that the superscript numbers indicate the weight associated with each statement.

$$\begin{aligned}
\mathcal{S} &= \{state.capital^3, state.state\_name^2, border\_info.state\_name^1, ...\}\\
\mathcal{S}' &= \{border\_info.state\_name^3, border\_info.border^2, state.state\_name^2, ...\}\\
\mathcal{S}'' &= \{max(state.population)^4, max(city.population)^3, state.population^3, ...\}
\end{aligned}$$

Fig. 3. A subset of SELECT clauses for q_1

3.4 Building the WHERE Clauses Set

Before generating WHERE clauses, the selection-oriented set of stems Σ should be divided into two distinct sets: Σ_L and Σ_R.

The set Σ_L contains stems that find their matching in IS and allow us to build the set of left-hand side expressions $\mathcal{W}_L \rightarrow \text{FIELD}^{w_i}$, where FIELD is defined above and computed with Σ_L in place of Π (w_i is its associated weight).

For the remaining stems $\Sigma_R = \Sigma - \Sigma_L$ we search for a match in the database: $\forall col \in$ IS.$Columns$, $\forall tab \in$ IS.$Tables$, generate
$\mathcal{W}_R = \left\{x|\pi_{count(*)}\left(\sigma_{col \cong x}(Geoquery.tab)\right) >= 0\right\}$.

Then, in order to build the WHERE clause set, $\mathcal{W}$, $\forall e_L \in \mathcal{W}_L, \forall e_R \in \mathcal{W}_R$ we first generate basic expressions $expr = e_L$ OP $_R$ and combine them by means of conjunctions and negations (see Section 2.2), keeping only those expressions $expr$ such that the execution of $\pi_{count(*)}\left(\sigma_{expr}(table)\right)$ does not lead to an error for at least a *table* in the database.

To understand how it works, let us introduce a new example question q_2: "*what are the capitals of states bordering New York?*". The $SDC^{opt}_{q_2}$ is similar to $SDC^{opt}_{q_1}$ except for the last three relations. Row (6) disappears while rows (7) and (8) are replaced by *amod*(york, new) and *dobj*(border, york), leading to $\Sigma' = \{$border, new, york$\}$. This set is split into $\Sigma'_L = \{$border$\}$ and $\Sigma'_R = \{$new, york$\}$.
We build $\mathcal{W}'_L = \left\{border\_info.border^3, border\_info.state\_name^2\right\}$ and
$\mathcal{W}'_R = \left\{'new\ york'^2, 'new\ mexico'^1, 'new\ jersey'^1, 'newark'^1\right\}$. Finally we generate the set of possible valid conditions and their weights:
$\mathcal{W} = \{border\_info.border = \text{‘}new\ york'^5, border\_info.state\_name = \text{‘}new\ york'^4, ...\}$.

Anyway, the set Σ_R could happen to be empty. For example, when the WHERE condition requires nesting: in this case e_R will be the whole subquery (e.g. Σ' in Table 1). It could be the case that also Σ_L is empty. In fact a query without a WHERE clause is valid (e.g. Σ'' in Table 1). In any case, even if there are no selection-based stems, $\mathcal{W}$ may not be empty (e.g. Σ in Table 1). Taking into account all tables and columns we can get more conditions: $\mathcal{W}*_R = \{tab.col$ such that $tab \in \pi_{table\_name}\left(IS.Columns\right)$ and $col \in \pi_{column\_name}\left(IS.Columns\right)\}$.

3.5 Building the FROM Clauses Set

The generation of the FROM clause $\mathcal{F}$ is straightforward given $\mathcal{S}$ and $\mathcal{W}$. This set will contain all tables to which clauses in $\mathcal{S}$ and $\mathcal{W}$ refer, enriched by pairwise joins.

As stated before, this information can be found running SQL queries over IS exploiting metadata stored in table KEY_COLUMN_USAGE (in short, K; see Figure 2). This table identifies all columns in the current databases that are restricted by some unique, primary key, or foreign key constraint. That is, for each usage of foreign key column in the table, we can determine how many aggregate table columns match that column usage. First, we extract tables appearing in $\mathcal{S}$ and $\mathcal{W}$ (i.e. words ending with dot), creating a set F. At the beginning $\mathcal{F}$=F. Then $\forall t_1, t_2 \in F\, \pi_{col\_name, ref\_col\_name}\left(\sigma_{table\_name=t_1 \wedge ref\_table\_name=t_2}(\text{IS}.K)\right)$ retrieves c_1, c_2 to perform the: join $\overset{t_1 \bowtie t_2}{c_1 = c_2}$. In this way $\mathcal{F}$ in enriched whit the two-table join t_1 `join` t_2 `on` $t_1.c_1 = t_2.c_2$. In addition we can allow for more distant joins by finding an intermediate table useful to link two tables that are not directly referencing each other. This can be done performing a complex join between two instances of KEYS with multiple conditions, but due to for lack of space this can not be illustrated here.

With respect to our example with question q_1 and its SELECT clauses shown in Figure 3, the set of FROM clauses is:
$\mathcal{F}' = \{state, border, state\ join\ border\ on state.state\_name = border\_info.border, ...\}$.

Note that there are no weights associated with FROM clauses because it is not possible to backtrack how many stems made each table appear in $\mathcal{F}$.

4 Generating Queries

In the previous section we saw how to create building blocks for queries starting from a question q. These elements should be paired together in a smart way to generate the set of queries that possibly answer q. This pairing is obtained by creating the Cartesian product between clauses sets from which non-valid, redundant and meaningless clauses are deleted. We use a weighting scheme to order the most probable correct candidate queries.

4.1 Clause Cartesian Product

In order to find possible answering queries we generate the set $\mathcal{A} = \{\mathcal{S}\times\mathcal{F}\times\mathcal{W}\} \cup \{\mathcal{S}\times\mathcal{F}\}$. Given that at least one such query exists there should be one pairing $\langle s, f, w\rangle \in \mathcal{A}$, such that the execution of `SELECT` s `FROM` f `[WHERE` w`]` retrieve the correct answer. Given that each clause set contains on average up to ten items, this product can result in a very huge set. Thus, when generating all pairings some preliminary conditions are verified, e.g. tables appearing in SELECT and WHERE clauses should appear in the FROM clause as well, otherwise the execution of that query will fail. This avoids generating incorrect queries and wasting time trying to execute them.

To give a simple example, we illustrate in Figure 4 some generated clauses for the question q_2 , together with possible pairings. The pairing $\langle s_1, f_1, w_1\rangle$ is not correct: it leads to the MySQL error `Unknown table: border_info`.

4.2 Pruning Useless Queries

Once the set $\mathcal{A}$ of all valid pairings is built, we additionally prune some of them which are not useful. For example, meaningless queries project the same field compared to a value in the selection (e.g. the pairing $\langle s_3, f_2, w_2\rangle$ answers the question "*Which state is New York?*" and is clearly useless).

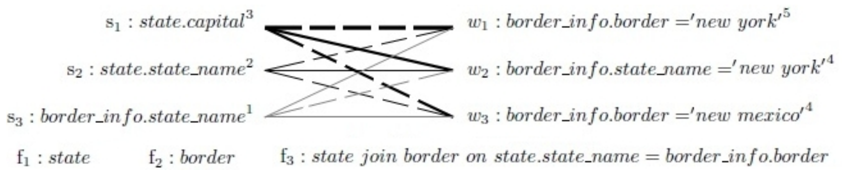

s_1 : $state.capital^3$
w_1 : $border\_info.border = 'new\ york'^5$
s_2 : $state.state\_name^2$
w_2 : $border\_info.state\_name = 'new\ york'^4$
s_3 : $border\_info.state\_name^1$
w_3 : $border\_info.border = 'new\ mexico'^4$
f_1 : $state$
f_2 : $border$
f_3 : $state\ join\ border\ on\ state.state\_name = border\_info.border$

Fig. 4. Possible pairing between clauses for q_2

Moreover there could be redundant queries that, if optimized, allow us to remove duplicates in the set, reducing its cardinality. For example, the pairing $\langle s_2, f_3, w_1 \rangle$ requires the columns *state.state_name* and *border_info.border* to be the same, so w_2 would select the same rows of w_2'(i.e. *state.state_name='new york'*), but this means that table *border_info* is no longer used and this pairing is equivalent to $\langle s_2, f_1, w_2' \rangle$ which, as said above, is meaningless.

4.3 Weighting Scheme

As introduced in the previous sections, we weigh each clause in $\mathcal{S}$ and $\mathcal{W}$ by counting how many stems in the original question originated that clause.

In particular, for the SELECT clause, if there is a table that matches with a stem, its weight is +2 while the matching with columns weighs +1 (common stems between table and column are not valid). Superlatives matching with aggregation operators count as +1.

For the WHERE clause, a weight is computed in the same way as for the left-hand side of the conditions and a +1 is added for each matching value in the right-hand side. In addition when dealing with nested queries, the WHERE clause inherits also the weight of the nested query.

The FROM clauses are not associated with weights. However, we will take into account how many joins are involved when ordering queries with the same weight.

When pairing clauses the total weight is obtained just summing up the weight of its components, and it is used to order the final set $\bar{\mathcal{A}}$ of possible useful queries from the most to the least probable.

Figure 4 highlights this *probabilistic* score (obtained by the heuristic one by normalization) through the thickness of connection lines. Dashed lines illustrate pruned queries. The final ordered set answering q_2 is the following one:
$\bar{\mathcal{A}}=\{\langle s_1, f_3, w_2\rangle^7, \langle s_3, f_2, w_1\rangle^6, \langle s_2, f_3, w_2\rangle^6, \langle s_1, f_1\rangle^3, \langle s_2, f_1\rangle^2,$
$\langle s_3, f_2\rangle^1\}$.
From the pairing with highest weight we derive the answering query, that is: `SELECT state.capital FROM state join border on state.state_name =border_info.border WHERE border_info.state_name='new york'`.

It is worth noting that more then a query can have the same weight. To deal with that, we implemented a comparator that privileges queries involving less joins and embed the most referenced table (e.g. `state` in the case of GeoQuery). See, for example, the order of the second and third pairings in $\bar{\mathcal{A}}$: they have been swapped since f_3 contains a join while f_2 doesn't.

5 Kernel Methods for Ranking Question/Query Mapping

Once an initial rank of the candidate SQL queries has been derived, we can rely on machine learning methods to improve the probability of finding the correct answer in the top position. The need of designing suitable representations of the question and query pairs makes this operation quite complex. For this purpose, we rely on kernel methods.

5.1 Kernel Methods

In kernel-based machines, both learning and classification algorithms only depend on the inner product between instances. In several cases this can be efficiently and implicitly computed by kernel functions by exploiting the following dual formulation: $\sum_{i=1..l} y_i \alpha_i \phi(o_i)\phi(o) + b = 0$, where o_i and o are two objects, ϕ is a mapping from the objects to feature vectors $\boldsymbol{x_i}$ and $\phi(o_i)\phi(o) = K(o_i, o)$ is a kernel function implicitly defining such mapping. In case of structural kernels, K determines the shape of the substructures that describe the objects above.

In the following section, we are going to first propose a structural representation of the question and query pairs, then we will illustrate the Syntactic Tree Kernel (STK) [2], which computes the number of syntactic tree fragments. In the last subsection we will show how to engineer new kernels from them, while the reranking kernel is presented in Sec. 5.5

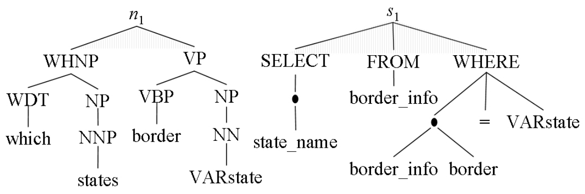


Fig. 5. Question/Query Syntactic trees

5.2 Representing Question and Queries Pairs

In Data Mining and Information Retrieval the so-called bag-of-words (BOW) has been shown to be effective to represent textual documents, e.g. [12,6]. However, in case of questions and queries, we deal with small textual objects in which the semantic content is expressed by means of few words and poorly reliable probability distributions. In these conditions the use of syntactic representation improves BOW and should be always used.

Therefore, in addition to BOW, we represent questions and queries using their syntactic trees, as shown in Figure 5: for questions (a) we used the Charniak's syntactic parser [1] while for queries (b) we implemented an ad-hoc SQL parser. The latter builds a SQL parse tree for each query following its syntactic derivation according to MySQL grammar. The grammar has been slightly modified to accommodate the usage of the symbol • for the production of *items* in the SELECT clause and in WHERE conditions. In such an SQL tree, the internal nodes are only the SQL keywords of the query plus the special symbol • whereas the leaves are names of tables and columns of the database, category variables or operators. Note that, although we eliminated comma and dot from the grammar, it is still possible to obtain the original SQL query, by just performing a preorder traversal of the tree. The above structures can be represented in a learning algorithm using the kernel described in the next section.

5.3 Syntactic Tree Kernels (STK)

Convolution tree kernels [2] compute the similarity between two trees T_1 and T_2 by counting the common sub-trees, without enumerating the whole fragment space. In more detail, let N_1 and N_2 be the set of nodes in T_1 and T_2, respectively. Moreover, let $I_i(n)$ be an indicator variable that is 1 if subtree i is rooted at n and 0 otherwise. Then the convolution kernel K over $T1$ and $T2$ is computed as:

$$STK(T1,T2) = \sum_{n_1 \in N_1, n_2 \in N_2} \Delta(n_1, n_2) \tag{4}$$

where

$$\Delta(n_1, n_2) = \sum_{n_1 \in N_1} \sum_{n_2 \in N_2} \sum_i I_i(n_1) I_i(n_2)$$

is computed efficiently using the following recursive definition:

- If the production rules[4] at n_1 and n_2 are different, then $\Delta(n_1, n_2) = 0$.
- If the production rules at n_1 and n_2 are the same and n_1 and n_2 are pre-terminals, then $\Delta(n_1, n_2) = \lambda$.
- If the production rules at n_1 and n_2 are the same and n_1 and n_2 are not pre-terminals, then:

$$\Delta(n_1, n_2) = \lambda \prod_{j=1}^{nc(n1)} (1 + \Delta(ch(n_1, j)), ch(n_2, j))$$

  where $nc(n_1)$ is the number of children of n_1 in the tree and the j-th children of node n_i is denoted by $ch(n_i, j)$ (note that $nc(n_1) = nc(n_2)$ since the production rule is the same). λ $(0 < \lambda < 1)$ is a decay factor to make the kernel less variable with respect to tree-fragment sizes.

5.4 Kernel Combination for Pairs

We need to represent the members of a pair and their interdependencies. For this purpose, given two kernel functions, $k_1(.,.)$ and $k_2(.,.)$, and two pairs, $p_1 = \langle n_1, s_1 \rangle$ and $p_2 = \langle n_2, s_2 \rangle$, a first approximation is given by summing the kernels applied to the components: $K(p_1, p_2) = k_1(n_1, n_2) + k_2(s_1, s_2)$. This kernel will produce the union of the feature spaces of questions and queries. A more effective kernel is the product $k(n_1, n_2) \times k(s_1, s_2)$, since it generates pairs of fragments, which are member of the Cartesian product of kernel spaces of the questions and queries. As additional feature and kernel engineering, we also exploit the ability of the polynomial kernel to add feature conjunctions. By simply applying the function $(1 + K(p_1, p_2))^d$, we can generate conjunction up to d features. Thus, we can obtain tree fragment conjunctions and conjunctions of pairs of tree fragments.

The next section will show how to use such kernels for an SVM-based reranker.

[4] In a syntactic tree a node with its children correspond to a production rule of the grammar that generated it.

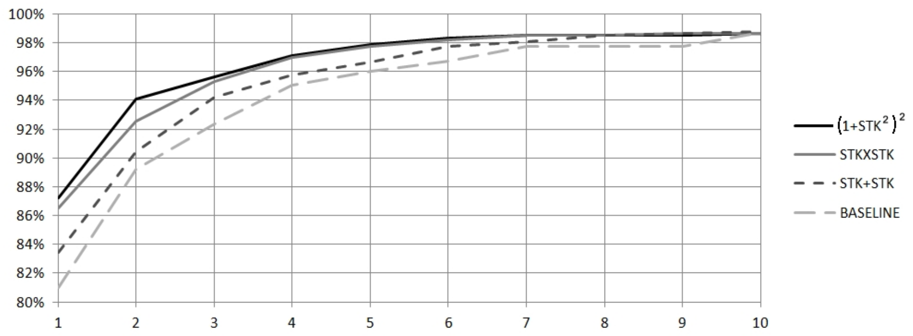


Fig. 6. Recall of the correct answer within different k positions of the system rank

5.5 Preference Reranker

Our reranking model consists in learning to select the best candidate from a given candidate set. In order to use SVMs for training a reranker, we applied the Preference Kernel Method [13]. In the Preference Kernel approach, the reranking problem – learning to pick the correct candidate h_1 from a candidate set $\{h_1, \ldots, h_k\}$ – is reduced to a binary classification problem by creating *pairs*: positive training instances $\langle h_1, h_2\rangle, \ldots, \langle h_1, h_k\rangle$ and negative instances $\langle h_2, h_1\rangle, \ldots, \langle h_k, h_1\rangle$. This training set can then be used to train a binary classifier. At classification time, pairs are not formed (since the correct candidate is not known), while, the standard one-versus-all binarization method is still applied.

The kernels are then engineered to implicitly represent the *differences* between the objects in the pairs. If we have a valid kernel K over the candidate space $\mathcal{T}$, we can construct a preference kernel P_K over the space of pairs $\mathcal{T} \times \mathcal{T}$ as follows: $P_K(x, y) =$

$$\begin{aligned} P_K(\langle x_1, x_2\rangle, \langle y_1, y_2\rangle) = K(x_1, y_1) + \\ K(x_2, y_2) - K(x_1, y_2) - K(x_2, y_1), \end{aligned} \tag{5}$$

where $x, y \in \mathcal{T} \times \mathcal{T}$. It is easy to show that P_K is also a valid Mercer's kernel. This makes it possible to use kernel methods to train the reranker. The several kernels defined in the previous section can be used in place of K[5] in Eq. 5.

6 The Experiments

We ran several experiments to evaluate the accuracy of our approach for automatic generation and selection of correct SQL queries from NL questions. We experimented with a well-known dataset GeoQuery developed in order to study semantic parsing.

[5] More precisely, we also multiply K for the inverse of rank position.

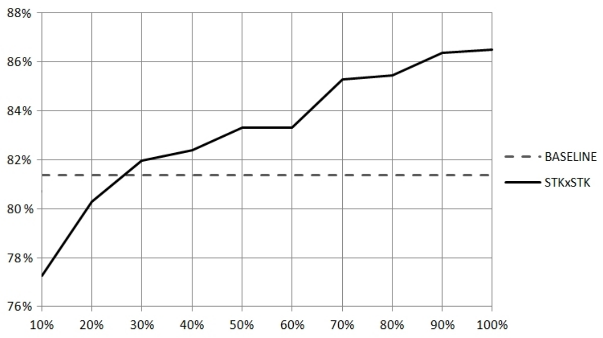


Fig. 7. Learning curve comparison between simple answer generator and the reranking model using the STK × STK kernel

6.1 Setup

To learn the reranker, we used SVM-Light-TK[6], which extends the SVM-Light optimizer [6] with tree kernels. i.e. Syntactic Tree Kernel (STK) as described in Section 5. We modeled many different combinations described in the next section. We used the default parameters, i.e. the cost and trade-off parameters = 1 (for normalized kernels) and $\lambda = 0.4$ (see Sec. 4).

To generate the set of possible SQL queries we applied our algorithm described in Section 3 to GeoQueries[7] corpus. We started from a set of 700 NL questions[8]. Thanks to our generative algorithm we discovered and fixed all errors and inconsistencies in SQL queries, except for 3 cases that still lead to a MySQL error. Indeed, since we can't test the correctness of our generated query (without a result set to compare with) we considered a subset of 697 pairs.

6.2 Generative Results

Given a question from GeoQuery, our algorithm was able to generate a correct SQL query in the first 25 in 95.3% of the cases. This also means that our system cannot answer to 33 questions. This is due to (1) empty clauses set $\mathcal{S}$ and/or $\mathcal{W}$, for example, "*How many square kilometers in the us?*" does not contain any useful stem; and (2) from mismatching nested queries, for example, "*Count the states which have elevations lower than what alabama has*" contains an implicit reference to a missing piece of question. In addition there are ambiguous questions like "*Which states does the colorado?*" from which we retrieve an incomplete dependency set.

For all remaining questions from which we succeed in generating an ordered list of possible queries, we find that the query on top of the list retrieves the

[6] `http://disi.unitn.it/~moschitt/Tree-Kernel.htm`

[7] Available at http://www.cs.utexas.edu/ ml/geo.html

[8] This are the first 700 questions of the 880 ones that Mooney's group [14] paired with logical formulas in Prolog and that Popescu et al. [10] manually converted into SQL.

correct result set in 82% of the cases. For the other questions, it can be found within the first 10 generated answers for 99% of the questions (once the 33 questions above have been removed). This can be observed in Figure 6, which plots the Recall (of the correct question) curve of the generative approach, i.e., the baseline. As pointed out in the graphic, the right query is found among the first three in 93% of the cases.

6.3 Reranking Results

Figure 6 also shows the plot for different rerankers using the following kernels: STK+STK, STK×STK and (1+STK×STK)^2, which provide better rankings (the first STK is applied to the question parse trees whereas the second STK is applied to the query derivation tree). For example, the latter kernel retrieved the correct answers 94% of times by only using the first two answers.

To better evaluate the results of our rerankers, we applied standard 10-fold cross validation and measure the average Recall and Std Dev. of selecting a query for each question. The results for different kernel models for reranking are reported in Table 2. The first column of Table 2 lists kernel combination by means of product and sum between pairs of basic kernels used for the question and the query, respectively. The other columns show the percentage of questions for which we found at least 1 correct answer in the top @X positions (average Recall@X over 10 folds ± Std. Dev).

The results are rather exciting since they compare favorably with the state-of-the-art. The best system on this datasets was designed in [15] and shows a Precision of 96.3% and a Recall of 79.3%, for an f-measure of 86.9%, while our system shows a Precision of 82.8% and a Recall of 87.2%, for an f-measure of 85.0% (when we include the 33 missing questions in the evaluation). Two main facts should be noted:

- our system performs just 2 points less than the system designed in [15] but it does not need any hand-crafted manual resource, i.e., the semantic trees manually designed in [15] for each question, and it is very simple to implement.
- unlike it has been done in previous work, we can also provide multiple ranked answers. If we select the first n candidates, we highly increasing the Recall

Table 2. Kernel combination recall (± Std. Dev) for GEO dataset

Combination	**Rec@1**	**Rec@2**	**Rec@3**	**Rec@4**	**Rec@5**
NO RERANKING	81.4±5.8	87.6±3.8	90.8±3.1	94.0±2.4	95.0±2.0
STK + STK	83.5±3.6	90.4±3.5	94.2±2.9	95.8±2.0	96.7±1.7
STK × STK	86.5±4.0	92.6±3.7	95.3±3.2	97.0±1.8	97.7±1.4
(1+STK^2)^2	87.2±3.9	94.1±3.4	95.6±2.7	97.1±1.9	97.9±1.4
BOW × STK	86.7±4.1	92.1±3.2	95.6±2.5	97.1±1.4	97.6±1.2

of the correct answers, e.g., within the first 2 we have a f-measure of 90% (considering the 33 missing questions).

Other closely related work, e.g., [4], suggests that lower results than ours can be obtained using different approaches. These rely either on semantic grammar specified by an expert user [9], or on enriching the information contained in the pairs [10] and implementing ad-hoc rules in a semantic parser [7,11]. Our system instead, requires no intervention since the database metadata already contains all the needed data.

Finally, we report the learning curve of one basic reranker in Figure 7, showing how recall of STK×STK increases for larger training sets. The plot reveals that as soon as we provide a reasonable percentage of training data (25% of the available data corresponding to 9 folds of 700 questions – one fold is used for testing) for reranking, the model improves on the baseline.

The main contribution of this research consist in the fact that given a NL question we can generate a set of mapping SQL queries. Moreover if we can rely on a relatively small set of correct pairs of questions and queries to train a SVM classifier, we are able to re-rank the set of generated pairs to select the correct one with a fairly high accuracy.

7 Conclusions and Future Work

In this paper, we have approached the question answering task of implementing a NL interface to databases by automatically generating SQL queries based on grammatical relations and matching metadata. To our knowledge, the underlying idea that we have proposed to build and combine clauses sets is novelty. Additionally, we are firstly experimented with a preference reranking kernel, which is able to boost the accuracy of our generative model.

Given the high accuracy, the simplicity and the practical usefulness of our approach, (e.g., we can generate the correct question in the first 5 candidates in 95% of the cases), we believe that our methods can be successfully used in the future for real-world applications.

In the future we plan to experiment with datasets in different domains (e.g. ATIS corpus). Moreover, given that current challenges in Semantic Web tackle similar problem [5] (scaling question answering approaches to Linked Data, i.e. Question Answering over Linked Data), it would be interesting to apply our algorithms to semantic search and question answering over RDF data.

References

1. Charniak, E.: A maximum-entropy-inspired parser. In: Proceedings of NAACL 2000 (2000)
2. Collins, M., Duffy, N.: New ranking algorithms for parsing and tagging: Kernels over discrete structures, and the voted perceptron. In: Proceedings of ACL 2002 (2002)

3. Garcia-Molina, H., Ullman, J.D., Widom, J.: Database Systems: The Complete Book, 2nd edn. Prentice Hall Press, Upper Saddle River (2008)
4. Giordani, A., Moschitti, A.: Corpora for automatically learning to map natural language questions into sql queries. In: Proceedings of LREC 2010, Valletta, Malta. European Language Resources Association (ELRA) (May 2010)
5. Granberg, J., Minock, M.: A natural language interface over the musicbrainz database. In: Proceedings of the 1st Workshop on Question Answering over Linked Data (QALD-1): Co-located with the 8th Extended Semantic Web Conference, pp. 38–43 (2011), QC 20120413
6. Joachims, T.: Making large-scale SVM learning practical. In: Schölkopf, B., Burges, C., Smola, A. (eds.) Advances in Kernel Methods (1999)
7. Kate, R.J., Mooney, R.J.: Using string-kernels for learning semantic parsers. In: Proceedings of the 21st ICCL and 44th Annual Meeting of the ACL, Sydney, Australia, pp. 913–920. Association for Computational Linguistics (July 2006)
8. MacCartney, B., de Marneffe, M.-C., Manning, C.D.: Generating typed dependency parses from phrase structure parses. In: Proceedings LREC 2006 (2006)
9. Minock, M., Olofsson, P., Näslund, A.: Towards building robust natural language interfaces to databases. In: Kapetanios, E., Sugumaran, V., Spiliopoulou, M. (eds.) NLDB 2008. LNCS, vol. 5039, pp. 187–198. Springer, Heidelberg (2008)
10. Popescu, A.-M., Etzioni, O.A., Kautz, H.A.: Towards a theory of natural language interfaces to databases. In: Proceedings of the 2003 International Conference on Intelligent User Interfaces, Miami. Association for Computational Linguistics (2003)
11. Ruwanpura, S.: Sq-hal: Natural language to sql translator
12. Salton, G.: Recent trends in automatic information retrieval. In: Proceedings of the 9th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval, SIGIR 1986, Pisa, Italy, September 8-10, pp. 1–10. ACM (1986)
13. Shen, L., Joshi, A.K.: An SVM-based voting algorithm with application to parse reranking. In: Proceedings of the Seventh Conference on Natural Language Learning at HLT-NAACL 2003, pp. 9–16 (2003)
14. Tang, L.R., Mooney, R.J.: Using multiple clause constructors in inductive logic programming for semantic parsing. In: Flach, P.A., De Raedt, L. (eds.) ECML 2001. LNCS (LNAI), vol. 2167, pp. 466–477. Springer, Heidelberg (2001)
15. Zettlemoyer, L.S., Collins, M.: Learning to map sentences to logical form: Structured classification with probabilistic categorial grammars. In: UAI, pp. 658–666 (2005)