Semantic Mapping Between Natural Language Questions and SQL Queries via Syntactic Pairing

Alessandra Giordani and Alessandro Moschitti

Department of Computer Science and Engineering
University of Trento
Via Sommarive 14, 38100 POVO (TN) - Italy
{agiordani,moschitti}@disi.unitn.it

Abstract. Automatically mapping natural language into programming language semantics has always been a major and interesting challenge. In this paper, we approach such problem by carrying out mapping at syntactic level and then applying machine learning algorithms to derive an automatic translator of natural language questions into their associated SQL queries. For this purpose, we design a dataset of relational pairs containing syntactic trees of questions and queries and we encode them in Support Vector Machines by means of kernel functions. Pair classification experiments suggest that our approach is promising in deriving shared semantics between the languages above.

Key words: Natural Language Interfaces for Database Querying; Kernel Methods; Tree Kernels

1 Introduction

Automatically mapping natural language into programming language semantics is a major and interesting challenge in the field of computational linguistics since it may have a direct impact on industrial and social worlds. For example, accessing a database requires machine-readable instructions that not everybody is supposed to know. Users should be able to pose a question in natural language without knowing either the underlying database schema or any complex structured machine language. The development of natural language interfaces over databases (NLIDBs), that translate the human intent into machine instructions used to answer questions, is indeed a classic problem that is becoming of greater importance in today's world.

This could be addressed by finding a mapping between natural language and the database programming language. If we know how to convert natural language questions into their associated SQL queries, it would be straightforward to obtain the answers by just executing a query. Unfortunately, previous work has shown that a full semantic approach to this problem cannot be applied, therefore shallow and statistical methods are required.

In this paper, we exploit mapping at syntactic level between the two languages and apply machine learning algorithms to derive the shared shallow semantics. We design a dataset of question and query pairs and represent them by

means of syntactic parse trees of the two respective languages. To encode trees in the learning algorithm, we use two different types of tree kernels and linear kernels, which are applied to pairs by means of advanced combinations. We carried out cross-validation experiments on the task of selecting correct queries given a target set of questions. The results show that our best kernel combination improves the baseline model of about 32%. The latter is a typical approach based on a linear kernel applied to the union of the bag-of-words from question and query texts. The most interesting finding is that the product between the two kernels representing questions and queries provides feature pairs, which can express the relational features between the syntactic/semantic representation of the two languages.

In the remainder, Section 2 introduces the problem of mapping questions into queries and illustrates the idea of our solution whereas Section 3 describes the technology to implement it, i.e. kernel methods. Section 4 shows our proposed algorithm to generate a training set of question and query pairs, Section 5 discusses our results and finally, Section 6 draws conclusions.

2 Automatic Mapping of Questions into SQL Queries

Studying the automatic mapping of questions into SQL queries is important for two main reasons: (a) it allows to design interesting applications based on databases and (b) it offers the possibility to understand the role of syntax in deriving a shared semantics between a natural language and an artificial language.

Given the complexity of theoretically modeling such relationship we use a statistical and shallow model. We consider a dataset of natural language questions $\mathcal{N}$ and SQL queries $\mathcal{S}$ related to a specific domain/database and we automatically learn such mapping from the set of pairs $\mathcal{P} = \mathcal{N} \times \mathcal{S}$. More in detail, (a) we assume that pairs are annotated as correct when the SQL query answers to the question and incorrect otherwise and (b) we train a classifier on the above pairs for selecting the correct queries for a question. Then, to map new questions in the dataset of the available queries, (c) we rank the latter by means of the question classifier score and by selecting the top one. In the following we provide the formal definition of our learning approach.

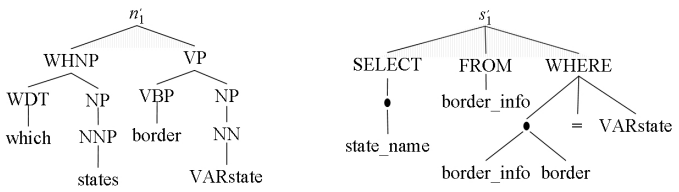

Fig. 1. Question/Query Syntactic trees

2.1 Pair Ranking

The problem of assigning a query (with its result) to a question, can be formally described as the following ranking problem: (i) given a question $n \in \mathcal{N}$ and a set of possible useful queries $\mathcal{S}$, we generate the set of possible pairs $P(n) = \{\langle n, s\rangle : s \in \mathcal{S}\}$; (ii) we classify them with an automatic categorizer; (iii) we use the score/probability output by such model to rank $P(n)$; (vi) we select the top ranked pairs.

For example, let's consider question n_1: "*Which states border Texas?*" and the following queries s_1: `SELECT state_name FROM border_info WHERE border='texas'` and s_2: `SELECT COUNT(state_name) FROM border_info WHERE border='texas'`. Since s_1 is a correct and s_2 is an incorrect interpretation of the question, the classifier should assign a higher score to the former, thus our ranker will output the $\langle n_1, s_1\rangle$ pair. Note that both s_1 and s_2 share three terms, *state, border and texas*, with n_1 but $\langle n_1, s_2\rangle$ is not correct. This suggests that we can't only rely on the common terms but we should also take into account the syntax of both languages.

1 SQL →SELECT ItemList FROM TableList |
SELECT ItemList FROM TableList WHERE Cond
...
5 ItemList → ItemList , Item | Item
6 Item → Table . Column | Column
7 Column → * | ColumnName
...
15 WhereCondition → AndCondition AND AndCondition
16 AndCondition → Condition OR Condition
17 Condition → NOT Condition | Operand | ...
18 Operand → Factor | Operand + Factor | ...
19 Factor → Term | Factor * Term | ...
20 Term → Value | SelectItem | SQL
21 SelectItem → Table . ColumnName | ColumnName

5* ItemList → ItemList • | •
6* • → Table | Column | Table Column
20* Term → Value | • | SQL

Fig. 2. Modified MySQL Grammar

2.2 Pair Representation

The aim of our research is to derive the shared shallow semantics in pairs by means of syntax. Thus we represent questions and queries using their syntactic trees, as shown in Figure 1: for the question (a) we use the output the Charniak's syntactic parser [1] whereas for the query (b) we use a modification of the SQL derivation tree.

To build the SQL tree we implemented an ad-hoc parser that follows the syntactic derivation of a query according to our grammar. Since our database system embeds a MySQL server, we use the production rules of MySQL, shown at the top of Figure 2, slightly modified to manage punctuation, i.e. rules 5*, 6* and 20* related to comma and dot, as shown at the bottom.

More in detail, we change the non-terminals *Item* and *SelectItem* with the symbol • to have an uniform representation for the relation between a table and its column in both the `SELECT` and `WHERE` clauses. This allows for matching between the subtrees containing table, column or both also when they appear in different clause types of two queries.

It is worth noting that rule 20* still allows to parse nested queries and that the overall grammar, in general, is very expressive and powerful enough to express complex SQL queries involving nesting, aggregation, conjunctions and disjunctions in the `WHERE` clause.

Note that, although we eliminated comma and dot from the original SQL grammar, it is still possible to obtain the original SQL query, by just performing a preorder traversal of the tree.

To represent the above structures in a learning algorithm we use tree kernels described in the following section.

3 Tree Kernels

Kernel Methods refer to a large class of learning algorithms based on inner product vector spaces, among which Support Vector Machines (SVMs) are one of the most well-known algorithms. The main idea is that the parameter model vector $\boldsymbol{w}$ generated by SVMs (or by other kernel-based machines) can be rewritten as

$$\sum_{i=1..l} y_i \alpha_i \boldsymbol{x}_i, \tag{1}$$

where y_i is equal to 1 for positive and -1 for negative examples, $\alpha_i \in \Re$ with $\alpha_i \geq 0$, $\forall i \in \{1, .., l\}$ $\boldsymbol{x}_i$ are the training instances.

Therefore we can express the classification function as

$$Sgn(\sum_{i=1..l} y_i \alpha_i \boldsymbol{x}_i \cdot \boldsymbol{x} + b) = Sgn(\sum_{i=1..l} y_i \alpha_i \phi(o_i) \cdot \phi(o) + b), \tag{2}$$

where $\boldsymbol{x}$ is a classifying object, b is a threshold and the product $K(o_i, o) = \langle \phi(o_i) \cdot \phi(o) \rangle$ is the kernel function associated with the mapping ϕ.

Note that it is not necessary to apply the mapping ϕ, we can use $K(o_i, o)$ directly. This allows, under the Mercer's conditions [2] for defining abstract functions which generate implicit feature spaces. The latter allow for an easier feature extraction and the use of huge feature spaces (possibly infinite), where the scalar product (i.e. $K(\cdot, \cdot)$) is implicitly evaluated.

In the remainder of this section, we illustrate some kernels for structured data: the Syntactic Tree Kernel (STK) [3], which computes the number of syntactic tree fragments and the Extended Syntactic Tree Kernel (STK_e) [4], which includes leaves in STK. In the last subsection we show how to engineer new kernels from them.

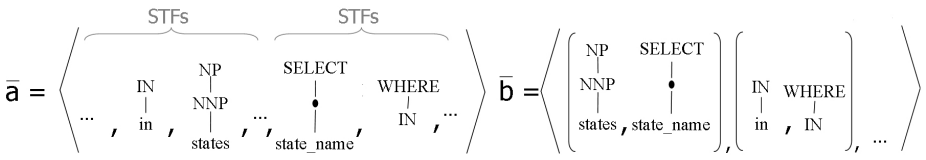

Fig. 3. Feature spaces for the tree pair in Figure 1 a) joint space STK+STK b) Cartesian product STK×STK

3.1 Syntactic Tree Kernel (STK) and its Extension (STK$_e$)

The main underlying idea of tree kernels is to compute the number of common substructures between two trees T_1 and T_2 without explicitly considering the whole fragment space. Let $\mathcal{F} = \{f_1, f_2, \ldots, f_{|\mathcal{F}|}\}$ be the set of tree fragments and $\chi_i(n)$ an indicator function equal to 1 if the target f_i is rooted at node n and equal to 0 otherwise. A tree kernel function over T_1 and T_2 is defined as

$$TK(T_1, T_2) = \sum_{n_1 \in N_{T_1}} \sum_{n_2 \in N_{T_2}} \Delta(n_1, n_2), \tag{3}$$

where N_{T_1} and N_{T_2} are the sets of nodes in T_1 and T_2, respectively, and $\Delta(n_1, n_2) = \sum_{i=1}^{|\mathcal{F}|} \chi_i(n_1)\chi_i(n_2)$.

The Δ function is equal to the number of common fragments rooted in nodes n_1 and n_2, and thus, depends on the fragment type. We report its algorithm for the evaluation of the number of syntactic tree fragments (STFs) [3].

A syntactic tree fragment (STF) is a set of nodes and edges from the original tree which is still a tree and with the constraint that any node must have all or none of its children. This is equivalent to state that the production rules contained in the STF cannot be partial.

To compute the number of common STFs rooted in n_1 and n_2, the STK uses the following Δ function [3]:
1. if the productions at n_1 and n_2 are different then $\Delta(n_1, n_2) = 0$;
2. if the productions at n_1 and n_2 are the same, and n_1 and n_2 have only leaf children (i.e. they are pre-terminal symbols) then $\Delta(n_1, n_2) = \lambda$;
3. if the productions at n_1 and n_2 are the same, and n_1 and n_2 are not pre-terminals then
$\Delta(n_1, n_2) = \lambda \prod_{j=1}^{l(n_1)} (1 + \Delta(c_{n_1}(j), c_{n_2}(j)))$,
where $l(n_1)$ is the number of children of n_1, $c_n(j)$ is the j-th child of node n and λ is a decay factor penalizing larger structures.

Figure 3.a shows some STFs of the left tree in Figure 1. STFs satisfy the constraint that grammatical rules cannot be broken.

STK does not include individual nodes as features. As shown in [4] we can include at least the leaves, (which in constituency trees correspond to words) by simply inserting the following step 0 in the algorithm above:
0. if n_1 and n_2 are leaf nodes and their labels are identical then $\Delta(n_1, n_2) = \lambda$;

3.2 Kernel Engineering

Kernel engineering can be carried out by combining basic kernels with additive or multiplicative operators or by designing specific data objects, e.g. the tree representation for the SQL syntax, to which standard kernels are applied. Since our data is a set of 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. For example, the explicit vector representation of the STK + STK space of the pair in Figure 1 is shown in Figure 3.a. The Syntactic Tree Fragments of the question will be in the same space of the Syntactic Tree Fragments of the query.

In theory a more effective kernel is the product $k(n_1, n_2) \times k(s_1, s_2)$ since it generates pairs of fragments as features, where the overall space is the cartesian product of the used kernel spaces. For example Figure 3.b shows pairs of STF fragments, which are essential to capture the relational semantics between the syntactic tree subparts of the two languages. In particular, the first fragment pair of the figure may suggest that a noun phrase composed by *state* expresses similar semantics of the syntactic construct `SELECT state_name`.

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.

4 Dataset Generation

In the previous sections we have defined models to automatically learn the mapping between questions and queries. Since we aim at having very accurate approaches we apply supervised techniques and, consequently, we need training data. More precisely, we need correct and incorrect pairs of questions and queries. Since in practical applications this is the most costly aspect, we should generate such learning set in a smart way. In this perspective, we consider that, in real world domains, we may expect to have examples of questions and the associated queries which answer to such information need. Such pairs may have been collected when users and operators of the database worked together for the accomplishment of some tasks. In contrast, we cannot assume to have available pairs of incorrect examples, since (a) the operator tends to immediately provide the correct query and (b) both users and operators do not really understand the use of negative examples and the need to have unbiased distribution of them.

Therefore, we need techniques to generate negative examples from an initial set of correct pairs. Unfortunately, this is not a trivial task since when mixing a question and query belonging to different pairs we cannot assume to only generate incorrect pairs, e.g. when swapping x with y in the two pairs $\langle$ *Which states border Texas?*, $x\rangle$ and $\langle$ *What are the states bordering Texas?*, $y\rangle$, we obtain other two correct pairs.

To generate a gold standard dataset we would need to manually check this aspect thus we design an algorithm to limit the human supervision. It consists of the following steps:

- Generalizing question and query instances: substitute the involved concepts in questions and their related field values in the SQL queries by means of variables (expressing the category of such values).
- Clustering the generalized pairs: intuitively each cluster represents the information need about a target semantic concept, e.g. "bordering state", common to questions and queries. This requires a limited manual intervention.
- Pairing questions and queries of distinct clusters, i.e. the cartesian product between the set of questions and the set of queries belonging to the pairs of a target cluster. This allows to find new positive examples that were not present in the initial corpus.
- Final dataset annotation: consider all possible pairs, i.e. cartesian product between all the questions and queries of the dataset, and annotate them as negatives if they have not been annotated as positives in the previous step.

We use the GeoQueries[1] corpus translated by Popescu et al. [5] as our initial dataset. It consists of 250 pairs of NL questions and SQL queries over a small database about United States geography. In the following we describe in detail all the steps through which the final dataset is generated.

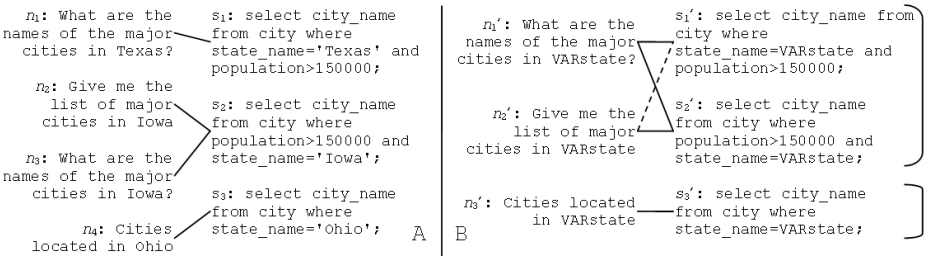

Fig. 4. Example of the initial corpus (*A, on the left*) and the generalized version (*B, on the right*). The latter is divided in two clusters (*identified by the two brackets*).

4.1 Pair Generalization

Our approach to automatically annotate pairs relies on automatically detecting if swapping members of different pairs produces correct or incorrect examples. For this purpose, we detect similar syntactic structures of questions and queries by generalizing concept instances with variables.

In a database concepts are represented as tables' fields. These are used in SQL to select data satisfying some conditions, i.e. concept constrained to a value.

[1] http://www.cs.utexas.edu/users/ml/nldata.html

Typically these values are natural language terms so we substitute them with variables if they appear in both questions and queries. For example, consider s_1 in Figure 1. The condition is `WHERE state_name = 'Texas'` and 'Texas' is the value of the concept state_name. Since 'Texas' is also present in the related question we can substitute it with a variable *VARstate* (one variable for each different concept). Our assumption is that questions whose answer can be retrieved in a database, tend to use the same terms stored in the database.

An example of the generalization phase is shown in Figure 4. On the left there is a set of four pairs containing four distinct questions and three related queries (connected by the lines) whereas on the right four generalized pairs are shown. We note that, after substituting instances with variables, both n_1 and n_3 are generalized into n'_1, which is thus paired with two distinct SQL queries, i.e. s'_1 and s'_2. This is not surprising since there can be more SQL queries that correctly retrieve an answer to a NL question. In this case we define them to be *semantically equivalent*, i.e. $s'_1 \equiv s'_2$. At the same time it is possible to write many NL questions that map to the same query.

It is worth noting that with the generalization process, we introduce redundancy that we eliminate by removing duplicated questions and queries. Thus, the output dataset is usually smaller than the initial one. However the number of training examples will be larger, not only because of the introduction of negatives but also due to the automatic discovering of new positives.

4.2 Pair Clustering and Final Dataset Annotation

Once the pairs have been generalized, we cluster them according to their semantic equivalence so that we can automatically derive new positive examples by swapping their members. We define semantically equivalent pairs those correct pairs with (a) equivalent NL questions or (b) equivalent SQL queries. Given that two equivalent queries should retrieve the same result set, we can automatically test their equivalence by simply executing them. Unfortunately, this is just a necessary condition (e.g. 'Texas' can be the answer of two different queries) therefore we manually evaluate new pairings obtained applying this condition.

Note that automatically detecting semantic equivalence of natural language questions with perfect accuracy is a hard task, so we consider as semantically equivalent either identical questions or those associated with semantic equivalent queries. We also apply transitivity closure to both members of pairs to extend the set of equivalent pairs.

For example, in Figure 4.b s'_1 and s'_2 retrieve the same results so we verify that they are semantically equivalent queries and we assign them to the same cluster (CL1), i.e. information need about the large cities of a state (with a population larger than 150,000 people). Alternatively, we can also consider that n'_1 and n'_2 are both paired with s'_2 to derive that they are equivalent, avoiding the human intervention. Concerning s'_3, it retrieves a result set different form the previous one so we can automatically assign it to a different cluster (CL2), i.e. involving questions about any city of a state. Note that, once n'_2 is shown to be semantically equivalent to n'_1 we can pair them with s'_1 to create the new

```
function CreateFinalDataset(LIST OF PAIRS I) returns MATRIX M
M[|𝒩|][|𝒮|] = -1  // stores the cluster id for each pair, it is initialized to -1
k=0               // at the beginnig the number of cluster is zero
begin
∀ ⟨n,s⟩ ∈ I s.t. M[n][s] = -1 do   // for all initial pairs not clustered yet
   begin
   k:=k+1                          // add a new cluster
   M[n][s]:=k                      // the pair ⟨n,s⟩ belongs to the new cluster
   ∀ ⟨n′,s′⟩ ∈ I do                // find other pairs that belong to it
   if (n = n′) OR (s = s′)         // if identical (indeed equivalent)
     then M[n′][s′]:=k             // also ⟨n′,s′⟩ belongs to the cluster
     else                          // check equivalence if similar result set
       if (fields(s) = fields(s′)) AND (min(s) = min(s′))
         then if res(s) = res(s′)  // if s and s′ retrieve the same result set
           then if n ≡ n′          // if the associated sentences are equivalent
             then M[n′][s′]:=k     // label also the pair ⟨n′,s′⟩ with this cluster id
   end
  ∀ k do                           // find all pairings within a cluster
   ∀ n, n′, s, s′ s.t. M[n][s]=k AND M[n′][s′]=k
    begin
    M[n][s′]:=k                    // swap all queries
    M[n′][s]:=k                    // swap all questions
    end
  return M
end
```

Fig. 5. Clustering Algorithm

pair highlighted with the dashed relation $\langle n'_2,s'_1\rangle$. Indeed the negative example set is $\langle n'_3, s'_1\rangle, \langle n'_3, s'_2\rangle, \langle n'_1, s'_3\rangle, \langle n'_2, s'_3\rangle$.

The above steps are formally described by the algorithm in Figure 5. It takes as input the generalized dataset as a list of correct pairs $I \subset \{\langle n, s\rangle : n \in \mathcal{N}, s \in \mathcal{S}\}$ and returns a matrix M storing all positive and negative pairs $\mathcal{P} = \mathcal{N} \times \mathcal{S}$. M is obtained by (a) dividing I in k clusters of semantically related pairs and (b) applying the transitive closure to the semantic relationship between member pairs. More in detail, we first initialize its entries with a negative value, i.e. $M[n, s] = -1 \forall n \in \mathcal{N}, s \in \mathcal{S}$.

Second, we group together each $\langle n, s\rangle$ and $\langle n', s'\rangle \in I$, if at least two of their members are identical. If not we test if the two query members, s and s' retrieve the same result set. Since this may be time consuming we run this test only if the selected columns in their `SELECT` clause are the same and if the two result sets share the same minimum.

Third, since the condition above is only necessary for semantic equivalence, in case we find the same result sets, we manually check if the natural language question members are semantically equivalent. This is faster and easier than checking the SQL queries.

Finally, once the initial clusters have been created, we apply the transitive closure to the cluster c_k to include all possible pairing between questions and queries belonging to c_k, i.e. $c_k = \{\langle n, s\rangle : n, s \in c_k\}$. We store in M the id of the clusters in the related pairs, i.e. $M[n][s] = k$ for each c_k. As a side effect all entries of M still containing -1 will be negative examples.

5 The Experiments

In these experiments, we study the effectiveness of our algorithm for automatically mapping questions into queries by testing the accuracy of selecting for each question of the test set its correct query. For this purpose, we learn a classifier of correct and incorrect pairs and we use it as a ranker for the possible queries as described in Section 2.

5.1 Setup

The query ranker consists in an SVM using advanced kernels for representing question and query pairs. We implemented the Syntactic Tree Kernel (STK) and it's extension (STK_e) described in Section 3 and several combinations in SVM-Light [6] software.

As test set, we use our dataset obtained from GeoQueries by applying our algorithm described in Section 4. After the generalization process the initial 250 pairs of questions/queries are reduced to 155 pairs containing 154 different NL question and 80 different SQL queries. We found 76 clusters, from which we generated 165 positive and 12.001 negative examples for a total of 154 $\times$ 70 pairs. Since the number of negatives is much greater than the positives, we have to eliminate negative pairings from the test set such that the learning is more efficient. Moreover, since we want to test the model with feasible pairs we have to preprocess the pairs to reduce the test set. We addressed these problems by keeping only pair that share at least 2 common stems since, intuitively, a positive pairing share at least a variable and a concept (eg. *VARstate cities*). Actually, we don't commit any error in the pre-processing if we classify a pair as incorrect when its question and query share only one stem or no stem at all. From our automatically generated negative examples set we excluded 10.685 pairs, reducing it to 1.316 examples.

To evaluate the results of the automatic mapping, we applied standard 10-fold cross validation and measure the average accuracy and the Std. Dev. of selecting the correct query for each question of the test set.

5.2 Results

We tested several models for ranking based on different kernel combinations whose results are reported in tables Table 1 and Table 2. The first two columns of Table 1 show the kernels used for the question and the query, respectively. More in detail, our basic kernels are: (1) linear kernel (LIN) built on the bag-of-words (BOW) of the questions or of the query, e.g. `SELECT` is considered a feature for the query; (2) a polynomial kernel of degree 3 on the above BOWs (POLY); (3) the Syntactic Tree Kernel (STK) on the parse tree of the question or the query and (4) STK extended with leaf features (STK_e).

Columns 3 and 4 show the average accuracy (over 10 folds) $\pm$ Std. Dev. of two main kernel combinations by means of product and sum. Note that we can also sum or multiply different kernels, e.g. LIN$\times STK_e$.

Table 1. Kernel combination accuracy

K1	**K2**	**K1×K2**	**K1+K2**
LIN	LIN	70.7±12.0	57.3±10.4
POLY	POLY	71.9±11.5	55.1±8.4
STK	STK	70.3±9.3	54.9±10.1
STK_e	STK_e	70.1±10.9	56.7±12.0
LIN	STK	74.6±9.6	56.8±10.0
LIN	STK_e	75.6±13.1	56.6±12.4
POLY	STK	73.8±9.5	56.4±10.1
POLY	STK_e	73.5±10.4	56.5±10.0
STK	LIN	64.7±11.5	56.7±9.7
STK_e	LIN	68.3±9.6	56.2±11.7
STK	POLY	65.4±10.9	55.2±9.5
STK_e	POLY	68.3±9.6	56.2±11.7

Table 2. Kernel engeneering results

Advanced Kernels	**Accuracy**
STK^2+POLY^2	72.7±9.7
$STK_e^2+POLY^2$	73.2±11.4
$(1+LIN^2)^2$	73.6±9.4
$(1+POLY^2)^2$	73.2±10.9
$(1+STK^2)^2$	69.4±10.0
$(1+STK_e^2)^2$	70.0±12.2
$(1+LIN^2)^2+STK^2$	75.0±10.8
$(1+POLY^2)^2+STK^2$	72.6±10.5
$(1+LIN^2)^2+LIN\times STK$	75.9±9.6
$(1+POLY^2)^2+POLY\times STK$	73.2±10.9
$POLY\times STK+STK^2+POLY^2$	73.9±11.5
$POLY\times STK_e+STK_e^2+POLY^2$	75.3±11.5

An examination of the reported tables suggests that: first, the basic traditional model based on linear kernel and BOW, i.e. LIN + LIN, provides an accuracy of only 57.3%, which is greatly improved by $LIN\times LIN=LIN^2$, i.e. by 13.5 points [2].

The explanation is that the sum cannot express the relational feature pairs coming from questions and queries, thus LIN cannot capture the underlying shared semantics between them. It should be noted that only kernel methods allow for an efficient and easy design of LIN^2, since the traditional approach would have required to build the cartesian product of the question BOW by query BOW. This can be very large, e.g. 10K features for both spaces leads to a pair space of 100M features.

Second, the K_1+K_2 column confirms that the feature pair space is essential since the accuracy of all kernels implementing individual spaces (e.g. kernels which are sums of kernels) is much lower than the baseline model for feature pairs, i.e. LIN^2.

Third, if we include conjunctions in the BOW representation by using POLY, we improve the LIN model, when we use the feature pair space, i.e. 71.9% vs 70.8%. Also, $POLY^2$ is better than STK^2 since it includes individual terms/words, which are not included by STK.

Next, the above point suggests that syntactic models can improve BOW although too many syntactic features (generated by STK) make the model unstable as suggested by the lower accuracy (70.1%) provided by $STK_e\times STK_e=STK_e^2$. This consideration leads us to experiment with the model LIN × STK and LIN × STK_e, which combine words of the questions with syntactic constructs of SQL queries. They produce high results, i.e. 74.6% and 75.6%, and the difference with

[2] Although the Std. Dev. associated with the model accuracy is high, the one associated with the distribution of difference between the model accuracy is much lower, i.e. 5%

previous models is statistical significant (90% confidence interval). This suggests that the syntactic parse tree of the SQL query is very reliable (it is obtained with 100% of accuracy) while the natural language parse tree, although accurate, introduces noise that degrades the overall feature representation. As a consequence it is more effective to use words only in the representation of the first member of the pairs. This is also prooved by the last four lines of Table 1, showing the low accuracies obtained when relying on NL synctactic parse trees and SQL BOWs. However, POLY $\times$ STK_e performs worse than the best basic model LIN $\times$ STK_e (80% confidence level).

Finally, we experimented with very advanced kernels built on top of feature pair spaces as shown in Table 2. For example, we sum different pair spaces, STK_e^2 and $POLY^2$, and we apply the polynomial kernel on top of pair spaces by creating conjunctions, over feature pairs. This operation tends to increase too much the cardinality of the space and makes it ineffective. However, using the simplest initial space, i.e. LIN, to build pair conjunctions, i.e. $(1+LIN^2)^2$, we obtain a very interesting and high result, i.e. 73.6% (statistically significant with a confidence of 90%). Using the joint space of this polynomial kernel and of simple kernel products we can still improove our models.

This suggests that kernel methods have the potentiality to describe relational problems using simple building blocks although new theory describing the degradation of kernels when the space is too complex is required.

Finally, to study the stability of our complex kernels, we compared the learning curve of the baseline model, i.e. LIN+LIN, with the those of best models, i.e. LIN$\times$$STK_e$ and $STK^2+(1+LIN^2)^2$. Figure 6 shows that complex kernels are not only more accurate but also more stable, i.e. their accuracy grows smoothly according to the increase of training data.

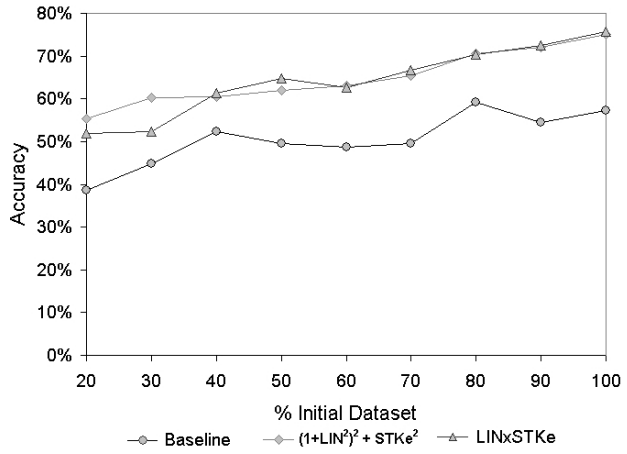

Fig. 6. Learning curves for GeoQueries corpora

5.3 Related Work

As the literature suggest, NLiDBs can be classifyied according to the approach employed in deriving an SQL query that retrieves the answer of a given NL question against a database. In this section we review three systems based on different approaches and that were also tested on the GeoQueries. For a complete review of many NLiDB refer to Chandra and Mihalcea [7].

Systems based on an authoring rely on semantic grammar specified by an expert user to interpret question over the database. CatchPhrase [8] is an authoring tool where the author is asked to name database elements, tailor entries and define additional concepts. According to the authors this tool achieves 80% recall and 86% precision.

Another approach is based on annotation. An example of a system that uses this approach is Precise [5]. Reducing the problem of finding a semantic interpretation of ambiguous phrases to a graph matching problem, authors achieves 100% precision on a subset of semantically tractable questions (77,5% recall).

The machine learning approach, that induces semantic grammar from a corpus of correct pairs of questions and queries, has been used in Krisp [9]. Krisp performs semantic parsing mapping sentences into their computer-executable meaning representations. For each production in the meaning representation language it trains an SVM classifier based on string subsequence kernels. Then it uses these classifiers to compositionally represent a natural language sentence in their meaning representations. Krisp achieves approximatively 94% precision and 78% recall. Our system, also based on the machine learning approach, doesn't decline to answer any questions and shows an accuracy of 76% when the SQL query of the pair with the highest rank is executed to retrieve the answer of the paired question.

Regarding the use of tree kernels for natural language tasks several models have been proposed and experimented [3, 10–18].

6 Conclusions

In this paper, we approach the problem of mapping natural into programming language semantics by automatically learning a model based on lexical and syntactic description of the training examples. In our study, these are pairs of NL questions and SQL queries, which we annotated by means of our semi-supervised algorithm based on the initial annotation available in the GeoQueries corpus.

To represent syntactic/semantic relationships expressed by the pairs above, we largely adopted kernel methods along with SVMs. We designed innovative combinations between different kernels for structured data applied to pairs of objects. To our knowledge, the functions that we propose for relational semantics description are novel. The experiments of the automatic question translation system show a satisfactory accuracy, i.e. 76%, although large improvement are still possible.

The main contributions of our study are: (i) we show that our automatic mapping between question and SQL queries is viable, (ii) in at least one task

we have proved that kernel products are effective, (iii) syntax is important to map natural language into programming languages and (iv) we have generated a corpus for future studies, which we are going to make publically available.

References

1. Charniak, E.: A maximum-entropy-inspired parser. In: Proceedings of NAACL'00. (2000)
2. Shawe-Taylor, J., Cristianini, N.: Kernel Methods for Pattern Analysis. Cambridge University Press (2004)
3. Collins, M., Duffy, N.: New ranking algorithms for parsing and tagging: Kernels over discrete structures, and the voted perceptron. In: Proceedings of ACL'02. (2002)
4. Zhang, D., Lee, W.S.: Question classification using support vector machines. In: Proceedings of the 26th annual international ACM SIGIR conference on Research and development in informaion retrieval, ACM Press (2003) 26–32
5. Popescu, A.M., A Etzioni, O., A Kautz, H.: 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) 149–157
6. Joachims, T.: Making large-scale SVM learning practical. In Schölkopf, B., Burges, C., Smola, A., eds.: Advances in Kernel Methods. (1999)
7. Chandra, Y., Mihalcea, R.: Natural language interfaces to databases, University of North Texas, Thesis (M.S.) (2006)
8. Minock, M., Olofsson, P., Näslund, A.: Towards building robust natural language interfaces to databases. In: NLDB '08: Proceedings of the 13th international conference on Natural Language and Information Systems, Berlin, Heidelberg (2008)
9. 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, Association for Computational Linguistics (July 2006) 913–920
10. Kudo, T., Matsumoto, Y.: Fast Methods for Kernel-Based Text Analysis. In Hinrichs, E., Roth, D., eds.: Proceedings of ACL. (2003) 24–31
11. Cumby, C., Roth, D.: Kernel Methods for Relational Learning. In: Proceedings of ICML 2003, Washington, DC, USA (2003) 107–114
12. Culotta, A., Sorensen, J.: Dependency Tree Kernels for Relation Extraction. In: ACL04, Barcelona, Spain (2004) 423–429
13. Kudo, T., Suzuki, J., Isozaki, H.: Boosting-based parse reranking with subtree features. In: Proceedings of ACL'05, US (2005)
14. Toutanova, K., Markova, P., Manning, C.: The Leaf Path Projection View of Parse Trees: Exploring String Kernels for HPSG Parse Selection. In: Proceedings of EMNLP 2004, Barcelona, Spain (2004)
15. Kazama, J., Torisawa, K.: Speeding up Training with Tree Kernels for Node Relation Labeling. In: Proceedings of EMNLP 2005, Toronto, Canada (2005) 137–144
16. Shen, L., Sarkar, A., Joshi, A.k.: Using LTAG Based Features in Parse Reranking. In: EMNLP, Sapporo, Japan (2003)
17. Zhang, M., Zhang, J., Su, J.: Exploring Syntactic Features for Relation Extraction using a Convolution tree kernel. In: Proceedings of NAACL, New York City, USA (2006) 288–295

18. Zhang, D., Lee, W.: Question classification using support vector machines. In: Proceedings of SIGIR'03, Toronto, Canada, ACM (2003)