PhD Dissertation

International Doctorate School in Information and Communication Technologies

DISI - University of Trento

Structural Mapping between Natural Language Questions and SQL Queries

Alessandra Giordani

Advisor:

Prof. Alessandro Moschitti

Università degli Studi di Trento

October 2012

Abstract

A core problem in data mining is to retrieve data in an easy and human friendly way. Automatically translating natural language questions into SQL queries would allow for the design of effective and useful database systems from a user viewpoint.

In this thesis, we approach such problem by carrying out a mapping between natural language (NL) and SQL syntactic structures. The mapping is automatically derived by applying machine learning algorithms. In particular, we generate a dataset of pairs of NL questions and SQL queries represented by means of their syntactic trees automatically derived by their respective syntactic parsers. Then, we train a classifier for detecting correct and incorrect pairs of questions and queries using kernel methods along with Support Vector Machines. Experimental results on two different datasets show that our approach is viable to select the correct SQL query for a given natural language questions in two target domains.

Given that preliminary results were encouraging we implemented an SQL query generator that creates the set of candidate SQL queries which we rerank with a SVM-ranker based on tree kernels. In particular we exploit linguistic dependencies in the natural language question and the database 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 sub-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 ker-

nels to reorder the list of question and query pairs, where both members are again 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.

Keywords[Natural Language Interfaces, Database, Question Answering, Kernel Methods, Tree Kernels]

Acknowledgements

I would like to thank my advisor Prof. Alessandro Moschitti for the valuable perseverance and helpful patience shown over the years.

Special thanks go to Periklis Andritsos for providing the initial problem and discussing possible solutions at early stages.

Prof. Roberto Basili, Prof. Philipp Cimiano, Prof. Michael Minock and Prof. Martha Palmer for having demonstrated genuine interest in my research and for having devoted to it their time and long-standing experience accepting to be members of my Ph.D. defence board.

Bonaventura Coppola for having implicitly instigated me to take my time in writing this thesis, as long as I did finish it.

Rohit Kate, for his help in obtaining the GeoQueries and RestQueries datasets, and Ana Maria Popescu for providing us with the corresponding queries translated into SQL.

All the anonymous reviewers for having read my papers thoroughly and for having helped me improve them.

My family, for their constant support.

The most stubborn and proudest part of me, for having guided me into completing another important step of my life.

The work compiled in this thesis has been partially supported by the European Community's Seventh Framework Programme (FP7/2007-2013) under the grants #247758: ETERNALS – *Trustworthy Eternal Systems via Evolving Software, Data and Knowledge, and #288024:* LIMOSINE – *Linguistically Motivated Semantic aggregation engiNes.*

Contents

List of Tables

List of Figures

Chapter 1

Introduction

It would appear that we have reached the limits of what is possible to achieve with computer technology, although one should be careful with such statements, as they tend to sound pretty silly in five years.

John von Neumann, 1949

The design of models for automatically mapping natural language semantics into programming languages has been always a major and interesting challenge in Computer Science since if effective they would have a direct impact on industrial and social worlds. For example, accessing a database requires machine-readable instructions that common users are not supposed to know. Ideally, they should only pose a question in natural language without knowing either the underlying database schema or any complex machine language.

The development of Natural Language Interfaces to DataBases (NLIDBs) that translate the human intent into database instructions is indeed a classic problem, which is becoming of greater importance in today's world. In fact, despite the numerous attempts made in the past thirty years, current solutions are still not applicable in real scenarios. A huge variety of systems has been proposed, with increasing performance, but the task is still challenging.

The central issue in NLIDBs is understanding the user intent when he/she poses a question to a system, identifying concepts and relationships between constituents and resolving ambiguities. Systems that embed a relational database should perform a translation from a natural language question into an SQL query. Suppose that a user types the following natural language question: NL_1: "*Which states border Texas?*" The answer to this question can be found in the Geoquery database, a collection of data regarding US geography developed by Tang and Mooney [2001]. With respect to that database, a translation of question NL1 into a SQL query could be the following SQL_1:

```
SELECT border_info.border
FROM border_info
WHERE border_info.state_name = 'Texas';
```

This query is trivial enough to be understood even by a reader who does not have any knowledge about how the database has been constructed or how SQL are executed. However, without this knowledge it could be hardly written using the proper syntax and semantics.

This task becomes more difficult as soon as we ask more complex questions that require nesting queries and merging sparse data joining more tables. A slightly different but more articulated question similar to NL_1 is the following, NL_2: "*Which capitals are in states bordering Iowa?*". It can be answered executing query SQL_2:

```
SELECT state.capital
FROM state
WHERE state.state_name IN
  SELECT border_info.border
  FROM border_info
  WHERE border_info.state_name = 'Iowa';
```

This is evaluated starting from the inner query (retrieving "*states bordering Iowa*") and then projecting only the column `capital` of table `state`, such that the name of the state is in the retrieved set.

The same result could be obtained without nesting, joining the two tables as shown in query SQL'_2:

```
SELECT state.capital
FROM state JOIN border_info
ON state.state_name = border_info.state_name
WHERE border_info.border = 'Iowa';
```

This is not surprising since there can be multiple SQL queries that correctly retrieve the same answer to a NL question, as well as a question could be rephrased in different ways (e.g. "*Capitals of states that borders Iowa.*", "*What are the capitals next to Iowa?*"). Both natural and artificial languages can be very expressive and have complex structures and finding a mapping using a rule-based approach would be too demanding, requiring a lot of human intervention.

Instead, we rely on a machine learning algorithm to derive such mapping exploiting structured kernels. The idea of using such approach arose looking at the similarity between syntactic tree structures of NL questions and SQL queries.

With respect to the previous examples, Figure 1.1 represents the NL questions with syntactic propositional parse trees and SQL queries with relational parse trees. It shows that NL_2 shares some similarities with NL_1, which are reflected also by corresponding relational translated trees: SQL_2 embeds $VIEW_1$, whose structure is identical to the one of SQL_1. Dashed lines and triangles reflect the similarities between the structure of SQL_2 and the substructure $VIEW_1$ of SQL_1. Dotted lines indicate additional relationships between NL_2 and SQL_2.

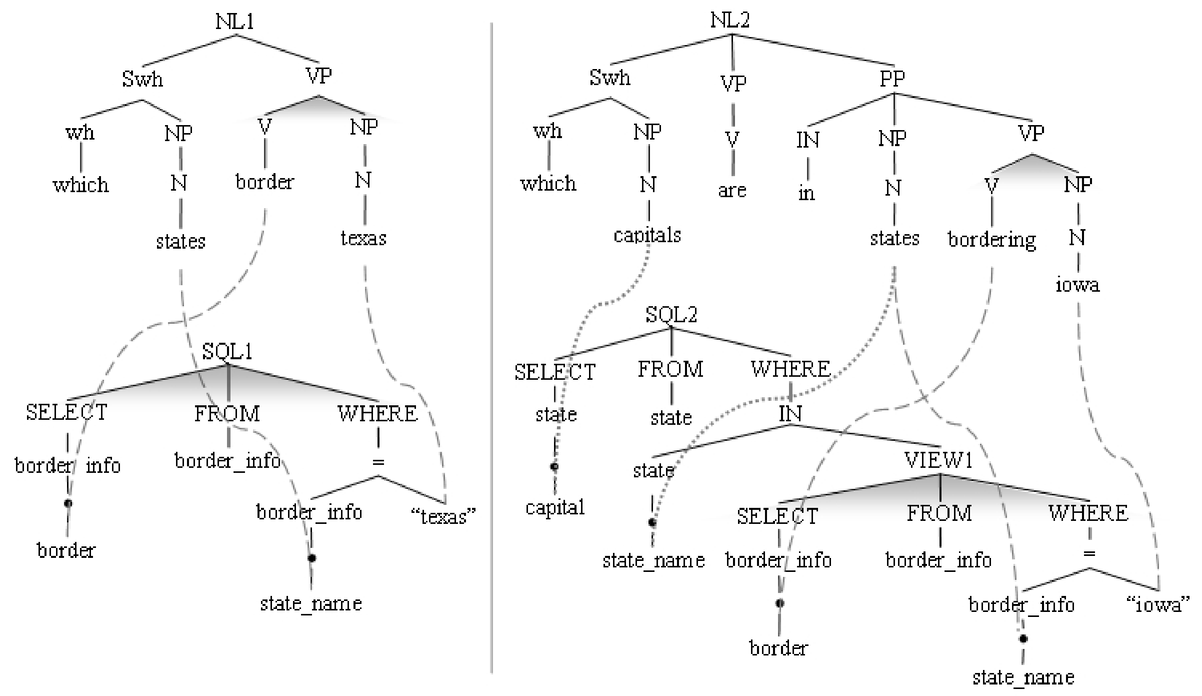


Figure 1.1: Propositional/relational pairs of trees and the relationships between their nodes

These parse trees can be derived from questions and queries using parsers as illustrated in Sections 2.1 and 2.2 respectively. Knowing this structure, the user would be able to straightforwardly substitute values of similar concepts (e.g. *Texas* VS *Iowa*) to create new SQL queries (see relational tree $VIEW_1$ in Figure 1.1). Generalizing similar concepts[1] can be easily performed exploiting the SQL constituents of the query.

From this pilot example, it is clear why we approach the problem of question answering as a matching problem. The main idea is deriving a shared semantics between natural language and programming language by automatically learning a model based the syntactical representation of the training examples.

Given pairs of trees, we can compute if and to what extent they are related using a tree kernel. The kernel represents tree-like structures in terms of their substructures and defines feature spaces by detecting common substructures. Exploiting Kernel Methods (Sections 2.3) we can operate in a Cartesian feature space without ever computing the coordinates of the data and relying on the cheaper and faster computation of the inner products between the images of all pairs of data in the feature space.

In Chapter 4 we illustrate how we can easily build a NLIDB implementing a machine learning algorithm to learn from an existing training set. First, starting from a set of correct pairs of questions and the related SQL queries, available for our target DBs, we need to produce incorrect pairs and additional correct pairs, i.e. negative and positive examples needed for training. This process requires that the pairs refer to general concepts in order to detect false negatives. We create clusters of pairs that represent the same information need and consider all pairing among the same clusters as positive examples and pairwise combinations between NL questions and SQL queries belonging to different clusters as negative examples.

[1]For example *Texas* and *Iowa* are istances of the general concept STATENAME.

Second, we propose a structural representation of the question and query pairs in terms of syntactic structures. We construct pairs of parse trees where the first tree is automatically derived by off-the-shelf natural language parsers whereas in order to build the second tree we implemented an ad-hoc parser that follows SQL grammar rules.

Third, we train Support Vector Machines with the above data, where the structural representation of the pairs is encoded by means of different types of kernels. Besides relying on linear and polynomial kernels, that computes the similarity of similar words, we exploit the potential of tree kernels and their combinations to compute the number of syntactic tree fragments. This allows us to automatically exploit the associative patterns between NL and SQL syntax to detect correct and incorrect pairs from an operational semantics viewpoint.

Finally, given a new question and the set of available queries (i.e. the repository of queries asked to the target DB), we produce the set of pairs containing such question and then we use the classifier to rank pairs in terms of *correctness*. We select the top scored pair to find the query that answer the given question.

Experimental results with this approach on two different datasets, GEOQUERIES and RESTQUERIES [Tang and Mooney, 2001] were encouraging, as described in Chapter 6. This preliminary experiment showed that (a) the approach is viable since we obtained a fairly high accuracy and (b) the advanced kernel combinations (i.e. product kernels) greatly improved basic models. The main contribution was demonstrating that we can exploit the space of feature pair, obtained with the product between kernels representing questions and queries, to express the relational features between their syntactic and semantic representations. However, this approach is not applicable to real world cases, since we can't always expect to have the set of all possible SQL queries that could be asked to a database.

In order to have a more useful and powerful approach, it should have been a generative one. It follows that, instead of learning a classifier, we have to generate plausible SQL queries and use a similar kernel-based re-ranker to select the one that best answers a given NL questions.

While in the first approach we needed at lot of positive pairings between NL questions and SQL queries to be given, in a generative approach we only need the answer (the result set or a SQL query that when executed retrieves it from the DB) for each NL question in the training set.

In Chapter 5 we illustrate how we can builds the sets of SELECT, FROM, WHERE clauses used to generate the set of possible SQL queries with respect to a given NL question based on its grammar dependencies. To prepare the reader to the core problem and proposed solution, we will first introduce the notion of typed dependencies and how to obtain a collapsed list of dependencies starting from an NL sentence. Then, we will describe 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. We also will introduce DB metadata, to show how executing SQL queries we can generate components of the final SQL query in a recursive fashion.

Combining all clauses in these three sets we obtain a huge set of SQL queries, possibly involving nesting and joining. However we also need to prune and weight queries from all possible combinations to generate an ordered set of meaningful queries among which we need find the answer.

Finally, the most probable answering query is retrieved using kernel-based re-rankers based on tree kernels. As already pointed out in our pilot leaning experiments, and confirmed by latest experimental results, discussed in section Chapter 7, thanks to our re-ranking models we show that product kernels are effective and efficient when we want to perform structural mapping between NL questions and SQL queries.

Our results compare favourably with state of the art systems. As illustrated in Chapter 3, the problem of automatically translating natural language (NL) questions into SQL queries is an interesting and appealing research in many research fields, i.e. data mining, information retrieval, artificial intelligence, etc.

In the last decade a variety of approaches have been proposed to translate the human intent into machine-readable instructions [Dale R., 2000; Tang and Mooney, 2001; Popescu et al., 2003; Ge and Mooney, 2005; Zettlemoyer and Collins, 2005; Kate and Mooney, 2006; Wong and Mooney, 2006; Minock et al., 2008]. Despite this, little progress has been made in developing a system that can be used by any untrained user without manual annotation and intervention.

A major challenge for mapping natural language to another language is recognizing syntactic variations of the same sentence meaning. For example, the question "*Which states border Texas?*" can be rephrased as "*Texas is next to which state?*". Previous work approach this problem with manually annotation of all these variations into the grammar, but it is tedious and error-prone. While in a restricted domain it is fairly easy to specify a grammar, in larger real-world applications it is not feasible due to the complexity and the subjectivity of all possible annotations. For this reason the ultimate goal of question answering is developing unsupervised methods that do not rely on annotated corpora.

In this research, we demonstrate that it is possible to fill the lexical gap between natural language expressions and database structure by relying on (i) the informative metadata embedded in all real databases, (ii) natural language processing methods, e.g., syntactic parsing, grammar dependencies, and (iii) advanced machine learning to build kernel-based rerankers.

Moreover we minimize the supervision effort required for learning by avoiding the annotation of the training set, exploiting the SQL queries paired with NL questions in the dataset only to obtain the real answers. Indeed we are not committed to any specific representation or language in our generative approach.

Similar semi-supervised machine learning approaches have been recently proposed [Clarke et al., 2010; Liang et al., 2011] but our system performs comparably given that our lever of supervision is much lower then the one requested by those systems to achieve slightly higher results.

We begin by reviewing the necessary background (Chapter 2) on natural language, databases and machine learning. In addition we give a survey of the state of the art in such fields (Chapter 3). We then describe our model for generating and classifying training pairs (Chapter 4), and then the reranking approach we use in addition to our SQL generation algorithm (Chapter 5). Finally, we present our experiments along with a comparison with related work (chapters 6 and 7). The results of this PhD thesis research (Chapter 8) confirm that this work provide some contribution to advance the field of Information Systems.

Chapter 2

Preliminaries

Natural language is what humans use for communication. It may be spoken, signed, or written but for our purposes we consider only written text. In the field of question answering this text is typically modelled as questions or sentences paired with the corresponding answer. That answer may be again a natural language sentence, a machine readable instruction in an artificial language, some structured data like tables, images and graphs or raw values (e.g. numbers of the result set of a database query).

In this research we consider only questions whose answers can be found in a database. Indeed we deal with pairs in two different languages, the human one and the Structured Query Language. In fact, to retrieve the final answer we need to execute a SQL query over a database to retrieve a result set.

In next chapters we will show how we address this question answering task by finding a mapping between natural language and the database programming language. Knowing how to convert natural language questions into their associated SQL queries, it is straightforward to obtain the answers by just executing a query. Unfortunately, previous work in Natural Language Understanding has shown the inadequacy of logic and rule-based approaches to this problem; in contrast shallow and statistical methods appear to be promising. Our solution indeed is based on robust machine learning algorithms, i.e. Support Vector Machines (SVMs), and effective approaches for structural representation, i.e. Sequence and Tree Kernels

This chapter introduces the reader to some basic knowledge in the fields of Natural Language, Databases and Kernel Methods while it gives a motivating example to clarify the overall process.

2.1 Natural Language Processing

Natural language understanding has been often referred to as an AI-complete problem because it requires extensive knowledge over multiple domains and the ability to manipulate it.

As previously stated, we do not perform natural language understanding but we apply shallow semantics models. However the natural language processing (NLP) problem of automatically extracting meaningful information from natural language input in order to produce a meaningful output (e.g. the correct answer to a given question) is still demanding. The problem is that natural language grammar is ambiguous and typical sentences have multiple possible interpretations. At the same time, the expressiveness of a language allows to have many semantically equivalent sentences, i.e. syntactically different questions that have the same semantics.

To cope with that we take into account only particular aspects of the natural language, considering only basic grammar relations holding between a subset of stems in a given NL question, relying only on syntax to derive the underlying semantics. This way we exploit fewer but enough information from the input NL question that is used by our mapping algorithm to find a mapping SQL query.

2.1.1 Stemming

Stemming is the process that reduces inflected and/or derived words to their root form. The obtained stem could not necessary be the word's morphological root, for example the words *city* and *cities* are associated with the stem *citi*.

In our framework we use the off-the-shelf stemmer written by Martin Porter as its stemming algorithm is widely used and has become the de-facto standard for English. We slightly modified his released and official free-software implementation[1] to accommodate the use of stems in order to better map those words whose stems ended with an *-y*. In these cases this letter just drops to allow the correct matching between, for example, *city* (that is now stemmed as *cit*) and *citizens*. This way we use stems to exploit conflation, a process also used by search engines to perform query broadening.

2.1.2 Parsing

The task of parsing deals with syntactically analyzing a string text to derive the grammatical structure that characterize it. The string of text should be first tokenized, using a lexical analyser to create tokens from the sequence of input characters. A grammatical analyzer then determines the relationship between the tokens (ie. words of the input string) and builds a data structure. The output of this process can be a parse tree (syntax tree) or other hierarchical structure.

In the following we introduce two kinds of off-the-shelf parsers that we embed in our framework to determine relationships between *stems* of input questions.

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

Charniak Parser

The aim of our research is to derive the shared shallow semantics within pairs by means of syntax. While early work on the use of syntax for text categorization used to be based on part-of-speech tags, e.g. Basili et al. [1999], the efficiency of modern syntactic parsers allows us to use the complete parse tree.

Thus we represent questions by means of their syntactic trees, obtained from the output of the open source[2] Charniak's syntactic parser [Charniak, 2000]. An example of such parse string for the sentence *Which capitals are in states bordering Iowa?* is the following.

```
(ROOT
  (SBARQ
    (WHNP (WDT which) (NNS capitals))
      (VP (VBP are)
        (PP (IN in)
        (NP (NNS states))
        (VP (VBG bordering)
        (NP (NN iowa)))))))
```

The corresponding visual representation of its parse tree is shown in Figure 2.1.

2.1.3 Stanford Dependency Parser

In addition to constituency parsing we exploit dependency parsing. While the first is used for representing questions by means of constituencies in a hierarchical way, the second one is useful to directly capture binary dependency relations holding between the words in a question.

[2]ftp://ftp.cs.brown.edu/pub/nlparser/

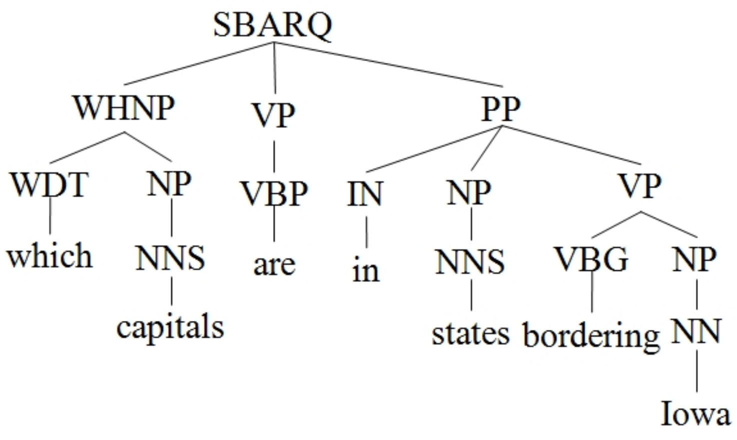


Figure 2.1: Syntactic tree of a NL question.

We use the Stanford Dependencies representation [Marie-Catherine de Marneffe and Manning, 2006] since it provides a simple and uniform description of binary grammar relations 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.

For this reason, as we will see in depth in section 5.1.2, we'll modify the list given as output from Stanford Parser removing useless relations (e.g. determinant, det(states-13, the-10) and reduce each *governor* and *dependent* to their stemmed form.

2.2 Databases

A database consists in a collection of data organized in possibly very large data structures.

It is worth noting that the term database does not apply directly to a database management system (DBMS). The combination of database (data collection and data structures) and the underlying DBMS is referred to as *database system*.

A large variety of database systems have been used and developed as soon as they started to be popular. In the past twenty years, relational systems have shown to be the most dominant, while the most spread database language is the standard SQL for the Relational model.

For this reason in the research we consider and describe only a relational system. We first introduce how data is organized according to the relational model, then we discuss how metadata is stored in the underlying database catalog. Last, we describe in detail the most common language associated with the relational model (SQL) and which subset of its grammar we can deal with in our framework. Following such grammar rules we developed a parser that builds a relational tree for a given SQL query.

2.2.1 Relational Databases

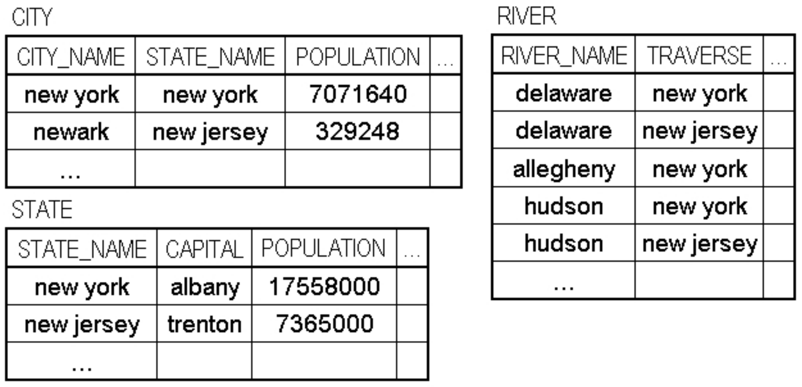
According to the relational model, data is organized into two-dimensional arrays called relations (database tables). Each relation has a heading and a set of tuples. The heading is an unordered possibly empty set of attributes (table's columns). Each tuple is a set of unique attributes and values (table's rows). Data across multiple tables is linked with a key, that is, the common attribute(s) between tables. The dominant language associated with the relational database is the Structured Query Language (SQL).

In this research we experiment with different relational databases, about:

- Geographic information regarding cities, rivers, mountains and lakes in US states and their surrounding states. It is called Geobase and is part of the GeoQuery corpus of R.Mooney and his group. This corpus is available in different sizes and languages.

- Restaurants information including names, addresses, ratings of Californian restaurants. This corpus has been also provided by Mooney's group and is part of RestaurantQuery dataset.

Figure 2.2 shows a fragment of the GeoQuery database. In particular this piece of information, distributed across the tables `RIVER`, `STATE` and


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	
...		

Figure 2.2: GeoQuery database fragment

`CITY` embeds the intrinsic information that `NEW YORK` is a *state name*, its *capital* is `ALBANY` and its *population* is around 17 million people. However `NEW YORK` appear to be also a *city name* in the homonymous *state* with a *population* of 7 million people. Last, `NEW YORK` shows up also on table *RIVER* under the field *traverse.* Whether this information regards states or cities is clear to the reader but from the DBMS point of view this relation should be stored as metadata in a specific data structure: the database catalog (called Information Schema in SQL systems).

2.2.2 Metadata and Information Schema

The DBMS manage the database(s) that resides in it by means of a storage engine. It stores all the information about the data (metadata) into internal data structures.

In a relational-database metadata is stored into tables. Some examples are shown in Figure 2.3 and are basically the following:

- A table containing all tables' names for every database, along with their size and number of rows.
- A table storing column names in each database, together with the information about which tables and database they are used in and the type of data they store.
- A table that keeps track of referred tables and columns by means of external keys.
- A table used to maintain database constraints to ensure database integrity.

In database terminology, this set of metadata is referred to as the catalog. In standard SQL the catalog can be accessed interacting with an internal database, called the Information Schema (in the reminder we refer to it as IS).

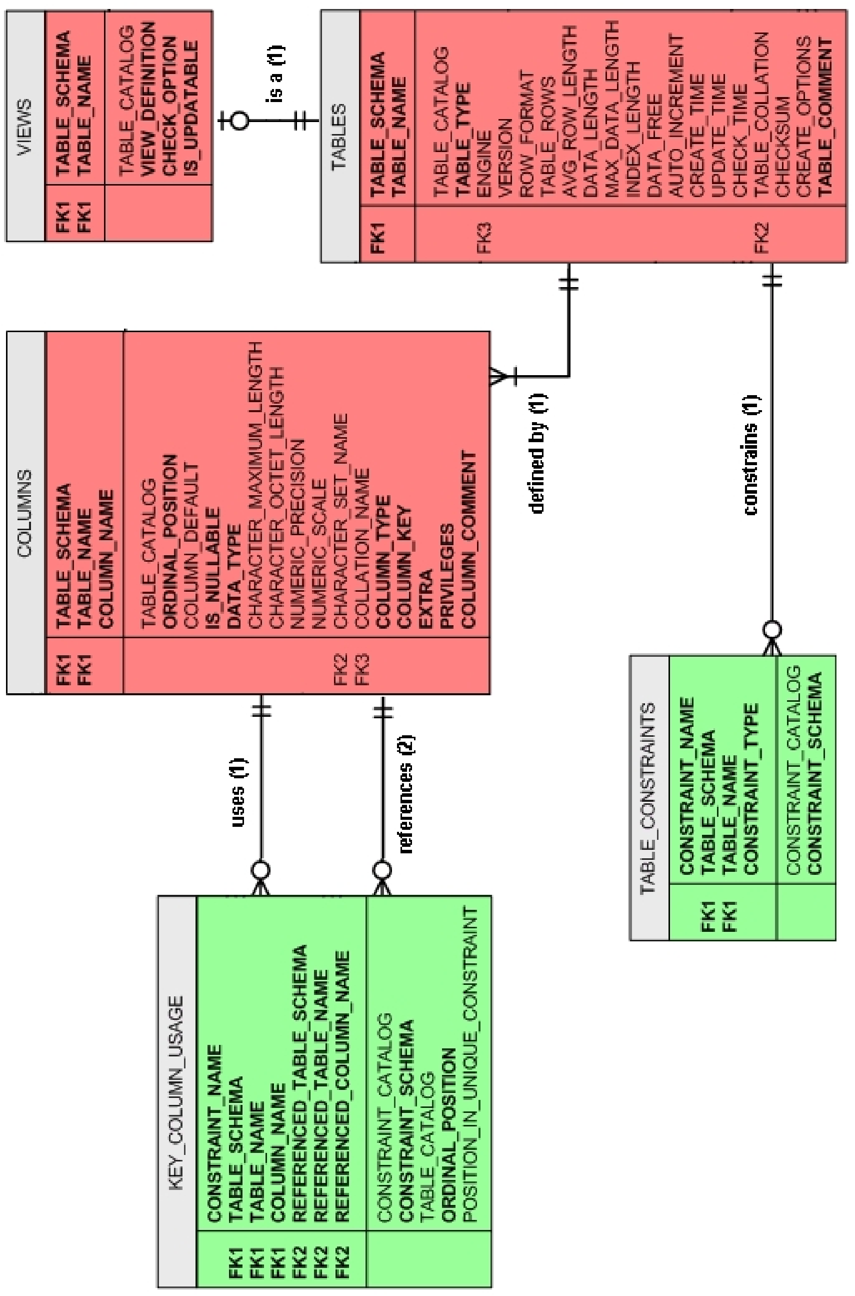
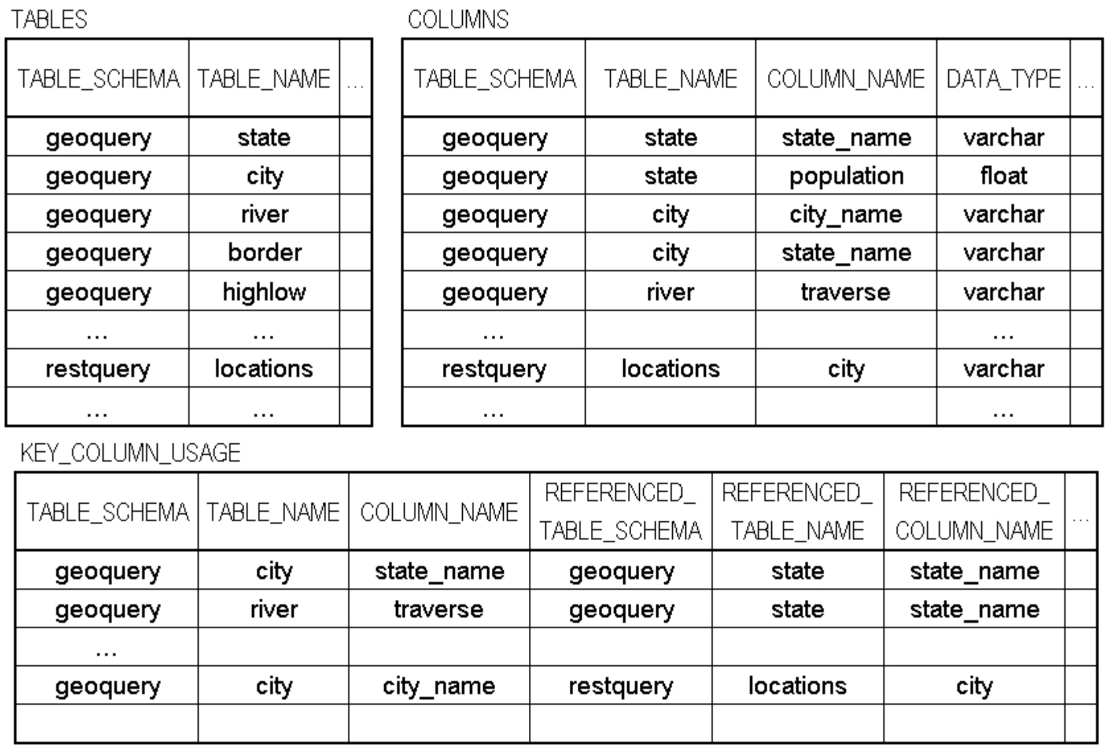


Figure 2.3: Conceptual Database Model diagram for the MySQL IS database



TABLES

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

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	
...			...	
restquery	locations	city	varchar	
...			...	

KEY_COLUMN_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	
...						
geoquery	city	city_name	restquery	locations	city	

Figure 2.4: IS fragment a SQL DBMS containing GeoQuery and RestaurantQuery

To show an example of what kind of metadata is typically stored into IS, lets look at Figure 2.4. It contains table names and column names of the GeoQuery and RestaurantQuery databases, but as actual values, not as table or column names (see Figure 2.2 for a comparison).

With respect to the previous example (introduced discussing Figure 2.2), the fact that `NEW YORK` shows up in table *RIVER* under the field *traverse* can be used to disambiguate the term `NEW YORK` as being referred to a *state name*.

2.2.3 SQL queries and Relational Algebra

SQL was one of the first languages developed for the relational model. Nowadays is the most widely used database language, supported by all the relational DBMSs.

The basic components of statements and queries are called clauses. The SELECT components is the most important: it indicates which columns should be kept in the final result (an asterisk specify that all columns in the tables should be kept). Tables that contain this data are indicated in the FROM clause (it can include optional JOIN sub-clause to specify the rule for joining tables). The WHERE clause is optional, and consist of a comparison predicate. In order to restricts the rows in the final result set, it eliminates all rows from the result set for which the comparison predicate doesn't hold. Other possible clauses, that we don't take into account are GROUP BY, HAVING and ORDER BY.

Hence, the general SQL query that our system can deal with has the form:

$$\text{SELECT } COLUMN \text{ FROM } TABLE \text{ [WHERE } CONDITION\text{]} \tag{2.1}$$

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:

$$\pi_{COLUMN}\left(\sigma_{CONDITION}(TABLE)\right) \tag{2.2}$$

We introduce this formalism to be more concise in all the definitions that follow in the remainder of the thesis.

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 [Garcia-Molina et al., 2008]. For our purposes the fact of having duplicates in the result adds noise; 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)`

In other words, every expression can produce scalar values or tables consisting of columns and rows of data. The result of a nested query (called subquery) can be used in another query via a relational operator or aggregation function.

An example of a complex WHERE condition could be the following one, referring to "*major non-capital cities excluding texas*":

```
WHERE city.population > 15000 AND
  (city.city_name NOT IN
    (SELECT state.capital FROM state)) AND
  NOT city.state_name LIKE "Texas"
```

The meaning of $TABLE$ is more straightforward, since it should just contain table name(s) to which the other two clauses refer. This clause could just be a single relation or a join operation, which combines records from two or more tables by using values common to each. We only deal with theta-joins where we take the Cartesian product of two relations and exclusively select only 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*").

2.2.4 SQL Parsing

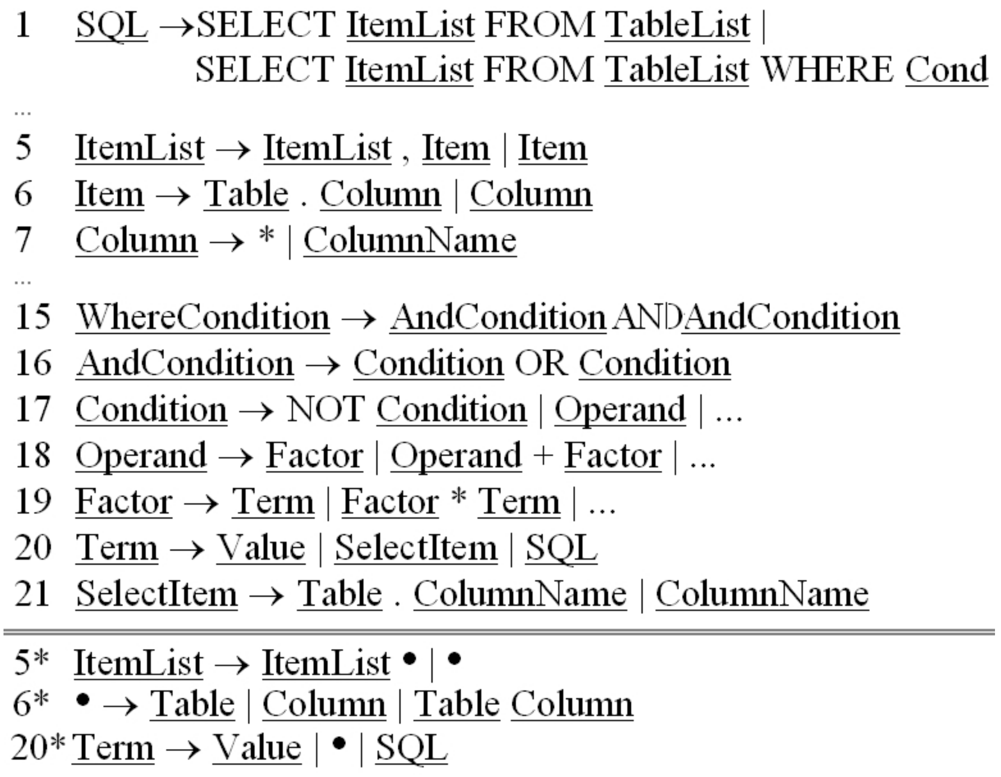
Since we aim at transforming NL trees into SQL trees, we need to represent also the SQL as a syntactic tree. To build the SQL tree we implemented an ad-hoc parser that follows the syntactic derivation of a query according to our grammar. Since our database system embeds a MySQL server, we use the production rules of MySQL, shown at the top of Figure 2.5, 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 of the relationship between a table and its column in both the SELECT and WHERE clauses. 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. 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.



```
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
```

Figure 2.5: Modified MySQL Grammar

2.3 SVMs and Kernel Methods

Kernel methods (KMs) define a class of learning algorithms used for pattern analysis typically associated with support vector machines (SVMs). The problem in pattern analysis is finding any kind of relations (e.g. rankings, correlations, classifications, etc.) in any types of digital data (e.g. text sequences, sets of points, vectors, trees, images, etc.).

KMs approach this problem by mapping the data into a high dimensional feature space, where each coordinate corresponds to one feature of the data items. This data is transformed into a set of points in a Euclidean space where a variety of methods can be used to find relations among them, and indeed, holding also in the original data.

KMs make extensively use of kernel functions, which are able to operate in the feature space without ever computing the coordinates of the data in that space, but just computing the inner products between the images of all pairs of data in the feature space. This operation is often computationally cheaper than the explicit computation of the coordinates.

The main idea is that the parameter model vector $\vec{w}$ generated by SVMs (or by other kernel-based machines) can be rewritten as

$$\sum_{i=1..l} y_i \alpha_i \vec{x}_i \tag{2.3}$$

where y_i is equal to 1 for positive and -1 for negative examples, $\alpha_i \in \Re$ with $\alpha_i \geq 0$, $\forall i \in \{1,..,l\}$ $\vec{x}_i$ are the training instances. Therefore we can express the classification function as

$$\text{Sgn}(\sum_{i=1..l} y_i \alpha_i \vec{x}_i \cdot \vec{x} + b) = \text{Sgn}(\sum_{i=1..l} y_i \alpha_i \phi(o_i) \cdot \phi(o) + b) \tag{2.4}$$

where $\vec{x}$ is a classifying object, b is a threshold and the product

$$K(o_i, o) = \langle \phi(o_i) \cdot \phi(o) \rangle \tag{2.5}$$

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 [Shawe-Taylor and Cristianini, 2004], 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 following section, we first propose a structural representation of the question and query pairs, then we report the two more adequate kernels for syntactic structure representation, i.e. the Syntactic Tree Kernel (STK) [Collins and Duffy, 2002], which computes the number of syntactic tree fragments and the Extended Syntactic Tree Kernel (STK$_e$) [Zhang and Lee, 2003b], which includes leaves in STK. In the last subsection we show how to engineer new kernels from them.

2.3.1 Tree Kernels

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

$$TK(T_1, T_2) = \sum_{n_1 \in N_{T_1}} \sum_{n_2 \in N_{T_2}} \Delta(n_1, n_2) \tag{2.6}$$

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).

Syntactic Tree Kernel (STK)

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 [Collins and Duffy, 2002]:

1. if the productions [3] 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 the node n and λ is a decay factor penalizing larger structures.

Figure 2.7.a shows some STFs of the NL and SQL trees in Figure 2.6. STFs satisfy the constraint that grammatical rules cannot be broken. For example, in this figure, *[VP [AUX NP]]* is a STF, which has two non-terminal symbols, *AUX* and *NP*, as leaves whereas *[VP [AUX]]* is not a STF.

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

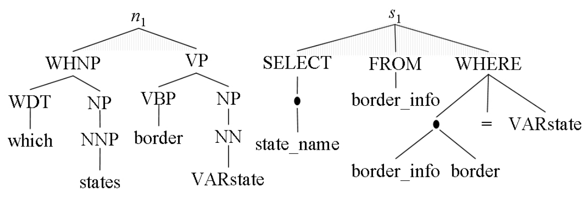


Figure 2.6: Question/Query Syntactic trees

Syntactic Tree Kernel Extension (STKe)

STK does not include individual nodes as features. As shown in Zhang and Lee [2003b] using its extension (STK$_e$) 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 [Zhang and Lee, 2003b]:

0. if n_1 and n_2 are leaf nodes and their labels are identical then $\Delta(n_1, n_2) = \lambda$;

2.3.2 Relational Kernels

Since our goal is finding common substructures shared by pairs of trees, as shown in Figure 2.6, we need to represent the members of a pair and their interdependencies. Then we can carry out kernel engineering by combining basic kernels with additive or multiplicative operators.

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

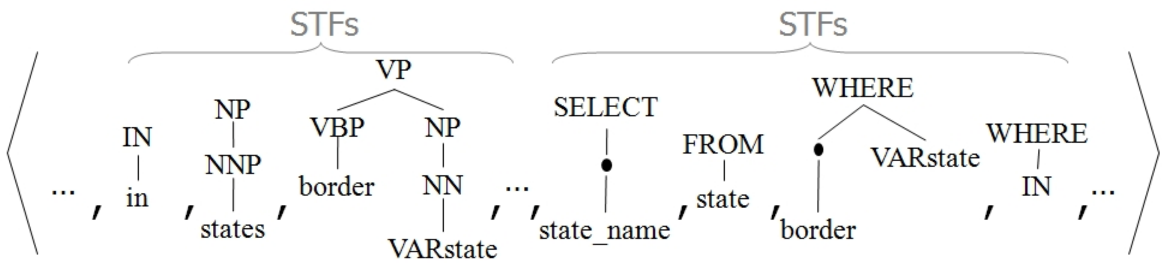


Figure 2.7: Joint space STK+STK for the tree pair in Figure 2.6

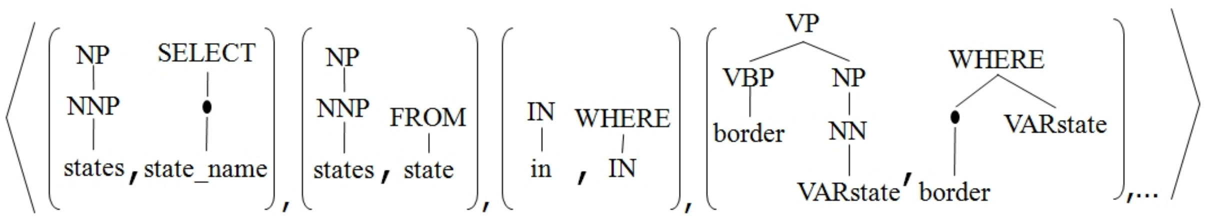


Figure 2.8: Cartesian product STK×STK feature spaces for the tree pair in Figure 2.6

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 space of STK of the pair in Figure 2.6 is shown in Figure 2.7. The Syntactic Tree Fragments of the question will be in the same space of the Syntactic Tree Fragments of the query.

Product Kernel

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 2.8 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`. Similarly the last fragment pair suggests that the verb phrase *border VARstate* semantically maps the syntactic construct `WHERE border = VARstate`. In other words, from the above feature pairs we can derive that the whole query may be a correct translation of the given question.

Polynomial Kernel

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 the results using different kernel combinations for pair representation.

2.4 Motivating Example

In order to illustrate our goal and proposed ideas to the reader, we give a short example and briefly discuss the overall process.

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 we've shown that the database schema also specifies constraints and data types.

IS can be inspected as a normal database, posing SQL queries to obtain useful fields to build a new SQL query. In practice, we can use the same technique and technology to generate an answer to a given question and retrieve the answer itself.

For example, an answer for the question "*Which rivers run through states bordering New York*" can be found in the GeoQueries corpus. This is associated with a spatial database whose structure is stored in IS as shown in Figure 2.4 (see page 20).

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 term *run* can be easily performed by looking at the less semantically distant metadata entry, i.e., *traverse*. This matching is furthermore 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, matching 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 2.2, but the key idea is to exploit related metadata information (i.e. primary and foreign keys, constraints, datatypes, etc.) to select the most plausible one.

After some generative steps we might end with a set of possible SQL queries including the following ones:

```
SELECT river.river_name
FROM river JOIN border_info ON river.traverse = border_info.border
WHERE border_info.state_name = 'New York';
```

The above query correctly answers to "*Which rivers run through states bordering New York*". However other similar mappings can be investigated:

```
SELECT river.river_name
FROM river JOIN city ON river.traverse = city.state_name
WHERE city.city_name = 'New York';
```

This query retrieves "*Names of rivers that run through the state of New York city*" mismatching the *bordering* information need. Another possible mapping query can be:

```
SELECT river.river_name
FROM river
WHERE river.traverse = 'New York';
```

Its result, for a mere coincidence, is equivalent to the previous one, so still mismatching. While these last two queries are valid queries and share some common stems with the correct one, we note that the first one maximizes this similarity. For this reason we introduce a method to weight generated queries according to the common stems.

However we can't just rely only on bag-of-stems to compute similarity. There can exist some special word or phrase in the NL question that maps to a whole SQL sub-structure.

For example, the word major, referred to cities or rivers, have different implicit translation in SQL. Let us analyze the mapping between "*Major cities in USA*" and "*Major rivers in US*" and their associated queries. With respect to GeoQueries dataset, matching queries are the following ones:

```
SELECT city.city_name
FROM city
WHERE city.population > 150000;
```

```
SELECT river.river_name
FROM river
WHERE river.length > 750;
```

If we consider just stems, we can't infer any mapping to translate *major* into any of its meaning. But if we take into account their syntactic trees we can investigate the matching between tree structures using structural kernels.

In particular, exploiting the product kernel between their trees, we generate a feature space that embeds pairing between substructures, e.g. the first fragment pair of the Figure 2.9 suggests that the noun phrase *major cities* expresses similar semantics of the syntactic construct `WHERE city.population`>`150000`. At the same time, *major rivers* is semantically similar to the syntactic construct `WHERE river.length`>`750`, with a similarity degree higher than the one with `WHERE city.population`>`150000`.

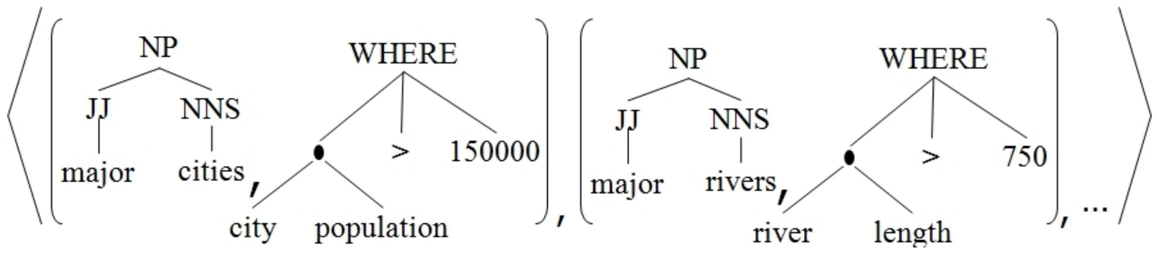


Figure 2.9: Sample feature space of the cartesian product between synctactic tree kernels.

Chapter 3

State of the art

The numerous attempts to design models for automatically answering natural language questions made in the past thirty years have shown that the task is much more difficult than was originally expected [Copestake and Sparck-Jones, 1990]. A huge variety of systems has been proposed, with increasing performances, but the task is still challenging.

While early systems were built to translate questions in Prolog language, in the last twenty years relational databases became very popular making Structured Query Language (SQL) the most widespread database language [1].

This translation usually takes place in three stages: a first step produces a tree-like structure from the natural language input; a second one transforms the tree into an internal logical representation, which is finally translated into a statement in the target artificial language. The systems reviewed in this section can be classified into three categories: spoken language understanding, keyword-based searching and question answering in NLI (i.e. user interfaces that handle natural language sentence in text format).

[1] In this chapter we will give an overview of most recent systems; for a detailed review of systems developed before the nineties, please refer to [Chandra and Mihalcea, 2006; Dale R., 2000].

3.1 Spoken Language Understanding

In 1990, the Advanced Research Project Agency defined an application domain, called ATIS (Air Travel Information Service), for collaboration and comparison of the result in the area of speech language understanding. The following paragraph illustrates the techniques used by some spoken language systems that addressed this challenge, as discussed in Pallett et al. [1994].

Phoenix uses a bottom-up semantic parser to build trees from the sentences and then extract information into slots of frames. These frames, which describe semantic entities (e.g. flight, airport, time and other concepts), are then used to produce the corresponding SQL queries. Another system that uses this approach is SRI, which combines a syntactic parser with a template matcher. This matcher fills a set of semantic structures similar to frames using some domain keywords. Chronus, by AT&T, embeds a conceptual statistical decoder that uses Hidden Markov Models to decode from an acoustic signal a list of concepts in order to build a bigram conceptual model. Also the BBN system represents concepts and sub-concepts with a tree structure, distinguishing between terminal and non terminal nodes to define a statistical model. The Chanel system uses instead a set of trees (called Semantic Classification Trees) to represent semantic rules. Such trees contain the list of all possible attributes of the SELECT statement and WHERE constraints.

Another speech understanding system based on statistical representation of semantics was developed in 1992 [Pieraccini et al., 1992]. Authors assumed a limited domain where the number of concepts - used to disambiguate the sentence - was finite. The system was tested in the ATIS domain and behaved well, answering most of the input sentences, but the use of a restricted vocabulary was too restrictive.

In 1993, an evaluation of these systems [Pallett et al., 1994] in the ATIS domain revealed that the ones performing better were Phoenix and BBN. The key feature of these approaches was the use of trees to represent the syntax of the sentences given in input and then identify semantic relations among leafs. However, this research program focused on handling continuous and spontaneous speech recognition, rather than on natural language processing.

However, with the advent of the Web and search engines, people got used to express their information need in terms of keyword combinations in place of typing whole sentences. For this reason, the focus of natural language understanding shifted from speech recognition to keyword-based searching.

3.2 Keyword Search

In the past decade, question answering has been also addressed by keyword-based search. It retrieves requested information both from documents and from databases but with different techniques: while in the first case the keywords are to be searched on a unique source, in the second case the information is split across multiple tables due to normalization.

In order to understand to what extent our system differ from previous approaches, we will now review some state-of-the-art systems that perform keyword-based search only on top of relational databases.

Among recent systems that enable keyword-based search we find Microsoft DBXplorer [Chaudhuri et al., 2002]. It uses a symbol table to store tables, columns, and rows of all data values, which is looked up during the search to identify the locations that contain all the keywords appearing in the question.

This table is very useful to disambiguate terms, but although compression methods to store the symbol table in the database are available, space and time requirements remain a critical factor. Anyway there is no need to maintain a symbol table if we can rely on the actual database and its underlying metadata.

The BANKS [Hulgeri et al., 2001] and ObjectRank [Balmin et al., 2004] systems apply ranking to keyword search over databases: results are ranked with respect to their relevance, computed using an approach similar to PageRank in BANKS while ObjectRank applies authority-based ranking. One beneficial feature of BANKS is that it also takes into account metadata while performing the search. Both these systems use graphs to model relational databases, where each node represents an object of the database. The ranked answer is a sub-graph where weighted nodes are ordered based on descending relevance. The same method is used in our approach to rank results in a way that reflects the correctness of the answers generated by the system with respect to the question, i.e. based on the number of matching substructures between the relational tree and the propositional tree.

Précis [Koutrika et al., 2006] is another system that uses both an inverted index and a directed schema graph to generate a new database and a personalized natural language synthesis of result. Besides, keyword question answering is implemented using huge inverted indexes and symbol tables that need to be rebuilt whenever the database is updated. Indeed, this approach is not suitable for very large databases where the high number of tables and rows would prohibit symbol table maintenance. In addition, it's worth noting that these systems don't consider solutions that include two tuples from the same relation. That is, they retrieve a single-value answer, while the solution is often a set of values or strings.

3.3 Question Answering in NLI

The ultimate way to handle the problem and allow to query very large data is the implementation of natural language interfaces (NLI). In particular, natural language interfaces to database (NLIDBs) take natural language questions (whole sentences as well as keywords) and translate the user intent into machine-readable instructions to retrieve the answers.

One of the first attempts made towards this goal was made by Hendrix [1977] with LIFER: A Natural Language Interface Facility. A parser contained within LIFER was able to translate sentences and requests into appropriate interactions with a database system. An interesting feature of LIFER was that the language it could handle was defined in terms of patterns, which used semantic concepts in the domain of application. LIFER used a simplified augmented transition network to analyse thee input sentence. Each pattern defined by the grammar corresponded to a possible path in the transition network.

LIFER was used at SRI as the natural language component of a system called "LADDER" for accessing multiple, distributed databases. In 1982, Hendrix left SRI to form Symantec and introduced its first commercial product, Q&A [Hendrix, 1986] that included an *intelligent assistant* to let users manipulate databases and produce reports by issuing commands or asking questions in English. Assuming a very simple database organization, English queries were translated into a hypothetical database query and then transformed into a series of actual database queries that took into account the actual organization of the database.

CHAT-80 [Waltz, 1978] is the best-known NLI of the early eighties. Entirely implemented in Prolog, it was able to answer rather complex questions, posed in English, about a database of geographical facts. The computation of the meaning (that is, the semantics) of an English query was guided by the syntactic structure of the query and was expressed as a logical formula. This formula was then transformed into the individual queries of the database needed to answer the original question. The code of CHAT-80 had widely circulated, giving the basis of several other experimental NLIs (e.g. MASQUE, i.e. Modular Answering System for Queries)

MASQUE/SQL [Androutsopoulos et al., 1993] is a modified version of the MASQUE system. It answers written English questions by generating SQL codes, while the original version generated Prolog queries. English questions are first transformed into expressions in a logical query language and then translated into SQL. The entity types of the world to which the database refers are organized in a hierarchical forest and the meaning of words is expressed as a logical predicate. As opposite to other inefficient approaches, where partial results are joined by the front-end, it uses DBMS specialized optimization techniques. It embeds a built-in domain editor that should be used by an expert to configure the system for new knowledge domains, to declare expected words and to define the meaning in terms of a logic predicate. Another disappointing aspect of this application is that if the SQL query fails, the system does not know which part of the query caused the failure.

The open-source natural language application SQ-HAL [Ruwanpura] needs to know the relationships between tables to create the SQL query. Besides having to manually enter these relationships, users are requested to suggest synonyms. An approach that requires user assistance defeats the original goal of building an NLIDB for naïve users. Let us recall that, as far as we use relational databases, relationships between tables can be automatically derived.

EASYASK[2], also known as English Wizard, is a commercial applica-

[2]http://www.easyask.com

tion that offers both keyword and natural language search over relational databases. The system crawls the data to automatically construct a contextual dictionary used to identify words that correspond to values or catalog attributes and generate an SQL statement. It incorporates approximate word matching, stemming and synonyms.

EQ, which stands for English Query [R., 2004], is a NLIDB implemented by Microsoft Corporation, as a part of the SQL Server. It creates a model, collecting database objects (tables, fields, joins) and semantic objects (entities, additional dictionary entries, etc). However, it only extracts few basic relationships and, thus, requires refining the model manually.

ELF, English Language Front End [3], is another commercial system that generates a natural language processing system for a database. This interface works with Microsoft Access and Visual Basic. The parser builds the parse tree from the input, by swapping and substituting the words with the SQL keywords wanted in the final result. The basic model is built automatically, extracting most of the relationships from the database. A comparison between these two systems [R., 2004] illustrated the superiority of the ELF system over EQ. The reason is that EQ, like many other natural language systems except ELF, relies on context-free grammars.

A system that, instead, enables casual user to query XML database, without any knowledge of the underlying schema, is NaLIX [Li et al., 2005]. It allows to query XML databases relying on schema-free xQuery and asking the users to rephrase those questions not understood by the system, interacting with a feedback module. While no domain-specific knowledge is used, they provide ontology-based term expansion controlled by the user when term disambiguation is needed. The approach of mapping grammatical proximity of tokens in the parsed tree into proximity of elements in the final xQuery statement is, to a certain extent, similar to

[3]http://www.elfsoft.com

our technique.

Many multilingual interfaces have also been proposed and are typically more complex that those that deal only with English. For example the tourism NLIDB Tiscover [Dittenbach M. and Berger, 2003], takes as input both German and English and the language is automatically detected using an n-gram-based text classification approach. To compensate for orthographic errors, a spell checker is used and word occurrence statistics from previous queries are taken. It uses language-dependent ontologies to identify relevant parts of the question and a domain-specific processing logic to define how these relevant parts are related to each other to build the appropriate database query. For each language they create a dictionary with concepts and synonyms, prepositions, adverbial and adjectival structures, proper names (e.g. names of cities and states). This means that all domain-dependent words have to be stored for each language, increasing the size of the domain dictionary and, thus, decreasing the portability.

Edite [Filipe and Mamede, 2000] is another NLIDB that answers written Portuguese, French, English, and Spanish questions about tourism resources by transforming them into SQL queries. However this translation is inefficient, being based on several mapping tables which, as said before, are highly dependent on database organization, domain and language. Translation is thus only possible in restricted domains with few proper names. The only beneficial feature of this system is the complete separation between the linguistic component and the database knowledge, which leads to guaranteed portability.

An NLIDB that accepts only Spanish natural language questions is presented in Barbosa et al. [2006]. This interface relies on synonyms and domain dictionaries, as well as on metadata dictionary, analyzing nouns, prepositions and conjunctions present in the queries issued to the database (similarly to our approach for exploiting grammar relationships of the given question). It is domain independent, which means that the interface can be used with different databases, but this requirement complicates the task of achieving high translation success. Moreover the experimental evaluation of this system its not reliable, since when formulating their questions, the users were given the database schema and information contained in the database, while this is never the case when a NLIDB is used.

Authoring systems (e.g. CatchPhrase [Minock et al., 2008]) rely on semantic grammar specified by an expert user (i.e. the author) to interpret question over a database. The author has to name database elements, tailor entries and define additional concepts. Up to the 1980s, most NLP systems were based on complex sets of hand-written rules. Starting in the late 1980s, however, there was a revolution in NLP with the introduction of machine learning algorithms for language processing.

To analyse the reliability of an NLI a theoretical framework has been proposed in Popescu et al. [2003]. Authors give the formal definitions of soundness and completeness and identify a class of semantically tractable natural language questions for which sound and complete NLIs can be built. They claim "if the sentence tokenization contains only distinct tokens and at least one of its value tokens matches a *wh*-value, the corresponding sentence is semantically tractable". Based on this theoretical analysis they have been able to build an NLI, called Precise, that is 100% accurate: it correctly answers all semantically tractable questions, otherwise requests rephrasing. They reduce the problem of finding a semantic interpretation of ambiguous natural language tokens as database elements to a graph matching problem. Then they find valid mapping(s) from a complete tokenization of a given question to a set of database elements that are finally converted into a SQL query (queries). Although this can be easily solved using the maxflow algorithm, computing the set of complete tokenization is equivalent to the NP-hard problem of exact set covering.

While previous work [Ge and Mooney, 2005; Zettlemoyer and Collins, 2005; Wong and Mooney, 2006] has explored how to learn a mapping from sentences to lambda-calculus meaning representations requiring supervision such as manual labelling, the best performing state-of-the-art systems adopt instead machine learning approaches to induce a semantic grammar from data consisting of sentences paired with their meaning representations.

For example, KRISP [Kate and Mooney, 2006] takes pairs of sentences and their computer-executable meaning representations as training input to find a mapping between sentences and Prolog assertions using an SVM classifier. For each production in the meaning representation language the model is built on top of SVM with string kernels. Then the classification is used to compositionally represent a natural language sentence in its meaning representation. While this approach performs well, it is constrained by learning a grammar that contains a fixed set of lexical items to model the meanings of input words and production rules to combine them in order to analyze the sentence meanings. In other words, it may fail to produce an answer to an unseen question since the grammar does not include the rules required to correctly parse it.

Another effective and efficient model that transforms sentences into hierarchical representations of their underlying meaning has been presented in Lu et al. [2008] (we refer to it as MODELIII+R). It builds a single hybrid tree of words, syntax and meaning representation by recursively creating nodes at each level according to a Markov process. Then the averaged perceptron algorithm is applied for discriminative reranking. Rather than relying on full syntactic annotations, it reduces the problem of mapping sentences to logical form to a learning algorithm that creates accurate structured classifiers in a way that requires little human assistance for new unseen domains. This is a generic model that does not require any domain dependent knowledge and could be applicable to a wide range of different domains.

There exist other unsupervised semantic parsers, for example the USP system [Poon and Domingos, 2009], that tackle the problem of learning a semantic parser using Markov logic. The system transforms dependency trees into semi-logical forms, inducing lambda forms and clustering syntactic variations of the same meaning. However it is not able to handle deeper linguistic phenomena such as quantification, negation, and superlatives. Moreover the scope of this system is to perform question answering over text, while in this research we focus on question answering over databases.

Another system, called UBL [Kwiatkowski et al., 2010] induces a probabilistic CCG grammar to represent the meaning of individual words and define how to combine meanings in order to represent complete sentences. Starting from a restricted set of lexical items, using higher order unification along with the CCG combinatory rules they learn new lexical entries, defining the space of possible grammars in a language- and representation-independent manner.

Recent work focusing on compositional semantics [Liang et al., 2011] and semantic parsing [Clarke et al., 2010] avoid the need for annotation by considering the end-to-end problem of mapping questions to answers. In Liang et al. [2011] authors introduced a new semantic representation (DCS, dependency-based compositional semantics) to highlight the parallel between syntactical dependencies and evaluation of latent logical forms. They give semantically-scoped denotations to syntactically-scoped trees. In addition they extended the model with an augmented lexicon to handle prototype words.

Our idea closely follows the unsupervised learning approach described in Clarke et al. [2010]. They generalize questions that describe the same information need and pair them directly to the final answer, instead of relying on meaning representations and their annotation. The SemResp approach uses the syntactic information to build a semantic interpretation exploiting a binary classification problem to implement a feedback module.

Additionally we used tree kernels. With respect to the other natural language tasks that employ tree kernels, in the literature [Collins and Duffy, 2002; Kudo and Matsumoto, 2003; Cumby and Roth, 2003; Culotta and Sorensen, 2004; Kudo et al., 2005; Toutanova et al., 2004; Kazama and Torisawa, 2005; Shen et al., 2003; Zhang et al., 2006; Zhang and Lee, 2003a], several models have been proposed and experimented.

The following table summarizes the features of the above mentioned question answering systems, highlighting the advantages and disadvantages of such approaches. A system is domain independent (Dom. I.) if it does not need to be re-configured when the domain changes while it is database independent (DB I.)if modifying the database it is not necessary to re-implement anything. Similarly, language independence (Lan.I.) reflects the ability of a system to answer questions of any language without having to implement different frameworks.

In the last row we introduce our system, called MaNaLa/SQL, that is an abbreviation for Mapping Natural Language into SQL. In Chapters 6 and 7, we will compare performance with the last six systems, being evaluated on the same data sets we consider.

Table 3.1: State of the Art Systems' Review.

Name	**Automatical Approach**	**Intervention Required**	**Dom. I.**	**DB I.**	**Lang.I.**
EasyAsk	Construct contextual dictionary	-	YES	YES	NO
NaLIX	Disambiguate using ontology	Rephrase questions	YES	NO	NO
SQ-Hal	Use table relationships	Define synonyms	NO	NO	NO
EnglishQuery	Extract few basic relationships	Model refinement	YES	NO	NO
ELF	Extract most relationships	-	YES	YES	NO
MASQUE/SQL	Semantics as logical predicates	Configure domain words	NO	NO	NO
Tiscover	Detect language, check spell	Construct dictionaries	NO	NO	NO
Edite	Exploit mapping tables	Construct dictionaries	NO	NO	NO
Precise	Maxflow algorithm	Rephrase questions	YES	NO	NO
Krisp	Learn a parser using String Kernels	Define the grammar	NO	NO	NO
CatchPhrase	Map typed user requests to SQL queries	Name and define concepts	NO	NO	NO
SemResp	Anwser Driven Semantic Parsing & Classification	-	YES	YES	NO
ModelIII+R	Generative Markov process & Reranking	-	YES	YES	NO
UBL	Induce Probabilistic Grammars from Logical Form	-	YES	YES	YES
DCS	Learn Dependency-Based Compositional Semantics	-	YES	YES	NO
MaNaLa/SQL	(Re)rank pairs using Tree Kernels	Adjust generation rules	YES	YES	YES

Chapter 4

Dataset Generation

The previous chapter introduced the reader to some basic knowledge in Natural Language, Databases and Machine Learning. Despite these three fields are separate worlds and have their own community, in the last fifteen years a lot of progress has been made in cross-disciplinary fields.

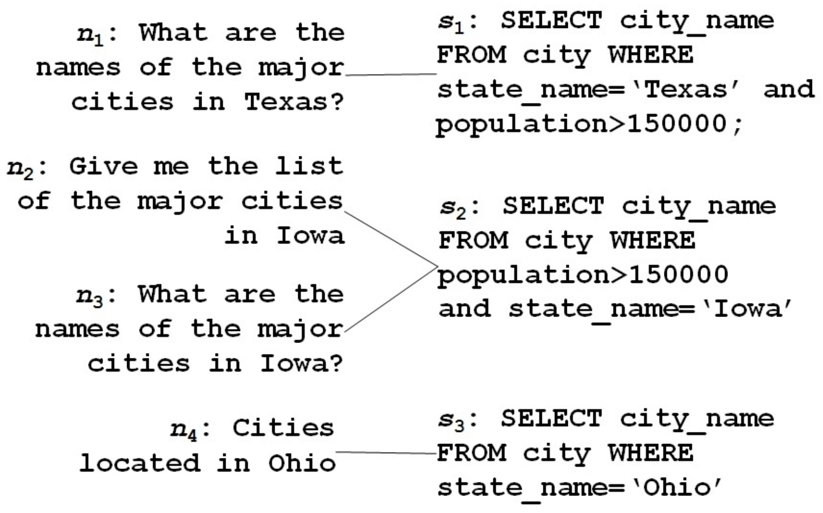
Nowadays, many successful interdisciplinary applications use NLP in Database and Information Systems (NLDB), Machine Learning (ECML) and Principles and Practice of Knowledge Discovery in Databases (PKDD) communities.

However, the idea of integrating databases, natural language and machine learning in a unique framework is rather novel and there is a lack of ready-to-use corpora. Since our goal was the development of a new method to map NL questions into SQL queries based on a machine learning approach, we needed training data, i.e. a set of positive and negative examples, to learn our classification function.

In practical cases, we can assume to have a set of positive examples consisting of correct question/query pairs. For example, correct pairs may be defined when databases are designed and validated. Also, we may ask the DB operator to collect the set of queries that she/he designed in response to typical questions asked by DB users., i.e. such that the execution of the query retrieves a correct answer for the question. Assuming the availability of negative examples is a more strong assumption since providing the correct query for an user information need is a more natural task than providing the incorrect solution. Therefore, to create negative examples, we had to use the initial set of questions and queries in the correct pairs and randomly pair them. Unfortunately, this can generate false negatives since different questions may have more than one answer (and vice-versa), thus a manual verification of such pairs is required. To reduce such costly manual intervention, we exploited the semantic equivalence between the pairs' members and its transitivity closure. This allowed us to extend the set of positive examples to all possible correct pairings, and label all remaining pairings as negative examples.

In the next sections, we describe our approach to automatically generate a gold standard dataset. The main steps are:

- Generalizing concept instances: substitute the involved concepts in questions and their related field values in the SQL queries, e.g. WHERE condition, by means of variables expressing the category of such values (e.g. *What are the major cities in Texas* becomes *What are the major cities in VARstate*).
- Clustering generalized pairs: each cluster represents the information need about a target semantic concept, e.g. "major cities in VarSTATE", common to questions and queries. The clustering can be performed semi-automatically exploiting the semantic equivalence between the pairs' members and its transitivity closure (requires a limited human supervision for validation).
- 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.



n_1: What are the names of the major cities in Texas?

```
s1: SELECT city_name
FROM city WHERE
state_name='Texas' and
population>150000;
```

n_2: Give me the list of the major cities in Iowa

n_3: What are the names of the major cities in Iowa?

```
s2: SELECT city_name
FROM city WHERE
population>150000
and state_name='Iowa'
```

n_4: Cities located in Ohio

```
s3: SELECT city_name
FROM city WHERE
state_name='Ohio'
```

Figure 4.1: Example of an initial dataset.

- 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 steps.

4.1 Generalizing Pairs

Since acquiring training data is the most costly aspect of our design, we should generate the learning set in a smart way. In this perspective, we assume that, in real world domains, we may find examples of questions and the associated queries answering them. 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 just 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 a query belonging to different pairs we cannot assume to only generate incorrect pairs, e.g. when swapping two different queries x with y in the two pairs:

⟨*Which states border Texas?*, x⟩
⟨*What are the states bordering Texas?*, y⟩

we obtain other two correct pairs. Queries x with y could be two syntactically different SQL queries that retrieve the same result set and be, indeed, semantically equivalent.

The aim of pair generalization is to make the detection of semantically equivalent questions and queries easy. Our approach consists in considering questions or queries having similar structures instantiated by the same semantic concepts. The latter are generalizations of important domain terms occurring both in the question and in the related query. For example, terms like *Texas*, *Iowa*, *Ohio*, etc., are substituted with the concept *VARstate* while instances like *Austin* or *Los Angeles* etc., are substituted with the concept *VARcity*.

Note that: (a) concepts correspond to column names (in the database) that naturally stores domain terms; and (b) concepts are expressed in the *WHERE* condition of the given SQL queries. We identify concepts exploiting the latter fact since it is less time consuming that the first that would require to manage large keyword tables.

It is important to note also that, thanks to this inference starting from the unambiguous SQL query, we disambiguate terms like *New York* as a *VARcity* or a *VARstate* based on the associated left-value in the *WHERE* condition. Similarly, we can generalize more that one concept in the same question.

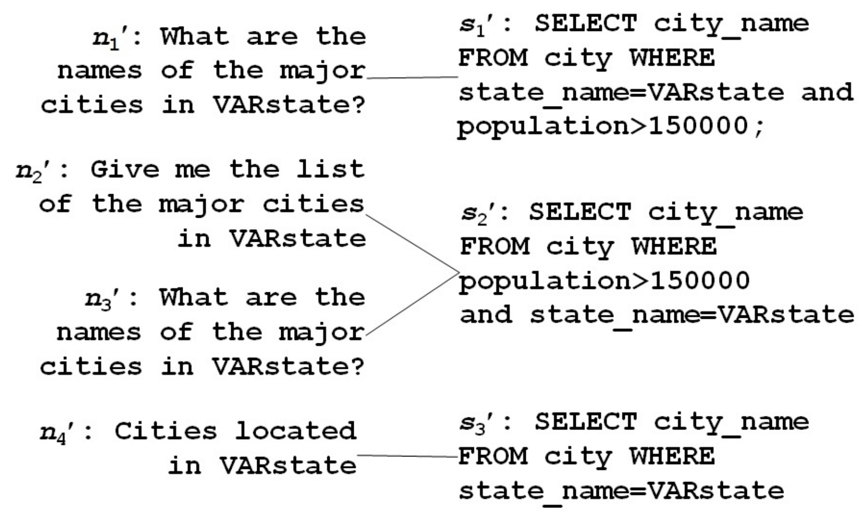

n_1': What are the names of the major cities in VARstate?
n_2': Give me the list of the major cities in VARstate
n_3': What are the names of the major cities in VARstate?
n_4': Cities located in VARstate
s_1': SELECT city_name FROM city WHERE state_name=VARstate and population>150000;
s_2': SELECT city_name FROM city WHERE population>150000 and state_name=VARstate
s_3': SELECT city_name FROM city WHERE state_name=VARstate

Figure 4.2: Example of the generalized dataset.

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 4.1. 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.

We apply the same rule to all given pairs of questions and queries of the initial dataset, and obtain a new generalized version. An example of a generalized dataset is shown in Figure 4.2. The initial pairs, consisting in 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 in Figure 4.1. 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 multiple SQL queries that correctly retrieve an answer to a NL question. In this case we define them to be *semantically equivalent*, i.e.

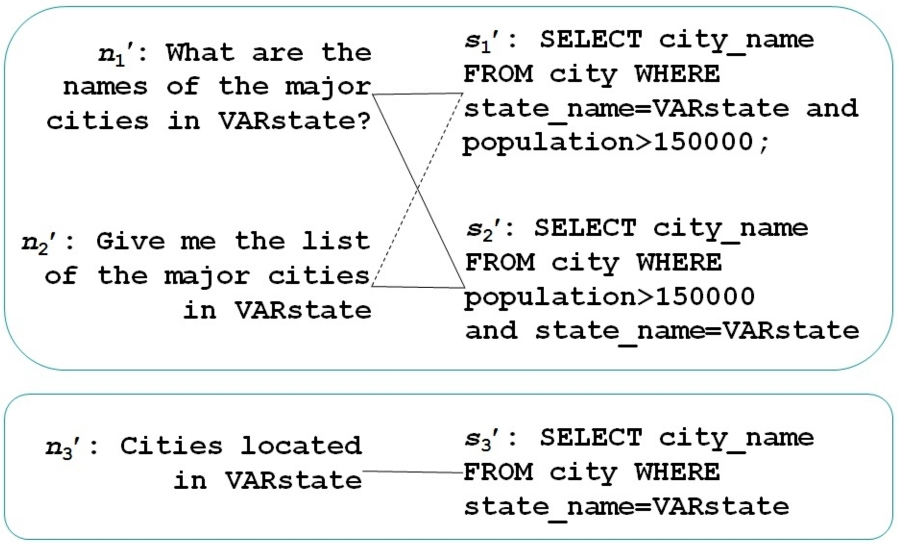

n1': What are the names of the major cities in VARstate?
s1': SELECT city_name FROM city WHERE state_name=VARstate and population>150000;
n2': Give me the list of the major cities in VARstate
s2': SELECT city_name FROM city WHERE population>150000 and state_name=VARstate
n3': Cities located in VARstate
s3': SELECT city_name FROM city WHERE state_name=VARstate

Figure 4.3: Discovering new positive examples inside the same cluster.

$s'_1 \equiv s'_2$. 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 then we can automatically derive new positive examples by swapping their members. We define semantically equivalent pairs those correct pairs with (a) equivalent NL questions, i.e. whose generalized version is the same or (b) equivalent SQL queries. Given that two equivalent queries must retrieve the same result set, we can automatically test their equivalence by simply executing them. Unfortunately, this is just a necessary condition (e.g. two different queries can have the same answer) 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 (after generalization) 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 4.3.b s'_1 and s'_2 retrieve the same result set so we verify that they are semantically equivalent queries (we may ask to a human being whether or not n'_1 and n'_2 are equivalent) 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 pair highlighted with the dashed relation $\langle n'_2, s'_1 \rangle$.

All the remaining pairings, obtained by the Cartesian product between generalized questions and queries, excluding those that we just labelled as correct pairing or, alternatively, by pairing questions and queries belonging to different clusters, are negative examples. Thus, in our sample dataset, the negative instance set is $\langle n'_3, s'_1 \rangle, \langle n'_3, s'_2 \rangle, \langle n'_1, s'_3 \rangle, \langle n'_2, s'_3 \rangle$, as shown in Figure 4.4.

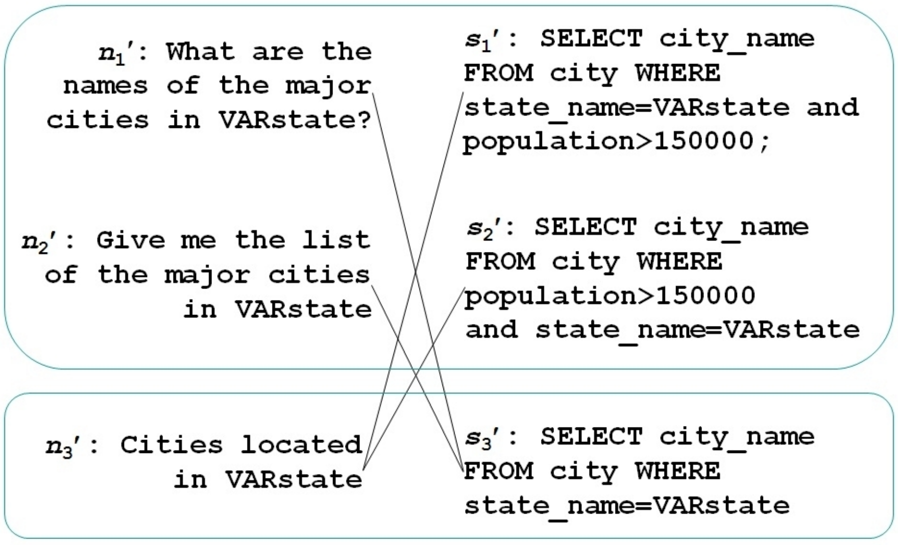

n_1': What are the names of the major cities in VARstate?
s_1': SELECT city_name FROM city WHERE state_name=VARstate and population>150000;
n_2': Give me the list of the major cities in VARstate
s_2': SELECT city_name FROM city WHERE population>150000 and state_name=VARstate
n_3': Cities located in VARstate
s_3': SELECT city_name FROM city WHERE state_name=VARstate

Figure 4.4: Discovering negative examples.

4.2.1 The Clustering Algorithm

The above steps are formally described by the algorithm in Figure 4.5. 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. $M[n, s] = -1 \forall n \in \mathcal{N}, s \in \mathcal{S}$.

Let us indicate with n and s the index of questions and queries in the matrix. 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

```
function CreateFinalDataset(LIST OF PAIRS $I$) returns MATRIX M
M[|$\mathcal{N}$|][|$\mathcal{S}$|] = -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
```

Figure 4.5: Clustering Algorithm

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.

4.3 Representing Pairs

Finally, we also show how to convert the pairs in syntactic structures suitable for the kernel based approaches. For example, let us consider question n_1:"*Which states border Texas?*" and the queries s_1: `SELECT state_name FROM border_info WHERE border='texas'` and s_2: `SELECT COUNT(state_name) FROM border_info`

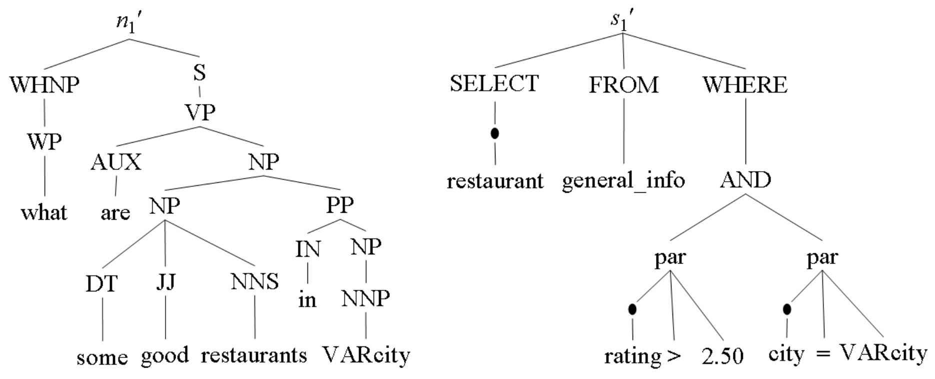


Figure 4.6: Question/Query Syntactic trees

`WHERE border='texas'`. Since s_1 is a correct and s_2 is an incorrect interpretation of the question (corresponding to "*How many states border Texas?*"), 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 cannot only rely on the common terms but we should also take into account the syntax of both languages.

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. Salton [1986]; Joachims [1999]. 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, [Moschitti, 2006; Moschitti et al., 2007; Moschitti and Quarteroni, 2008; Chali and Joty, 2008; Shen and Lapata, 2007; Surdeanu et al., 2008].

Therefore, in addition to BOW, we represent questions and queries using their syntactic trees. As shown in Figure 4.6 for questions (a) we use the Charniak's syntactic parser (Section 2.1.2) while for queries (b) we implemented an ad-hoc SQL parser (Section 2.2.4)

Chapter 5

SQL Query Generation

In the perspective of question answering (QA) targeting the information of databases (DBs), the automatic system only needs to execute one or more Structured Query Language (SQL) queries that retrieve the answer to the posed natural language question. This does not necessarily imply that a semantic parser has to be designed for mapping the meaning of the questions to the one of the queries. Indeed, recent work Giordani and Moschitti [2009] has shown that machine learning algorithms, exploiting syntactic representations of both questions and queries, can be used to automatically associate a question with the corresponding SQL queries. One limitation of the approach above is that the set of possible queries, which a user would execute on the DB, must be known in advance. This because the method can only verify if a given query probably retrieves the correct answer for the asked question: it cannot generate new queries. This limitation is critical as the design of a generative parser which, given a question, feeds the model above with a reasonable set of candidate queries seems inevitably to fall in the category of semantic parsers.

This chapter will demonstrate that it is possible to avoid full-semantic interpretation by relying on (i) a simple SQL generator, which both exploits syntactic lexical dependencies in the questions along with the target

DB metadata; and (ii) advanced machine learning such as kernel-based rerankers, which can improve the initial candidate list provided by the generative parser.

The idea of point (i) can be understood by noting that database designers tend to choose names for entities, relationships, tables and columns according to the semantics of the application domain. Such logic organization is referred to as *catalog*, and in SQL systems it is stored in a database called INFORMATION_SCHEMA (IS for brevity, see Section 2.2.2).

The values stored in IS along with their constraints and data types are important metadata, which is useful to decode a natural language question about the DB domain in a corresponding SQL query. For example, given the IS associated with a DB, shown in Figure 2.4 and 2.2, if we ask a related question, q_0: *Which <u>rivers</u> <u>run through</u> <u>New York</u>?*, a human being will immediately derive the semantic predicate *run_through*(*rivers*, *New York*) from it. Then she/he will associate the argument *river*, which is also the question focus, with the table `RIVER`. Once the latter is targeted, she/he will select the column `TRAVERSE`, which being a synonym of *run through*, provides the same predicative relation asked in the question. Finally, by instantiating the available argument, *New York*, in such predicate, she/he will retrieve the set {`Delaware`, `Allegheny`, `Hudson`} from the column `RIVER_NAME`, i.e., the missing argument (as they are in the same row of *New York*).

The above example shows that several inference steps must be performed to retrieve the correct answer. In particular, lexical relations must be extracted from the questions, e.g., using dependency syntactic parsing and predicate arguments must be expanded with their synonyms or related concepts, e.g., using Wordnet Miller [1995].

Additionally, ambiguity and noise play a critical role in deriving the interpretation of the question described above but we can exploit metadata to verify that the selected sense is correct, e.g., from the fact that *New York* in this database is in the column TRAVERSE, we can gather evidence that the sense of *running-through* matches the one of *traverse*.

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 searched for their stems. Therefore, the general idea is to generate all possible (even ambiguous) queries exploiting related metadata information (i.e., primary and foreign keys, constraints, datatypes, etc.) to select the most probable one using a ranking approach.

Last but not least, we deal with nested SQL queries and complex questions containing subordinates, conjunctions and negations. We designed a generative algorithm based on the matches between lexical dependencies and SQL structure, which allows for building a set of feasible queries.

Starting from the general syntactic formulation of an SQL query, i.e.,:

```
SELECT column
FROM table
[WHERE condition]
```

we generate the set of column, table and condition terms using the lexical relations in the questions. The relation arguments can be generalized using Wordnet and disambiguated using metadata and the execution of the resulting query candidates in the reference DB. Once the list of candidates is available, we can apply supervised rerankers to improve the accuracy of the system. For this step, we improved on the model proposed in Giordani and Moschitti [2009], by designing a preference reranker based on structural kernels. The input of such model consists of pairs of syntactic trees of the questions and queries, where for the query we use their derivation tree provided by the SQL compiler.

For example if a database interface is queried asking for an information, e.g. "*Capital of Texas*", a query in a database language should be executed to retrieve the answer. If the target database is GeoQueries and the language is SQL, its answer can be retrieved executing the query:

```
SELECT state.capital
FROM state
WHERE state.state_name = 'Texas';
```

The result set obtained executing this query is *Austin*. We can see that both language share the words *capital* and *Texas*, but the way they are combined to form the query is not trivial. While this information is hidden and implicit to the reader (thanks to his gained knowledge and experience), the interface may understand that Texas is a *occurrence* of state just because it is stored in the database, whereas the notion that a state has a capital is embedded in the metadata.

In a NL interface to a SQL database, if we want to generate all possible queries for a question q we first need to find their possible SELECT, FROM and WHERE clauses ($\mathcal{S}, \mathcal{F}$ and $\mathcal{W}$ sets) and then combine them in a smart way such that the resulting queries are all syntactically correct and meaningful. Generated queries can be ordered based on some heuristics but if we train a support vector machine and then use it as a reranker we can improve the probability of finding the answer in the top position.

5.1 Building Clauses Sets

The basic idea of our generative parser is to produce queries of the form:

$$\exists s \in \mathcal{S}, \exists f \in \mathcal{F}, \exists w \in \mathcal{W} \text{ s.t. } \pi_s\left(\sigma_w(f)\right) \text{answers q,} \tag{5.1}$$

where question q is represented by means of SDC_q and $\mathcal{S}, \mathcal{F}, \mathcal{W}$ are the three sets of clauses (argument of SELECT, FROM and WHERE). The answering query, $\pi_s(\sigma_w(f))$, can be 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 q. This section shows how we extract terms from the question and categorize them in order to build the sets of clauses above and then how to combine them for generating a query candidate also associated with a generation score.

In Section 2.1.3 we introduced the dependency list (SDC_{q_1}) for the complex NL question "*What are the capitals of the states that border the most populated state?*". The answer to this question can be retrieved using a deeply nested query: first of all (1) *the most populated state* should be selected, then (2) with respect to this state all its *bordering states* are retrieved and finally (3) it retrieves their *capitals*.

Relations holding between stems of the given questions allow to recognize if they will be used in the $\mathcal{S}$ and/or in the $\mathcal{W}$ sets. Moreover, the hierarchy represented by a dependency list allows to follow the correct recursive steps to generate nested SQL queries with respect to subordinates in the NL question.

Indeed, given a question q we start from its SDC_q and (a) prune and stem its components, (b) add synonyms, (c) create the sets of stems select and/or project oriented and (d) keep only dependencies possibly used in the recursive step to generate nested queries. Finally (e) we look for matching stems both in the metadata and in the database to build $\mathcal{S}$ and $\mathcal{W}$. Building the set $\mathcal{F}$ from $\mathcal{S}$ and $\mathcal{W}$ is straightforward.

5.1.1 Motivating Examples

We now briefly discuss some examples to introduce the objective of each individual step and clarify how the entire process is carried out.

From the simple question "*Capital of Texas*" introduced above we have to derive the correct answering query based on only two stems. The key of categorizing stems (Section 5.1.3) 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 by the column name under where it appears (Section 5.1.5).

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. the root of the sentence and the subject dependent are typically used for projections). This list needs to be preprocessed (section 5.1.2) to take into account only relevant relations between *stems* of the question.

Let us consider for example the question: "*What is the capital of the most populous state?*" and its answering query:

```
SELECT state.capital
FROM state
WHERE state.population =
  (SELECT max(state.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, see Section 5.1.4).

Finally we introduce one last example to clarify Section 5.1.6. 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 state.capital
FROM ...
WHERE border_info.border = 'Texas':
```

The problem is: how can we fill in the dots in the FROM clause? Fields *capital* and *border* belong respectively to tables *state* and *border_info*. From 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
```

5.1.2 Optimizing the Dependency List

As introduced in Section 2.1.3, we do not use all grammatical relations provided in output by the Stanford Dependency parser. For this reason before processing the list of dependencies we filter it pruning useless relations and removing from *gov*ernors and *dep*endents the appended number indicating the position of the word in question q. Moreover we eliminate stems of 3 or less characters since they would introduce too much noise in retrieving matching strings. In contrast, useful function words such as *in, of, not, or, and,* etc. are embedded in the dependency type, e.g., *prep_**of**(capital, state)*.

Then, *gov*s and *dep*s are reduced to stems (as discussed in Section 2.1.1).

Finally, we enlarge the probability to match stems and the metadata by means of substring matching (similar to what is done in Wikipedia or in Google query search). A list of synonyms for each term can be also created using Wordnet and can be used to increase the matching coverage with the metadata terms.

We call the preprocessed list SDC_q^{opt} and we use it for the next step.

For example, with respect to the original SDC_{q_1} introduced in Section 2.1.3, the optimized list is the following:

$$SDC_{q_1}^{opt} = root(\text{ROOT-0, are-2}), nsubj(\text{are-2, capital}),$$
$$prep\_of(\text{capital, state}), nsubj(\text{border, state}),$$
$$rcmod(\text{state, border}), advmod(\text{populat, most}),$$
$$amod(\text{state, populat}), dobj(\text{border, state})$$

Figure 5.1: Resulting $SDC_{q_1}^{opt}$

5.1.3 Stem Categorization

To build $\mathcal{S}$ and $\mathcal{W}$ sets, we identify the stems that can most probably participate in the role of projection (i.e., composing the SELECT argument) and/or selection (composing the WHERE condition). Accordingly, we create two sets of terms Π and Σ. The main idea is that some terms can be used to choose the DB table and column where the answer is contained whereas others tend to indicate properties (i.e., table rows) useful to locate the answer in the column. For example, in case of q_0 (see introduction): *Which rivers* may indicate that *river* is a SELECT argument whereas *run_through* and *New York* may be part of the WHERE argument, thus forming the query: SELECT *river_name* FROM *river* WHERE *traverse=“new york”*.

The Algorithm

We use SDC_q to automatically extract and classify such terms and relations from a question. For example, the *dep* of *root* is typically the main verb (i.e., relation) of the question, which can be used to derive properties of the question focus. Thus, it tends to be of type selection (i.e, it belongs to Σ set) like in *root(*ROOT*, run)*. In case of an *nsubj*, the *gov* is typically a verb relation and it can be used to build *specializers* whereas the *dep*, i.e., the subject, is most probably a projection candidate such as for example in *nsubj(run, rivers)*. In the following, we report our heuristics for term categorization. It should be noted that they do not produce a precise and disjoint term separation but the obtained two sets are smaller than the overall term set, thus reducing the computational complexity in generation. We analyze grammatical dependencies *rel*(*gov,dep*) in SDC_q in their parse order and classify their arguments (stems) in Π and Σ categories according to the following rules:

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}. Set the flag *hasRoot* to true. 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. If it starts with *nsubj*, we use it to add stems to Π. Set the flag *hasSubj* to true.
   - if $gov \in \Sigma$ add *dep* to Π and remove *rel* from SDC_q^{opt}.
   - otherwise if $gov \notin \Sigma$ and *hasRoot* is false, add *gov* to Σ and *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):
   - if $gov \in \Pi$ and if there is no *table.column* like[1] *gov.dep* add *dep*

[1]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 $\pi_{table\_name}\,(\sigma_{table\_name \cong dep \wedge column\_name \cong gov}(IS.Columns))$. 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%".

to Σ, otherwise, also add *dep* to Π. Remove *rel* from SDC_q^{opt}.

- otherwise if $gov \notin \Pi$ *hasRoot* and *hasSubj* are false add *gov* to Π. If there is not any *table.column* like *gov.dep* add *dep* to Σ, otherwise, also add *dep* to Π. Remove *rel* from SDC_q^{opt}.
- otherwise keep it, since we will need it in the recursive steps.

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 dependencies 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 algorithm recursively building Π', Σ', Π'' and Σ'' until SDC_q^{opt} is empty.

Projection-oriented Set Π

The algorithm creates this set looking for stems that should be used to project a certain database field. These are the stems that are involved in the sentence or in its subordinates, as subjects (or objects, if the sentence is passive). If a preposition refer to stems that are both components of the same table (e.g. *prep_of(capital, state)* referring to `state.capital`) and one of the two has already been labelled as projection-oriented, the other one is also added to Π. To populate this set the algorithm only exploits metadata.

Selection-oriented Set Σ

The root of the syntactic tree is typically the focus of the sentence. It is usually a verb, specifying a condition that the answer should satisfy. For this reason, everything that could be useful to build the expression that select those tuples is included in this set. Again, stems that are part of the same table are kept together (e.g. *rcmod(state, border)* referring to `border_info.state_name`) while those that are not directly related to the focus of the sentence will be used in recursive instances of the algorithm.

Recursive Steps

These rules should be applied iteratively to each relation until the list is empty. When a nesting query is needed (e.g. a new subject is discovered, a superlative occurs, etc) the algorithm is recursively performed until no more rules can be applied, and the resulting Π' and Σ' are combined. The algorithm stops the iteration after the third recursive step as far as there resulting queries would otherwise be too complicated and since the number of generated queries grows exponentially. Once all the stems are categorized and/or the list is empty, the algorithm exploits the sets Π and Σ to build matching clauses (i.e. clauses that contain one or more matching stems) starting from the inner sets.

Before discussing how this is performed, let us clarify the above-listed rules by means of a real example. Figure 5.2 shows the initial list of optimized dependencies $SDC_{q_1}^{opt}$ and all projection and/or selection oriented sets generated through two recursive steps.

At the first iteration, we use $ROOT$ to add *are* to Σ. Then, the *nsubj(are,capital)* suggests that the subject *capital* may be a focus of a projection thus we included in Π. Additionally, given *prep_of(capital, state)*, *state* is a modifier of the subject thus it may have the same role and we include it in Π. We immediately verify this assumption by automatically checking that there is an occurrence of `state.capital` in IS. Being been already processed the three dependencies above are deleted from $SDC_{q_1}^{opt}$ obtaining $SDC_{q_1}^{opt\prime}$ (consisting of relations 4-8) used in the next iteration (note also that since *are* is a short stem, it should be deleted from Σ).

In the second iteration, there is no *root* dependency anymore thus we start from *nsubj(border,state)* which leads to add *border* to Σ' and *state* to Π'. Additionally, since the relative clause dependency *rcmod(state,border)* is supported by the occurrence of `border.state_name` in IS, *border* is also added to Π. In contrast, *dobj(state,border)* is not supported by the presence of `state.border` in IS, thus their terms are not added to the sets. The dependency is deleted from $SDC_{q_1}^{opt\prime}$ obtaining $SDC_{q_1}^{opt\prime\prime}$ (consisting in relations 6-7) for the last iteration.

In the third and last iteration, we still have the *mod* dependencies, thus we add all their stems to Π'' and delete their associated dependencies from the list.

5.1.4 SELECT Clauses

We use the set Π to retrieve all the metadata terms that match with its elements: this will produce S according to the generative grammar shown in Figure 5.3. The arguments of the grammar are derived by executing several queries to find all matching stems and retrieve a list of *table.column* terms augmented by aggregation operators (we also associate a plausibility weight w_i with each element of $\mathcal{S}$, see Section 5.2.3). For example,

(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),	$\Sigma' = \{\text{border}\}$
(6)*advmod*(populat, most),	
(7)*amod*(state, populat),	$\Pi'' = \{\text{most, populat, state}\}$
(8)*dobj*(border, state)	$\Sigma'' = \phi$

Figure 5.2: Categorizing stems into projection and/or selection oriented sets

considering the IS scheme in Figure 2.2.2, the SELECT clauses that are generated from Π, whose elements are listed in the right side of Figure 5.2, are shown in Figure 5.4 (the superscript numbers just indicate the weight associated with each statement).

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

Figure 5.3: Clauses generative grammar for fields matching stems in Ξ

Extracting Matching Fields

According to this grammar, the stems can map with table names as well as with column names. To retrieve such fields from the metadata, the following SQL query is executed over the database catalog every time we need to look up for a matching with stem x in the metadata of a fixed `DB`.

```
SELECT table_name,column_name
FROM INFORMATION_SCHEMA.columns
WHERE (table_name LIKE "%x%" OR column_name LIKE "%x%")
AND table_schema = "DB"
```

This query looks for a partial (substring) matching of the stem x among all the table names (retrieving also all the columns belonging to the matched tables) and among all the column names (extracting also the names of the table to which they belong). The result set consist of a two column table, and the partial list of fields $\mathcal{S}$ is obtained linking each row with the dot separator.

Adding Aggregation Operators

In the previous step we obtain a list of fields $\mathcal{S}$ in the format `T.C` that can be extended adding some aggregation operators based on the type of the fields. If the field type is compatible with numbers we apply all the operators (sum, average, minimum and maximum). Otherwise, if it is a textual field it makes sense to apply only the operator that counts how many different values appear in the field.

The DB catalog can be inspected in the following way to retrieve the subset of textual fields that are extended adding `COUNT(T.C)`.

```
SELECT table_name,column_name
FROM INFORMATION_SCHEMA.columns
WHERE table_name = T AND column_name = C
      AND (data_type = "TEXT" OR data_type LIKE "%CHAR%")
      AND table_schema = "DB"
```

All the other fields that are in $\mathcal{S}$ but not in the result set of this query, are extended adding clauses `SUM(T.C)`, `MAX(T.C)`, `MIN(T.C)`, `AVG(T.C)` and `COUNT(T.C)`.

For example, considering the IS scheme in Figure 2.2.2, the SELECT clauses originated from Π of Figure 5.2 are shown in Figure 5.4. Note that the superscript numbers indicate the weight associated with each statement.

In fact we associate a weight w_i to each element of $\mathcal{S}$, according to the procedure described in Section 5.2.3. In particular we take into account

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

Figure 5.4: A subset of SELECT clauses for q_1

that words ending with *-st* are superlatives so that they can map with operators MAX and MIN.

5.1.5 WHERE Clauses

For generating the WHERE clauses, we need to divide Σ in two distinct sets: Σ_L and Σ_R, for the left-and right-hand side of the condition, respectively. The set Σ_L contains stems matching the IS metadata terms. Σ_L is used to generate the left condition $\mathcal{W}_L$, with the rule $\mathcal{W}_L \rightarrow$ FIELD, where FIELD is the same of Figure 5.3, where Σ_L is used in place of Ξ (this is the same task as illustrated in Section 5.1.4).

In contrast, $\Sigma_R = \Sigma - \Sigma_L$ is used to generate $\mathcal{W}_R$ as follows:
$\forall col \in \text{IS}.Columns$, $\forall tab \in \text{IS}.Tables$,

$$\mathcal{W}_R = \left\{ x | \pi_{count(*)} \left(\sigma_{col \cong x}(Geoquery.tab) \right) \geq 0, x \in \Sigma_R \right\}. \tag{5.2}$$

It is essentially a database lookup, seeking for those values that match stems of Σ_R. This is necessary since we tokenized and stemmed all the words in the original sentence and we have to make a step backward to gather the right matchings. For example, if Σ_R contains the stems *new* and *york*, $\mathcal{W}_R$ will be first populated with values *"new york"*, *"new mexico"*, *"newark"*, and then finding another mapping with *"new york"* through the stem *york*, the correct one becomes also the one with higher weight (the weighing scheme will be discussed later on).

It is worth noting that non-matching stems may semantically match a whole condition and need to be handled carefully. For example *major*, if associated with *city*, is translated into `city.population>150000` while when talking about river is associated with `river.length>750` [Cimiano and Minock, 2009]. This kind of knowledge should be handled manually, adding such clauses to $\mathcal{W}$ when the stem *major* appears in the question.

Creating Basic Expressions

To build the WHERE clauses set $\mathcal{W}$, we first generate basic expressions in the form $expr = e_L$ OP e_R, $\forall e_L \in \mathcal{W}_L, \forall e_R \in \mathcal{W}_R$. If the type of e_L is numerical then OP=$\{<, >, =\}$, otherwise we apply the LIKE operator.

To better understand how it works, let us introduce a new example question q_2: "*What are the capitals of states bordering New York?*". The list $SDC^{opt}_{q_2}$ and the derived sets of stems are shown in Figure 5.1. The set Σ' is split into $\Sigma'_L = \{\text{border}\}$ and $\Sigma'_R = \{\text{new, york}\}$. 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, i.e., $\mathcal{W} = \{$`border_info.border='new york'`^{5},`border_info.state_name='new york'`^{4},...$\}$.

Nested Expressions

In a query that requires nesting the right hand side of the WHERE clause can't be derived directly from stems (e.g. Σ in Table 5.1). In fact, those stems are used in the recursive step to obtain the nested subquery (e.g. "*states bordering New York?*"). Indeed, $\mathcal{W}\ _R$ can contain also whole subqueries, and in that cases the operators that builds the entire expression

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

Table 5.1: Another example of a subordinate question that requires nesting.

could be OP=$\{<, >, =,$IN$\}$. In particular, the IN operator works well also when the inner query retrieves a textual result set of one or more rows.

Inheriting Basic Expression and Combining Them

It is worth noting that it may happen that one or more stem originate a WHERE clause that should be shared across nested subqueries and in different levels of recursion. Take for example the question "*What is the biggest city in Texas?*" and it's answering query:

```
select city.city_name
from city
where city.state_name = 'texas' and
      city.population = (select max(city.population)
                         from city
                         where city.state_name = 'texas')
```

The clause `city.state_name = 'texas'` appears twice: in the inner query it is used to retrieve the maximum population of cities in the state of Texas. This number is used in the outer query seeking for the city name to which this number of citizen corresponds. The outer clause `city.state_name = 'texas'` is needed to ensure that it does not retrieve cities with the same population number but not in Texas.

Similarly, for the question "*What is the largest city in a state that borders Texas?*" the same clause is shared by two subqueries.

```
select city.city_name
from city
where city.state_name in (select border_info.border
                         from border_info
                         where border_info.state_name='texas')
and city.population=(select max(city.population)
                     from city
```

```
                          where city.state_name in
                             (select border_info.border
                              from border_info
                              where border_info.state_name='texas'))
```

The need of sharing WHERE clauses between recursive instances applies only to basic expressions (those that are not combined together and do not involve nesting). For this reason the set $\mathcal{W}$ is composed by the set of basic clauses $\mathcal{W}^G$ that are globally considered, and the set $\mathcal{W}^L$ of clauses that are used only locally for nested queries. The final $\mathcal{W}$ at each iteration is indeed the union of these sets enriched by their combinations by means of conjunctions and negations. In particular, $\forall e', e'' \in \mathcal{W}^L$, new expressions e' AND e' are added to $\mathcal{W}^L$ only if e'_L is different from e''_L . Then, all these clauses are combined with clauses in $\mathcal{W}^G$ using AND/NOT operators (see Section 2.2.3), keeping only those expressions *expr* so 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.

Dealing with Missing Pieces

It could happen that the set Σ_R is empty. For example, when the WHERE condition requires nesting: in this case e_R will be the whole subquery (e.g. Σ' in Figure 5.2). 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 Figure 5.2). In any case, even if there are no selection-based stems, $\mathcal{W}$ may not be empty (e.g. Σ in Figure 5.2). 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 $col \in \pi_{column\_name}\left(IS.Columns\right)\}$.

5.1.6 FROM Clauses

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, KEYS; see Figure 2.2). 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.

Retrieving Used and Useful Tables

First of all, we extract tables appearing in $\mathcal{S}$ and $\mathcal{W}$. This is performed looking at all the words that end with a dot in the clauses sets, creating a set F. Note that there are no weight associated with FROM clauses because it's not possible to backtrack how many stems made each table appear in F.

Looking up in Metadata for Table Joins

At this point $\mathcal{F}$=F. Then we add two-table joins pairing tables of F: $\forall t_1, t_2 \in F$ we find c_1, c_2 using the following query:

$$\pi_{column\_name, ref\_column\_name}\left(\sigma_{table\_name=t_1 \wedge ref\_table\_name=t_2}(\text{IS}.Keys)\right) \tag{5.3}$$

The result set is used to add to $\mathcal{F}$ the join ${}_{t_1}\bowtie_{t_2}^{}{}_{c_1=c_2}$, that is: t_1 `join` t_2 `on` $t_1.c_1 = t_2.c_2$.

Going back to our example with question q_1, the set of FROM clauses is: $\mathcal{F}' = \{$`state,border_info,state join border_info on state.state_name=border_info.border,...`$\}$.

Two-jumps JOIN Clauses

In addition we can allow for more distant joining finding an intermediate table useful to link two tables that are not directly referencing each other. This table is typically the main table, that is, the one that is referred by all the other tables (e.g. In *GeoQuery* it is *state*). This can be done executing the following SQL query that joins two instances of KEYS for each tables pair `T1, T2` in F and their columns `C1, C2`.

```
select count(*)
from key_column_usage as tmp1,key_column_usage as tmp2
     on tmp1.referenced_table_name=tmp2.referenced_table_name
where tmp1.referenced_table_name=F and
      tmp1.referenced_column_name=tmp2.referenced_column_name
      and tmp1.table_schema="db" and
      tmp1.table_name=T1 and tmp2.table_name=T2 and
      tmp1.column_name=C1 and tmp2.column_name=C2
```

If the returned value is not zero, we can add to $\mathcal{F}$ the join $\underset{C_1=C_2}{T_1 \bowtie T_2}$.

In this way, even if *border_info* and *river* are not directly linked, we can derive the far joining between the two:

```
river JOIN border_info ON river.traverse=border_info.state_name
```

5.2 Composing 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.

5.2.1 Clause Cartesian Product

In order to find possible answering queries we generate the set $\mathcal{A}$={SELECT $\mathcal{S}\times$ FROM $\mathcal{F}\times$ WHERE $\mathcal{W}\}$ $\cup$ {SELECT $\mathcal{S}\times$ FROM $\mathcal{F}\}$. 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 contain on average up to ten items, their product can result in a very huge set. Thus, when generating all pairing 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 to generate incorrect queries and to waste time trying to execute them.

A pair $\langle s, f, w\rangle$ or $\langle s, f\rangle$ is added to $\mathcal{A}$ only if it satisfy the rules:

- if s is in the format $t.c$, then $t. \in f$.
- for every $t.c$ appearing in w^L, that is, in a left-hand sided expression,(before a $\{=, >, <, \text{IN}\}$) then $t. \in f$.

The notation $t. \in f$ ensures that the table name does not conflict with some column name ($t.$ just stands for the concatenation of t with the dot symbol used to separate tables and columns).

If one of these two rules is not satisfied, executing the corresponding query would lead to an `Unknown table` error. To make a simple example, we illustrate in Figure 5.5 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`.

The second rule ensures that $t.c$ appears in the running query, and not only in a subquery. For example, when answering "*capital cities*", the system recognizes that the following query is incorrect since table `city` does not appear in the f.

```
SELECT state.capital
FROM state
WHERE city.city_name IN
  (SELECT state.capital
   FROM state)
```

Note that in the inner query there is no WHERE clause, so the second rule holds for every f. A correct answering query to our running example is generated and checked for correctness in two levels: first the inner query is generated and added to $\mathcal{A}'$, then it is embedded in the WHERE clause of the outer query, adding to $\mathcal{A}'$ the following query:

```
SELECT city.city_name
FROM city
WHERE city.city_name IN (SELECT state.capital
                         FROM state)
```

5.2.2 Pruning Useless Queries

Once the set $\mathcal{A}$ of all valid pairings is built, we additionally prune some of them that are not useful, obtaining the final set $\bar{\mathcal{A}}$. For example, when answering "*Mississippi rivers*", the system disambiguates between the state and the river called Mississippi, since generating the following query is meaningless (projecting the same field compared to a value in the selection is useless).

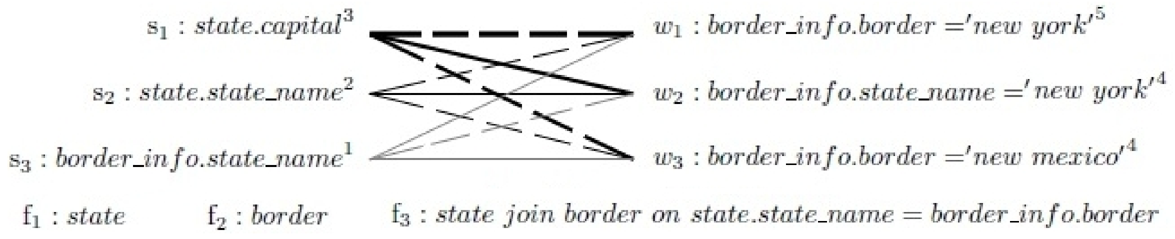
```
SELECT river.river_name
FROM river
WHERE river.river_name = "Mississippi"
```

Moreover there could be redundant queries that if optimized allow us to remove duplicates in the set, reducing its cardinality. Looking at the query on the top of this page, that retrieves all cities that are also capitals, since every capital is also a city, it can be optimized considering only the inner query.

In practice every pair $\langle s, f, w\rangle$ is added to $\bar{\mathcal{A}}$ only if it satisfy the following rules.

1. if $f = t$ and $t.c$ appears in w^L (before a $\{=, \text{IN}\}$) as a single clause (not combined with AND/OR and other expressions), then $t.c \notin s$.
2. if f is in the format t_1 JOIN t_2 ON $t_1.c_1 = t_2.c_2$ then $t_1 \in s$ AND $t_2 \in w^L$ OR $t_1 \in w^L$ AND $t_2 \in s$.
3. if f is in the format t_1 JOIN t_2 ON $t_1.c_1 = t_2.c_2$ and $t_1.c_1 \in s$ then $t_2.c_2 \notin w^L$.
4. if f is in the format t_1 JOIN t_2 ON $t_1.c_1 = t_2.c_2$ and $t_2.c_2 \in s$ then $t_1.c_1 \notin w^L$.

For instance, with respect to Figure 5.5, the pairing $\langle s_3, f_2, w_2\rangle$ answers the question "*Which state is New York?*" and is clearly useless. An example of a redundant query follows from the pairing $\langle s_2, f_3, w_1\rangle$. It requires that the columns *state.state_name* and *border_info.border* are the same, so w_2 would select the same rows of w'_2(i.e. *state.state_name='new york'*),



s_1 : state.capital^3
s_2 : state.state_name^2
s_3 : border_info.state_name^1
w_1 : border_info.border ='new york'^{5}
w_2 : border_info.state_name ='new york'^{4}
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

Figure 5.5: Possible pairing between clauses for q_2. Actually Π' have been merged with Π and Σ' with Σ, optimizing by avoiding nesting with equivalent joining.

but this means that table *border_info* is no longer used and this pairing is equivalent to $\langle s_2, f_1, w'_2 \rangle$ that, as said above, is meaningless.

Regarding those pairs that do not involve any WHERE conditions, i.e $\langle s, f \rangle$, they are added to $\bar{\mathcal{A}}$ only if f does not involve joining tables i.e. it is a single table. In practice we add a lot of simple queries, even though they are trivial and potentially useless. However, as we already seen in the example of "*capital cities*" they are needed in recursive steps and, sometimes, also as final answering queries (e.g. $\langle s_3, f_2 \rangle$ would retrieve the correct answer to question "*Which states border another state?*"). Nevertheless, these kind of queries are very useful when handling questions with superlatives, e.g. "*What is the capital of the most populous state?*" (see the example previously reported in page 64).

5.2.3 Weighing Scheme

As introduced in 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 name are taken into account only once, without summing both weights). Superlatives matching with aggregation operators counts 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 values 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 the weight of its components, and it's used to order the final set $\bar{\mathcal{A}}$ of possible useful queries from the most probable to the less one.

Figure 5.5 highlights this *probabilistic* score by means of the thickness of connection lines (dashed lines illustrate pruned queries). The final ordered set answering q_2 is the following one:

$$\bar{\mathcal{A}} = \{\langle s_1, f_3, w_2\rangle^7, \langle s_3, f_2, w_1\rangle^6, \langle s_2, f_3, w_2\rangle^6, \langle s_1, f_1\rangle^3, \langle s_2, f_1\rangle^2, \langle s_3, f_2\rangle^1\} \tag{5.4}$$

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 than a query can have the same weight. To deal with that, we implemented a comparator that privilege 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 consists of a unique table.

5.3 Reranking Model

Once an initial rank of the candidate SQL queries has been derived, we can rely on machine learning methods for improving the probability of finding the correct answer in the top position. The complexity is given by the need of designing suitable representations of the question and query pairs. For this purpose, we rely on kernel methods.

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

5.3.1 Preference Reranking

Our reranking model consists in learning to select the best candidate from a given candidate set. To use SVMs for training a reranker, we applied Preference Kernel Method [Shen and Joshi, 2003]. 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); instead, the standard one-versus-all binarization method is still applied.

5.3.2 Preference Kernel

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: $P_K(x, y) =$

$$P_K(\langle x_1, x_2\rangle, \langle y_1, y_2\rangle) = K(x_1, y_1) + K(x_2, y_2) - K(x_1, y_2) - K(x_2, y_1), \quad (5.5)$$

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[2] in Eq. 5.5.

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

Chapter 6

Experimental Evaluation on Classifying Mapping

We ran several experiments to evaluate the accuracy of our approach in automatically selecting correct SQL queries for NL questions, where the selection of the correct query is modeled as a ranking problem. The ranker is constituted by SVMs and the kernels described in Section 2.3. We addressed the problem of finding a query whose result answers to a question according to the following ranking problem.

Given a question $n \in \mathcal{N}$ and the complete set of the available queries $\mathcal{S}$, we classified the set of all possible pairs $P(n) = \{\langle n, s \rangle : s \in \mathcal{S}\}$. Then we used the classification score to rank the element of $P(n)$ and select the pair with the highest score.

To show the generality of our approach we created two different datasets by applying our algorithm described in Section 4 to two different corpora.

6.1 Setup

To learn the classifier we used SVM-Light-TK[1], which extends the SVM-Light optimizer Joachims [1999] with tree kernels. i.e. Syntactic Tree Kernel (STK) and its extension (STK_e) as described in Section 2.3.2. We

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

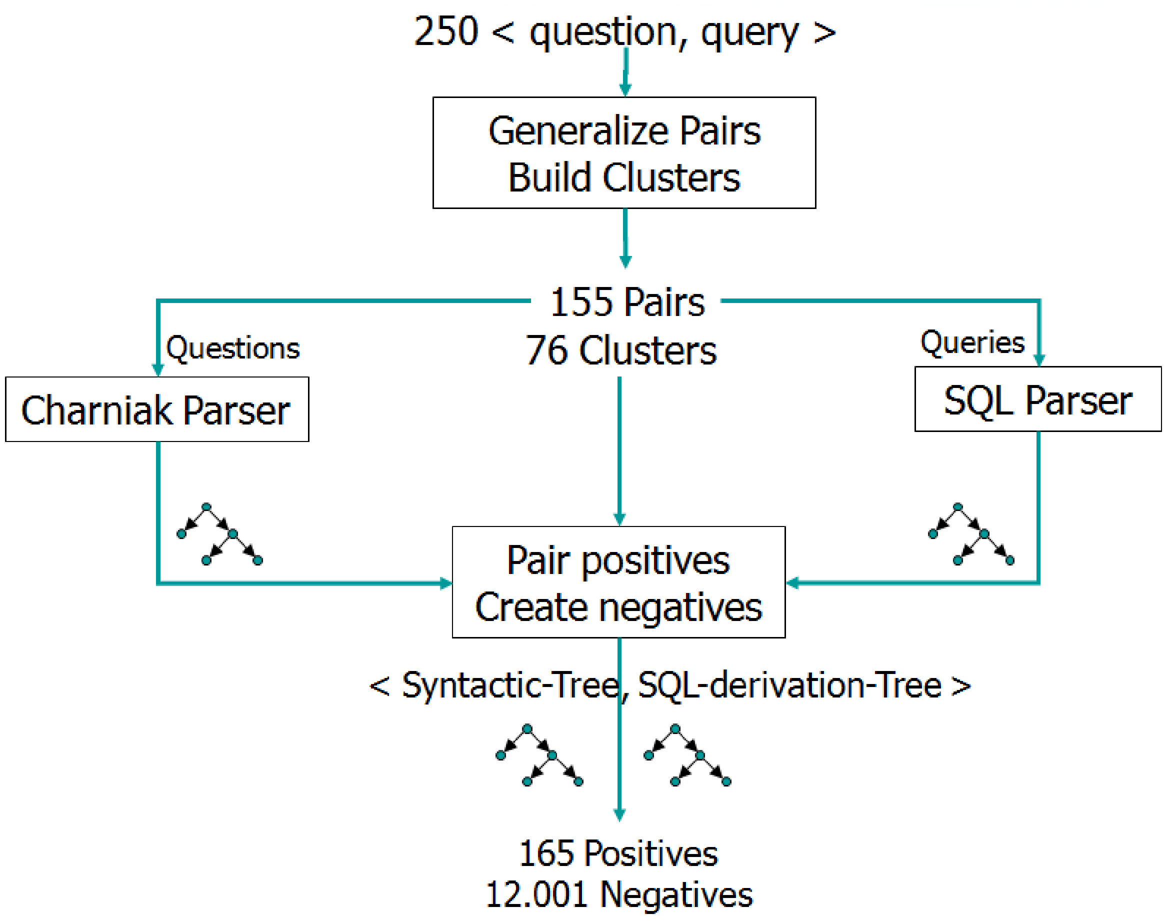


Figure 6.1: Building the GEO dataset starting from GEOQUERIES

model 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. 2.3.2).

To generate our datasets we applied our algorithm described in Section 4 to GEOQUERIES250 and RESTQUERIES corpora[2].

The first corpora is about geography questions. After the generalization process the initial 250 pairs of questions/queries were reduced to 155 pairs containing 154 NL question and 79 SQL queries. We found 76 clusters,

[2]Questions in both corpora were originally collected from a web-based interface and manually translated into logical formulas in Prolog by Mooney's group [Tang and Mooney, 2001]. Popescu et al. [2003] manually converted them into SQL. Thanks to our clustering algorithm we discovered and fixed many errors and inconsistencies in SQL queries.

from which we generated 165 positive and 12.001 negative examples for a total of 154 $\times$ 79 pairs. Such dataset will be referred to as GEO.

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 address these problems by keeping only pairs whose members have at least two words in common since, intuitively, a positive pairing share at least a variable and a concept (e.g. *VARstate cities*). In this way we excluded 10.685 incorrect pairs. Actually, among the excluded pairs there are 10 positive examples. However since the NL questions of those pairs are paired with other equivalent SQL queries in those pairs that are not exlcuded, this doesn't affect the learning. This dataset, that in the remainder we call GEO, indeed consists of 155 positive and 1.316 negative examples for a total of 1.471 pairs.

It is worth noting that even if an extension of this dataset is available GEOQUERIES880 and it has been widely used as a benchmark by other system, we decided to avoid evaluating our classification approach on this larger corpus. The reason is twofold: first of all, as shown in Table 6.1, the number of negative examples generated with our algorithm grows exponentially in the number of initial given pairs. Even if we can reduce the number of negative examples discarding pairs of questions and queries that do not have stems in common, the proportion with positive examples remains unbalanced, leading to a mistrained model. Secondly, after conducting a pilot examination of the first 700 initial pairs[3], we discovered 243 redundant questions: only the variable changes (e.g. *Texas* VS *Iowa*) so when we generalize them as "VARstate" only one occurrence is

[3]a subset of the 880 NL question provided by Mooney paired with SQL queries translated by Popescu that we automatically checked and then manually corrected.

Table 6.1: Number of pair generated starting from increasing percentage of the GEO880 corpus.

Initial Pairs	**Gen. Positives**	**Gen. Negatives**	**Red. Negatives**
200	171	10779	3507
400	283	25717	8616
800	587	126502	48103

taken into account. Moreover even if they have different NL questions a lot of them are paired with the same generalized SQL. While this larger set contains 207 clusters, taking into account the GEOQUERIES250 subset, consisting of a selection of 250 pairs, we discover 76 cluster representatives. While the redundancy rate is the same in both corpus (each information is represented with 3.3 pairs on average) learning results achieved on the smallest one are usually lower compared with evaluation on the larger corpus. However we expect to perform better with the smallest one, given that our approach can deal with generalized pairs in stead of relying on training and test sets containing multiple instances of the same pair.

The second dataset regards questions about restaurants. The initial 250 pairs were generalized by 197 pairs involving 126 NL questions and 77 SQL queries. We clustered these pairs in only 26 groups which lead to 852 positive examples and 9.702 negatives. Such dataset will be referred to as REST. To speed up the training algorithm we eliminate some negative examples considering as irrelevant those pairs that share only one stem or no stem at all. In this way we exclude 2.450 incorrect pairs and we don't commit any error during this pre-processing. The final dataset, that indeed consists of 7.252 pairs, will be referred to as REST.

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.

6.2 Results

6.2.1 Evaluation on Geo dataset

We tested several models for ranking based on different kernel combinations whose results are reported on Table 6.2 and Table 6.3. The first column of Table 6.2 lists kernel combination by means of product and sum between pairs of basic kernels used for the question and the query, respectively. The latter column shows the average accuracy (over 10 folds) $\pm$ Std. Dev.

More in detail, our basic kernels are: (1) linear kernel (LIN) built on the bag-of-stems (BOS) of the questions or of the query; (2) a polynomial kernel of degree 3 on the above BOSs (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). Note that we can also sum or multiply different kernels, e.g. POLY$\times$STK.

An examination of the reported figures suggests that: first, the basic traditional model based on linear kernel and BOS, 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. 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%. 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; the traditional approach would have required to build the Cartesian product of the question BOS by query BOS. This can be very large, e.g. 10K features for both spaces leads to a pair space of 100M features.

Second, the baseline model LIN+LIN confirms that the feature pair space is essential since the accuracy of all kernels implementing individual

Table 6.2: Kernel combination accuracies for GEO dataset

Kernel Combination	**Accuracy $\pm$ Std. Dev.**
LIN + LIN	57.3$\pm$10.4
LIN $\times$ LIN	70.7$\pm$12.0
POLY $\times$ POLY	71.9$\pm$11.5
STK $\times$ STK	70.3$\pm$9.3
STK$_e$ $\times$ STK$_e$	70.1$\pm$10.9
LIN $\times$ STK	74.6$\pm$9.6
LIN $\times$ STK$_e$	**75.6$\pm$13.1**
POLY $\times$ STK	73.8$\pm$9.5
POLY $\times$ STK$_e$	73.5$\pm$10.4
STK $\times$ LIN	64.7$\pm$11.5
STK$_e$ $\times$ LIN	68.3$\pm$9.6
STK $\times$ POLY	65.4$\pm$10.9
STK$_e$ $\times$ POLY	68.3$\pm$9.6

spaces (e.g. kernels which are sums of kernels) is much lower than the baseline model for feature pairs, i.e. LIN^2 and this is the reason why they are not listed in these tables.

Third, if we include conjunctions in the BOS 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 term/word bigrams, which are not included by STK.

Next, the lower accuracy provided by STK_e^2 suggests that syntactic models can improve BOS although too many (possibly incorrect) syntactic features (generated by STK_e) make the model unstable. This consideration leads us to experiment with the model LIN $\times$ STK and LIN $\times$ 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 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 6.2, 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).

Moreover, we experimented with very advanced kernels built on top of feature pair spaces as shown in Table 6.3. 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

Table 6.3: Advanced kernel combination accuracies for GEO dataset

Advanced Kernel Combination	**Accuracy ± Std. Dev.**
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
$(1+LIN^2)^2+STK^2$	**75.6±8.3**
$(1+POLY^2)^2+STK^2$	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

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 improve 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, in order 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 6.2 shows that surprisingly, complex kernels are not only more accurate but also more stable, i.e. their accuracy grows smoothly according to the increase of training data.

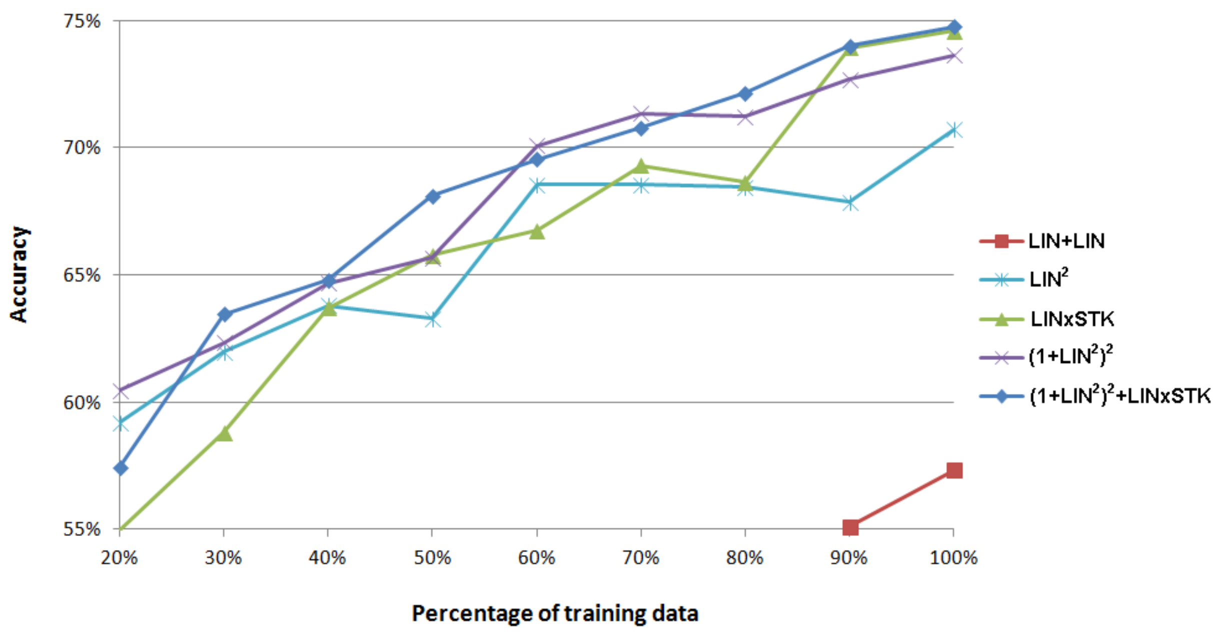


Figure 6.2: Learning curves for GeoQueries corpora

6.2.2 Evaluation on Rest dataset

The results of the experiment on the second dataset are reported on tables Table 6.4 and Table 6.5.

The baseline model, i.e. LIN + LIN, provides an accuracy of only 20.9%, which compared with the 37.1% of linear feature pair model, i.e. LIN^2, confirms that product combination of kernels is more accurate than their sum. Including conjunctions in the BOS representation, i.e. using POLY^2, we obtain a very high result. Moreover if we combine stems of the question with syntactic SQL subtrees, using POLY×STK, we outperform both POLY^2 and STK^2 by 4 points at a 95% confidence level. This confirms that it is better to use stems in the representation of the first member of the pairs and syntactic parse trees in the second member. Nevertheless, STK performs worse when taking into account also leafs (STK_e), since using terms is not relieabe as shown by the low BOS accuracy on this dataset.

Given that n-grams based text representation technique has shown to

Table 6.4: Kernel combination accuracies for Rest dataset

Kernel Combination	**Accuracy ± Std. Dev.**
LIN + LIN	20.9±11.9
LIN × LIN	37.1±16.2
POLY × POLY	**74.5±14.0**
STK × STK	71.8±10.8
$\text{STK}_e \times \text{STK}_e$	62.5±11.6
$\text{SK}_1 \times \text{SK}_1$	38.8±13.8
$\text{SK}_3 \times \text{SK}_3$	67.4±11.1
LIN × STK	79.1±11.5
$\text{LIN} \times \text{STK}_e$	77.2±12.8
POLY × STK	**82.3±11.8**
$\text{POLY} \times \text{STK}_e$	78.0±12.2
$\text{SK}_1 \times \text{STK}$	80.5±13.2
$\text{SK}_1 \times \text{STK}_e$	77.9±11.8
$\text{SK}_3 \times \text{STK}$	**81.7±13.3**
$\text{SK}_3 \times \text{STK}_e$	78.5±11.5

Table 6.5: Advanced kernel combination accuracies for REST dataset

Advanced Kernels Combination	**Accuracy ± Std. Dev.**
$(1+\text{LIN}^2)^2$	52.5±10.2
$(1+\text{POLY}^2)^2$	74.5±14.0
$(1+\text{STK})^2$	71.8±10.8
$(1+\text{STK}_e)^2$	62.5±11.6
$(1+\text{SK}_1)^2$	52.5±10.2
$(1+\text{SK}_3)^2$	69.8±10.0
$(1+\text{POLY}\times\text{STK})^2$	**84.7±11.5**
$\text{STK}^2+\text{POLY}^2$	78.6±11.9
$(1+\text{POLY}^2)^2+\text{POLY}\times\text{STK}$	**78.1±13.8**
$\text{POLY}\times\text{STK}+\text{STK}^2+\text{POLY}^2$	78.6±11.9

outperform bag-of-words approaches Lodhi et al. [2000], we experimented also using String Kernel (SK). Results confirm that using 3-grams (SK_3^2) to represent questions is a better choice than the BOS representation (SK_1^2 or LIN^2). Nevertheless, including conjunctions in the BOS representation, it performs even better.

We also experimented with advanced kernel combinations. Results are listed in table Table 6.5. Applying the polynomial kernel on top of polynomial feature pair spaces, i.e. $(1+\text{POLY}^2)^2$, we obtain a very high result, i.e. 75.4%. We found that the advanced kernel combination $(1+\text{STK}\times\text{POLY})^2$ outperforms[4] the best kernel combination POLY×STK.

Figure 6.3 illustrates the learning curve of the best kernels, i.e. the kernel combination is POLY×STK and the advanced kernel STK×POLY+ $(1+\text{POLY}^2)^2$, along with the baseline. The figure shows that the best kernel

[4]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 (about 1.8%). Considering also that we used 10 folds, it is easy to verify that the first is better than the second at 80% of confidence limit.

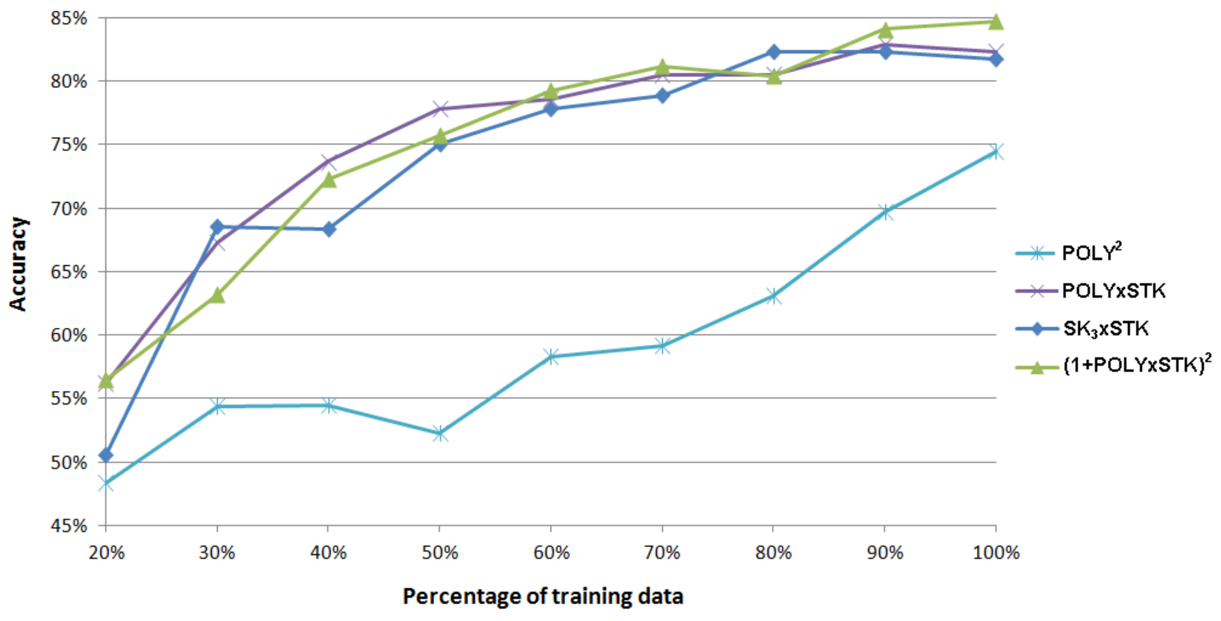


Figure 6.3: Learning curves for RestQueries corpora

including the syntanctic information is superior to the very accurate and rich kernels based on only BOS.

6.3 Related Work

In this section we discuss some NLIDBs that have been tested on GeoQueries or RestQueries datasets. Many systems have been evaluated in terms of precision and recall, whereas our performance is evaluated instead in terms of accuracy, intended as the percentage of all questions for which correct query were retrieved. Since we want to compare our results to others, we represented their results by means of accuracy.

However, even if we can compare only systems that were evaluated on the same dataset and similar experimental set-up, we report also their evaluation on GeoQueries880 to show that, the smallest is the training corpus, the more challenging is the learning.

CatchPhrase [Minock et al., 2008] is an authoring system that has been evaluated on GeoQueries250. In particular, two students were asked to author the system to cover 100 of the 250 questions each. Then the remaining questions were split in 2 test sets and translated by the system first into logical queries in tuple calculus representation and then into SQL queries. The average accuracy was 69%.

Krisp [Kate and Mooney, 2006] adopts a machine learning approach to induce a semantic grammar from a corpus of correct pairs of questions and queries. The reported experiments using standard 10-fold cross validation show an accuracy of 70% and of 75% on GeoQueries880 and GeoQueries250, respectively. Another evaluation of this system is provided by Popescu et al. [2003]. According to the authors results on GeoQueries are approximatively 78% recall and 94% precision, for an f-measure of 85%, whereas on RestQueries both precision and recall are 97%. However authors did not publish the correct number of geographical queries nor experimental-setup information.

In Precise [Popescu et al., 2003], the derivation of semantic interpretation of ambiguous phrases is reduced to a graph matching problem. Authors claim to achieve 100% precision on a subset of questions while rejecting semantically intractable questions for a final recall of 77.5% and the 95% for GeoQueries880 [5]. and RestQueries respectively.

In particular they consider a generated query as correct if it is the same of the one manually translated by an expert. In particular the expected SQL query (one with equivalent result set) is correct it is produced in the top K queries (being K a small constant) obtained by removing *unlikely* candidate queries. However we used their output as training example source and discovered many inconsistencies in the translation. In addition,

[5]It is not clear which dataset they used. They claim that each database has been tested on a set of *several hundred* English questions. They cite Tang and Mooney [2001], that actually refers to 800 queries, while the evaluation in Minock et al. [2008] suggest that they use the 250 subset. However this is not recall1.

Precise outputs a set of queries among which one correctly corresponds to an ambiguous question, while, in contrast others always retrieve at most one possibly correct query.

The performance of the above mentioned systems were originally measured according to different definitions of precision and recall since they refuse to generate a correct answer in particular output conditions. In contrast our approach allows for always having one answer, therefore it can be measured with the more appropriate accuracy measure. Using the accuracy we note that our approach is comparable to Krisp obtaining the same measure, i.e. 76% (our result) vs 75% (Krisp), on GeoQueries250. Regarding the comparison with Precise, it should be noted that we found several errors in the SQL testset in Popescu et al. [2003] (many of them do not return the correct values and other were syntactically incorrect), so we cannot provide a reliable interpretation of their results.

Moreover, our performance are also higher than the one achieved by CatchPhrase.

There exist other systems [Zettlemoyer and Collins, 2005; Wong and Mooney, 2006; Tang and Mooney, 2001; Ge and Mooney, 2005] that were tested on GeoQueries880 with different experimental-setup, so results are not directly comparable. However we perform similarly to Krisp, that compares favourably with them.

Recently, two systems, DCS [Liang et al., 2011] and SemResp [Clarke et al., 2010] have been proposed and evaluated on a subset of GeoQueries880 consisting of 250 randomly selected sentences for training and 250 for testing. According to authors, the inference problem is less constrained than previous approaches thus limiting the training data to 250 examples is due to scalability issues. They also prune the search space by limiting the number of logical symbol candidates per word (on average 13 logical symbols per word). The basic and unsupervised system SemResp achieves an accuracy of 73.2% whereas an extension of this approach involving supervision and annotated logical form shows an accuracy of 80.4%. Similarly, DCS achieves an accuracy of 78.9% with minimal supervision, while relying on prototype triggers the system DCS+ is 87.2% accurate. However, their experimental-setup is much different and even if they seem to perform better, with respect to these systems, our system do not require supervision or restrictions on the training dataset.

In addition since our approach can exploit database query logs to find all possible positive and negative training pairs our system is more portable and robust. In fact Mooney's group uses database and queries developed in Prolog, but real-word application use the widespread language SQL and it is not straightforward to map queries on one language in another (otherwise Popescu et al. could have automatically translated the datasets instead of having an expert to manually generate it).

As a last remark, recent work tends to avoid human supervision and the costly annotation of databases. For example, the approach described in Lu et al. [2008] do not rely on annotation and shows Precision of 91.5% and a Recall of 72.8%, for an f-measure of 81.1% in the GeoQueries250 dataset. Since this result is obtained applying re-ranking and the same approach applied to the larger GeoQueries880 corpus leads to an improvement in f-measure of 85.2%, we stopped experimenting with GeoQueries250 and classification algorithms and moved to the next step. As discussed in Chapter 5 and then evaluated in the following one, we will consider larger datasets to exploit reranking abilities of the engineered kernels designed so far.

With respect to the RestQueries dataset the comparative evaluation is more complex, since in our opinion Precise performance is corrupted by the wrong manual translations they use to evaluate correctness. Even if it is reasonable to believe that they real accuracy on this dataset is approximatively 85%, we can't directly compare with this result since the experimental test sets are different. However, we believe that this dataset do not reflect the complexity of a real-word applications since its structure is trivial, so we stopped experimenting with it.

It is worth noting that this approach, in contrast with previous generative approaches, retrieves the best matching query among the given set of all possible queries. One could argue that we can't find a correct answer to a given unseen NL question if the SQL query was not present in the initial dataset. While we still believe that relying on query logs is not so restrictive, since they represent frequent and required queries asked to DBs, we also implemented a generative model whose evaluation is presented the following chapter.

Chapter 7

Experimental Evaluation on Reranking Mapping

In the previous chapter we have discussed our preliminary experiments and shown the efficiency of Structured Kernels over question answering. Thanks to our classification approach we are able to retrieve the best matching query among a given set of all possible queries. However, we can't find a correct answer to a given unseen NL question if the SQL query is not present in the initial dataset. Indeed, we may not always rely on query logs even if they should reliably represent frequent asked queries to a database.

Moreover, in Chapter 3 we have illustrated that recent system tend to avoid the need of relying on an expensively annotated dataset, reducing the mapping problem to a simpler and less constrained one: deriving a mapping from natural language questions to machine readable statements by only exploiting the final real-world answers.

For these reasons we evaluated our generative approach enriched by preference reranking in order to show that we can perform semantic parsing exploiting only syntactic similarity, without the need of grammars, lexicons and annotation.

Starting from a corpus of natural language questions $\{x_1, x_2, \dots, x_m\}$ paired with their answers $\{a_1, a_2, \dots, a_m\}$, we generate for each question x_i a set of SQL queries $\{y_i^1, y_i^2, \dots, y_i^k\}$. Given that this set is ordered according to a matching weight, we report generative results of the task of selecting the top scored pair as the most correct. However, at training time we know which y_i is the correct one, i.e. the one whose result set is equivalent to the given answer a. Indicating y_1^i as the correct candidate, we use a preference reranker to learn a binary classifier by creating the pairs $\langle y_i^1, y_i^2\rangle, \dots, \langle y_i^1, y_i^k\rangle$ as positive training instances and $\langle y_i^2, y_i^1\rangle, \dots, \langle y_i^k, y_i^1\rangle$ as negative ones.

Indeed, given an unseen question n we generated a set of possibly mapping SQL queries $\mathcal{S}$. Then we re-ranked the set of pairs $P(n) = \{\langle n, s\rangle : s \in \mathcal{S}\}$ and used the top-ranked element of $P(n)$ to select the candidate $\langle n, s\rangle$ such that the execution of s retrieves the answer a to question n. At classification time, since the correct candidate is not known and pairs are not formed, we apply the standard one-versus-all binarization method.

7.1 Setup

After developing and integrating our generative approach (Chapter 5), we ran several experiments to evaluate the accuracy of our approach for automatic generation and selection of correct SQL queries from questions. We only experimented with the GeoQueries880 dataset, since the other dataset was too simple an we didn't expect to obtain large improvements.

To generate the set of possible SQL queries we applied our algorithm described in Chapter 5 to the GeoQueries corpus. We considered the full GeoQuery annotation (Geo880) but we used the subset of 700 pairs (henceforth Geo700) since they had been translated by Popescu et al. [2003] from Prolog data to SQL queries.

Additionally, to compare with latest systems Clarke et al. [2010]; Liang et al. [2011], which used a subset of 500 pairs, hereafter GEO500, we annotated the remaining 180 pairs as they were included in GEO500. The latter was randomly split by Clarke et al. [2010] in 250 pairs for training and 250 pairs for testing. The data is slightly *easier* since the number of logical symbols per word are limited to an average of 13 logical symbols. It is worth noting that even if we manually annotated missing questions with their answering SQL queries, we only used them for extracting the answer from the database and evaluate the pair correctness (so we do not really use the SQL queries).

However, pairs that were already annotated in the original GEO700 contained some errors and inconsistencies in SQL queries that we fixed, 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.

To learn the reranker, we used SVM-Light-TK[1], which extends the SVM-Light optimizer [Joachims, 1999] with tree kernels. i.e. Syntactic Tree Kernel (STK) as described in Section 2.3. We modelled the same combinations that in our pilot experiment has shown to be more effective. We used the default parameters, i.e. the cost and trade-off parameters = 1 (for normalized kernels) and $\lambda = 0.4$ (see Sec. 2.3.2).

To evaluate the reranking results of on GEO700, we applied standard 10-fold cross validation and measure the average recall and the standard deviation of selecting the top ranked query as the correct one (i.e. that retrieves the same result as the annotated one). With respect to GEO500 we kept the original split of 250 training samples and 250 test samples.

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

7.2 Generative Results

We carried out the first experiment on GEO700. Our algorithm could generate a correct SQL query in the first 25 candidates for 95.3% of the cases but could not answer to 33 questions. This was due to (i) empty clauses set $\mathcal{S}$ and/or $\mathcal{W}$, for example, "*How many square kilometers in the US?*" does not contain useful stems; and (ii) mismatching in nested queries, for example, "*Count the states which have elevations lower than what Alabama has*" contains an implicit reference to the missing information. In addition, there were incomplete questions like "*Which states does the Colorado?*" from which we retrieved an incomplete dependency set.

When our algorithm can generate an ordered list of possible queries, the top query is correct for 82% of the cases. Additionally, the correct answer is contained in the first 10 candidates for 99% of the cases (excluding the 33 questions above).

This can be observed in Figure 7.1, which plots the Recall (of the correct question) curve of the generative approach, i.e., the baseline. As pointed out in the graphic, the correct query is found among the first three in 93% of the cases. In fact with our generative algorithm we are able to assign a high score to the correct pairings, but the pairings between a given question and the correct generated query is not always the highest scoring option. The reason is that we are very flexible when generating all possible queries in order to reach an higher recall. However this leads to lower precision and the need for improving it.

We obtain similar results with the GEO500 subset: we fail to generate an answer in 18 out of the 250 pairs of the test set. We also found that the correct answer is 78% of the times in the top position while it can be retrieved among the first top seven in 98% of the cases.

While our ranking based on heuristic weights is rather robust and pro-

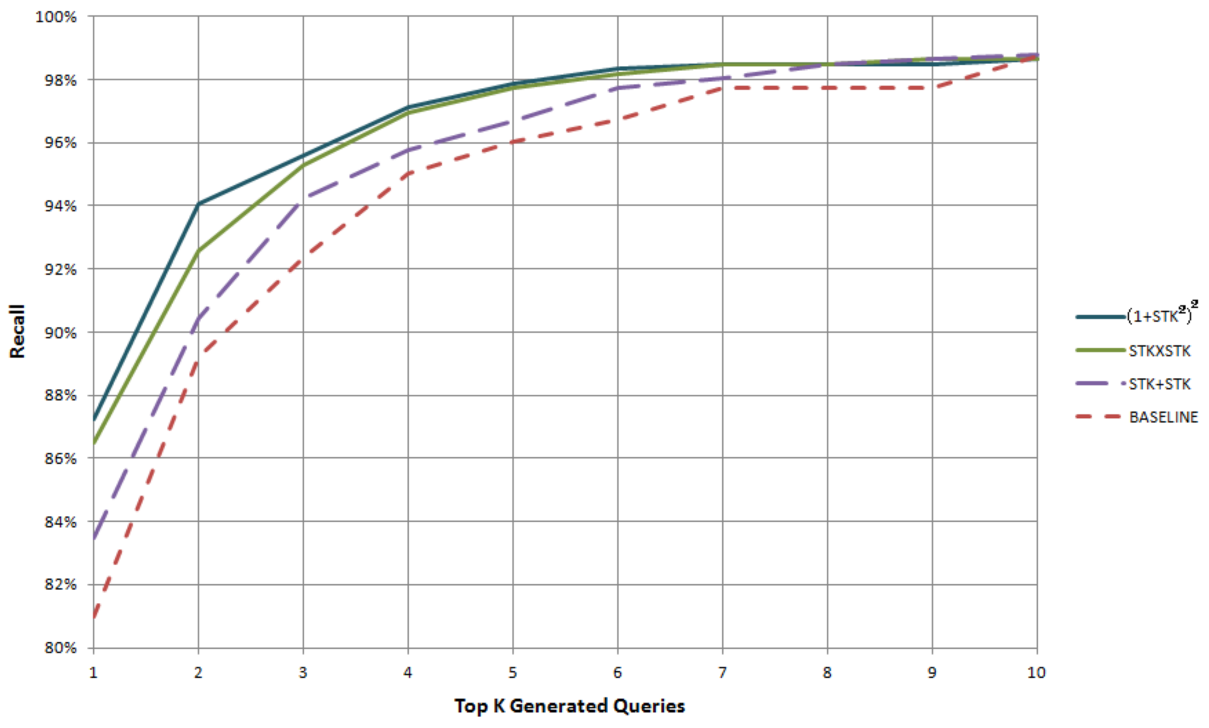


Figure 7.1: Recall of the correct answer within different top k positions of the system rank for GEO700 dataset

duce high recall, the accuracy on the top candidate can then be promisingly increased with reranking.

7.3 Reranking Results

To improve the accuracy of our generative model, we used a preference reranking approach (introduced in Section 5.3).

Figure 7.1 shows the plots of several systems' Recall (of at least one correct answer) according to different k-top candidate answers. In addition to the generative model (Baseline), the graph shows the accuracy of different rerankers applied to the Baseline: these use the following kernels: $\mathrm{STK}_n+\mathrm{STK}_s$, $\mathrm{STK}_n\times\mathrm{STK}_s$ and $(1+\mathrm{STK}_n\times\mathrm{STK}_s)^2$ as the K factor in Eq. 5.5, where n indicates kernels for questions and s for queries[2], re-

[2]In the following we omit such indices, recalling that the first kernel is always applied to the question

spectively. We note that reranking remarkably improves the results, e.g., $(1+\text{STK}_n \times \text{STK}_s)^2$ retrieves the correct answers 94% of times by only using the first two answers.

To assess our findings, we applied standard 10-fold cross validation and measured the average Recall in selecting a correct query for each question. The results for different models on GEO700 are reported in Table 7.1. The first column lists the 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 Recall of at least 1 correct answer in the top k positions (more precisely the average of Recall@k over 10 folds ± Std. Dev).

Additionally, we evaluated the same kernels for reranking pairs generated from the GEO500 dataset. Figure 7.2 shows the Recall curve of STK+STK, STK×STK and $(1+\text{STK}\times\text{STK})^2$ along with the baseline results already introduced in section 7.2.

For this benchmark the trade-off between different kernel is not as clear as the one evidenced in Figure 7.1. The reason could be that GEO700 have been evaluated using 10-fold cross validation, while GEO500 have been split into 250 training pairs and 250 test pairs. While STK×STK

parse trees whereas the second kernel is applied to the trees of query derivations.

Table 7.1: 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
BOW × STK	86.7±4.1	92.1±3.2	95.6±2.5	97.1±1.4	97.6±1.2
$(1+\text{STK}^2)^2$	**87.2±3.9**	**94.1±3.4**	95.6±2.7	97.1±1.9	97.9±1.4

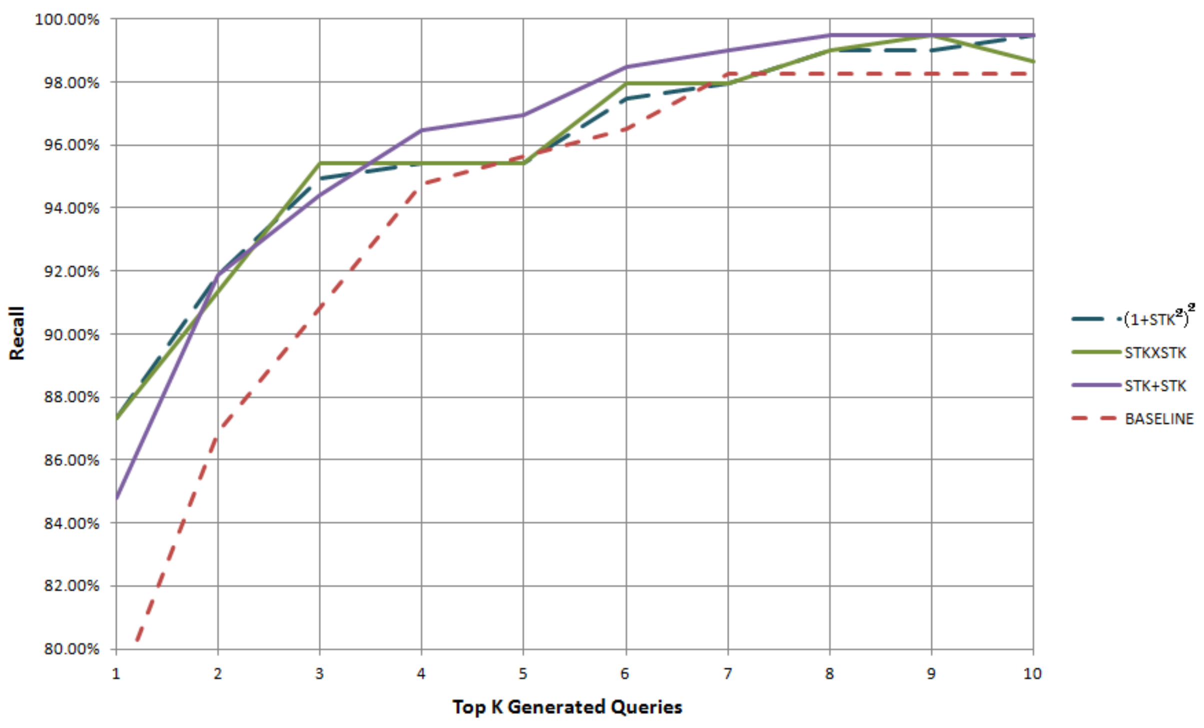


Figure 7.2: Recall of the correct answer within different top k positions of the system rank for GEO500 dataset

outperforms STK+STK in the first position of the re-ranked list of pairs, expanding our search space performances of advanced kernel combination STK×STK and $(1+\text{STK}\times\text{STK})^2$ do not improve consistently. However we can see that whatever k we used, reranking performances on the top k search space outperform the baseline of at least 2 points.

Using STK+STK we obtain a Recall of 84.77%, while if we exploit the product STK×STK, we achieve 87.31%. These results are rather exciting since they compare favorably with the state-of-the-art.

Finally, we report the learning curve of one basic reranker STK×STK in Figure 7.3. The plot shows that, as soon as a reasonable amount of training data is used (i.e., 25% of 9 folds of 700 questions – one fold is used for testing), the reranker improves on the baseline.

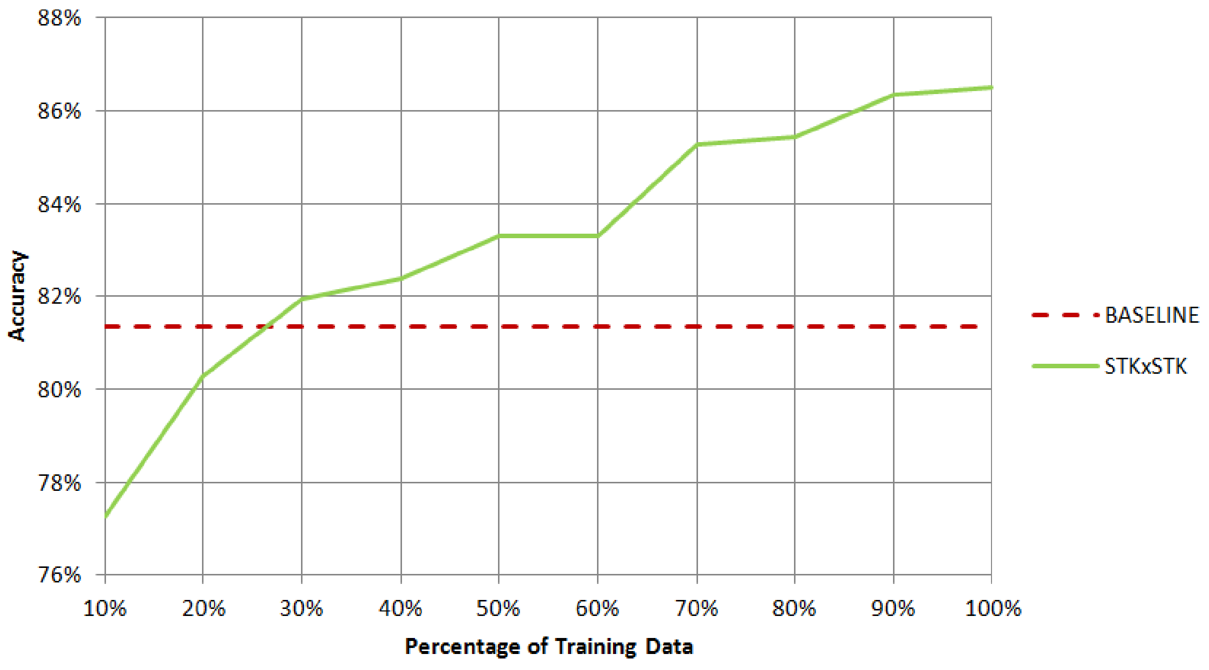


Figure 7.3: Learning curve

7.4 Related Work

In this section we compare our result with similar learning approaches, ignoring a substantial piece of related research that is not directly comparable to our approach.

Early work on semantic parsing [Tang and Mooney, 2001] required either the definition of rules and constrains in an ILP framework or manually produced meaning representations [Ge and Mooney, 2005; Wong and Mooney, 2006], which are costly to produce. Additionally, authoring systems where developed by specifying a semantic grammar [Minock et al., 2008], which requires large effort of human experts.

Table 7.2 shows the f-measure of some state-of-the-art systems to which we compare. Such systems were tested on GeoQueries, according to different experimental setups and data versions. The first half of the table reports on systems exploiting the annotated logical form (deriving the answer) whereas the last five rows show the f-measure of systems only exploiting the training pairs, questions and answers.

PRECISE [Popescu et al., 2003] is the only system evaluated on GEO700 in terms of correct SQL queries. The value reported in the table refers to the correctness of answering questions if the expected SQL query (i.e., one with equivalent result) is produced by one of the top k queries[3]. Also our system can provide multiple answers and if we select the first k candidates, we highly increase the Recall (within the first 2 we have an F1 of 90%). Note that [Popescu et al., 2003] needed to rephrase some queries to achieve their result.

Another system similar to ours, which applies SVMs and string kernel is KRISP Kate and Mooney [2006]. The major difference is that it requires meaning representations (following a user-defined MR grammar), instead of SQL queries.

Recent work has also explored learning to map sentences to meaning representations suitable for applying lambda-calculus [Zettlemoyer and Collins, 2005; Wong and Mooney, 2007]. This kind of system require a large amount of supervision. In particular, the system in Zettlemoyer and Collins [2005] 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%. Thus, our system trades-off 2 points of accuracy for avoiding large work for handcrafting resources, i.e., the semantic trees manually annotated for each question. Moreover, our system is much simpler to implement.

A more recent work [Lu et al., 2008] does not rely on annotation and shows a Precision of 89.3% and a Recall of 81.5%, for an f-measure of 85.2%. Their generative model coupled with a discriminative reranking technique (MODELIII+R) is conceptually similar to our approach.

[3]Being k a small constant, not better defined by the authors.

Table 7.2: Comparison between state of the art learning systems.

System Name	**Human Supervision**	**Geo500**	**Geo700**	**Geo880**
PRECISE	Rephrase question	-	87%	-
KRISP	Specify the grammar	-	-	81%
MODELIII+R	-	-	-	85%
SEMRESP	Define a Lexicon	80%	-	-
UBL	Specify a CCG Lexicon	-	-	89%
MANALA/SQL	-	-	85%	-
SEMRESP	Define a Lexicon	73%	-	-
UBL	Specify a CCG Lexicon	-	-	85%
DCS	Define Lexical Triggers	79%	-	89%
DCS$^+$	Define an Augmented Lexicon	87%	-	91%
MANALA/SQL*	-	**87%**	85%	-

It it worth noting that even though we used a dataset where each natural language question is annotated (paired) with an SQL query, our generative and reranking models[4] do no rely on this annotation since we exploit the paired SQL query only to retrieve the answer. For this reason our approach is more flexible and adaptable since we just need natural language questions to be paired with its answer instead of its corresponding logical form.

SEMRESP [Clarke et al., 2010] also learn a semantic parser from question-answer pairs. They achieve the highest accuracy when tested on annotated logical forms whereas when tested on answers their accuracy is lower (80% vs. 73% in f-measure). In contrast, our system, evaluated on answers, outperforms their best system in all setting, e.g., (85% vs. 80%).

[4]We refer to this system as MANALA/SQL*, since it is an extension of the original MANALA/SQL that performed semantic mapping exploiting (NL,SQL) trees. We report also the f-measure of this pilot model in producing the correctness annotated SQL for sake of comparison, even if it has been discussed in the previous Chapter.

Another system evaluated both with logical forms and with answers is UBL [Kwiatkowski et al., 2010]. Starting from a restricted set of lexical items and CCG combinatory rules, it is able to learn new lexical entries and achieves the best performance with Geo880 when trained with logical forms.

In contrast, the best performing system that does not exploit the annotation of the Geo880 is DCS [Liang et al., 2011]. It involves a lower level of supervision with respect to UBL since it requires having a set of lexical triggers (enriched by prototype triggers in DCS$^+$) that is a much weaker requirement than having a CCG lexicon.

However, the comparison of the systems above with ours on Geo500 shows that ours largely outperforms DCS (87% vs. 78% in f-measure). Our system performs comparably to the version enriched with prototype triggers, DCS$^+$, even though we do not exploit such manual resources.

In summary, our system is competitive with other supervised parsers as it: (i) only relies on the answers, i.e., without using any annotated meaning representations (e.g. Prolog data, MR, Lambda calculus, SQL queries); and (ii) requires much less supervision since there is no need to build semantic representation. Our manual intervention only regards the definition of few synonym relations, i.e., *border* and *next to* as synonyms for *traverse*, since there are not such relations in Wordnet. The rest of the lexicon is induced by the database metadata or obtained exploiting Wordnet.

Finally, our system is competitive with the state-of-the-art defined in Lu et al. [2008]. In fact, considering that the performance of ModelIII+R on Geo880 has been tested using and independent test set of 280 pairs and a 600 training pairs, we can conclude that we perform comparably with this unsupervised system. Even though we evaluated our results using 10-fold cross validation on Geo700, our training data contains a similar number of pairs (i.e. 670) and we obtain the same results (85%). This is not surprising since we use a very similar approach, i.e., a generative model coupled with discriminative reranking. However, while the system above learns a parser on meaning representations, we only need natural language questions their answers (of course targeting a DB).

Chapter 8

Conclusions

In this research, we propose two novel models for mapping NL into SQL based on robust machine learning algorithms, e.g. Support Vector Machines (SVMs), and effective approaches for structural representation, e.g. Sequence and Tree Kernels Lodhi et al. [2000]; Collins and Duffy [2002]; Vishwanathan and Smola [2003]; Moschitti [2006]. More specifically, since computational linguistics research Winograd [1972] has shown that such mapping problem cannot be addressed with a full semantic approach, our solution has to rely on shallow and statistical methods.

We approach the problem of deriving a shared semantics between natural language and programming language by automatically learning models based on a syntactic representation of the training examples that we use both for classification and re-ranking purposes. In our experiments we consider pairs of NL questions and SQL queries as training examples.

We automatically annotated the pairs by means of our algorithm starting from a given initial annotation. In particular we experimented with the annotation available in GeoQueries250 and RestQueries corpora. Our semisupervised algorithm allowed for adding new positive pairs, creating negative example set and also fixing some errors [1].

[1]Our datasets are be publicly available so that other system can compare with our benchmark corpora. `http://projects.disi.unitn.it/iKernels/`

To represent syntactic/semantic relationships expressed by training pairs, we encode such pairs in SVM by means of kernel functions. We designed innovative combinations between different kernels for structured data applied to pairs of objects, that, to the best of our knowledge, represent a novel approach to describe relational semantics between NL and SQL languages.

Given the good results in classification, we have approached question answering targeting database information by automatically generating SQL queries in response of the posed question. The generative model exploits grammatical dependencies and metadata and can deal with complex natural language questions, containing subordinates, conjunction and negation and nested SQL queries.

Additionally, we firstly experimented with a supervised preference reranking kernel, which is able to boost the accuracy of our generative model (which is instead unsupervised). The underlying idea that we propose for building and combining clauses sets is novel.

It should be noted that state-of-the-art systems depend on corpora specifically annotated for the tasks implemented by these systems. This is a major limitation in their success, usability and portability. These corpora are often very large since systems perform better with larger training data. In fact, finding a method for effectively learning from limited amounts of data is challenging. In this respect, our approach is very valuable as we do not need any annotated training set and tailored grammar or lexicon, since we use database metadata and grammar dependencies in the NL questions, where the answer can be just a NL text.

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 experiments, we have shown that our generative model, when coupled with a tree-kernel based reranking, achieves state-of-the-art performance when tested on two publicly available corpora.

The main contributions of this study are: (i) we generated a dataset of positive and negative pairs of NL questions and SQL queries represented by means of their syntactic trees; (ii) we learned a classifier for detecting correct and incorrect pairs of questions and queries using kernel methods along with SVMs; (iii) we implemented an SQL query generator that creates a weighted list of candidate SQL queries where the top scored is the correct one with a fairly high precision and (iv) we exploited an SVM ranker trained with structural kernels to reorder the candidate list to improve the generation performance.

Finally, we have also shown that kernel product can be effective and that syntax is important to map into programming languages but also for generating new queries. Nevertheless we have shown that automatic generation of semantic grammars is viable.

8.1 Future Work

In the future we plan to experiment with datasets in different domains. One example could be the no longer current airlines reservation system included in the ATIS distribution [Pallett et al., 1994].

Moreover, given that current challenges in Semantic Web tackle similar problem Cimiano and Minock [2009] (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 (e.g. testing our model on the MusicBrainz corpus).

Some of our experiments have shown that the major limitation of our approach is dealing with questions whose answers are empty (e.g. "*State bordering Alaska?*"). The reason is that we can't recognize if an empty answer is the outcome of a correct SQL query generation or the result of something wrong in the process. While other systems overtake this problem (e.g. for the JOBS corpus, where almost half of the answers are empty, Liang et al. [2011] randomly generated values for each empty field) a fair solution could be the development of a *SQL meaning checker* that recognizes when an empty answer is a plausible result or not.

Last but not least, since our learning algorithm is language independent, we plan to extend our generative model to other languages (the GEOQUERIES250 questions are also available in other languages).

Our system could be largely improved augmenting the SQL grammar used by our generative algorithm, taking into account GROUPBY, ORDER, and LIMIT clauses to handle more difficult questions, like for example "*What it the total population of the ten largest capitals in the US?*".

Moreover, this approach could be easily extended to allow for cross-domain questions, as long as IS embeds shared metadata between multiple databases. For example, if we have both GEOQUERY and RESTQUERY data in the same database systems, we could find an answer for cross-domain questions like "*Which is the best Italian restaurant of the capital of California?*".

We are also interested in investigating ways to apply the classification approach to the inverse task. This could be particularity useful in real world database systems where students try to write their own SQL queries. They would benefit from a feedback from an automatic system that enriches the query result with its possible interpretation in human language.

In addition we would like to extend this research by focusing on advanced shallow semantic approaches such as predicate argument structures Giuglea and Moschitti [2006].

Bibliography

Androutsopoulos, I.; Ritchie, G., and Thanisch, P. Masque/sql– an efficient and portable natural language query interface for relational databases. In *Proceedings of the 6th International Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems*, pages 327–330. Gordon and Breach Publishers Inc, 1993.

Balmin, Andrey; Hristidis, Vagelis, and Papakonstantinou, Yannis. Objectrank: Authority-based keyword search in databases. In Nascimento, Mario A.; Özsu, M. Tamer; Kossmann, Donald; Miller, Renée J.; Blakeley, José A., and Schiefer, K. Bernhard, editors, *VLDB*, pages 564–575. Morgan Kaufmann, 2004. ISBN 0-12-088469-0.

Barbosa, Juan Javier González; Rangel, Rodolfo A. Pazos; C., I. Cristina Cruz; H., Héctor J. Fraire; de L., Santos Aguilar, and Pérez, Joaquín. Issues in translating from natural language to sql in a domain-independent natural language interface to databases. In Gelbukh, Alexander F. and García, Carlos A. Reyes, editors, *MICAI*, volume 4293 of *Lecture Notes in Computer Science*, pages 922–931. Springer, 2006. ISBN 3-540-49026-4.

Basili, R.; Moschitti, A., and Pazienza, M.T. A text classifier based on linguistic processing. In *Proceedings of IJCAI 99, Machine Learning for Information Filtering*, 1999.

Chali, Yllias and Joty, Shafiq. Improving the performance of the random walk model for answering complex questions. In *Proceedings of ACL-08: HLT, Short Papers*, pages 9–12, Columbus, Ohio, 2008. URL `http://www.aclweb.org/anthology/P/P08/P08-2003`.

Chandra, Yohan and Mihalcea, Rada. Natural language interfaces to databases. University of North Texas, 2006. Thesis (M.S.).

Charniak, E. A maximum-entropy-inspired parser. In *Proceedings of NAACL'00*, 2000.

Chaudhuri, S.; Das, G., and Narasayya, V. Dbexplorer: A system for keyword search over relational databases. In *Proceedings of the 18th Int. Conf. on Data Engineering, San Jose, USA*, 2002.

Cimiano, Philipp and Minock, Michael. Natural language interfaces: What is the problem? - a data-driven quantitative analysis. In *NLDB*, pages 192–206, 2009.

Clarke, J.; Goldwasser, D.; Chang, M., and Roth, D. Driving semantic parsing from the world's response. In *CoNLL*, 7 2010. URL `http://cogcomp.cs.illinois.edu/papers/CGCR10.pdf`.

Collins, M. and Duffy, N. New ranking algorithms for parsing and tagging: Kernels over discrete structures, and the voted perceptron. In *Proceedings of ACL'02*, 2002.

Copestake, A. and Sparck-Jones, K. Natural language interfaces to databases. *Natural language interfaces to databases. The Knowledge Engineering Review, Special Issue on the Application of Natural Language Processing Techniques*, 1990.

Culotta, Aron and Sorensen, Jeffrey. Dependency Tree Kernels for Relation Extraction. In *ACL04*, pages 423–429, Barcelona, Spain, 2004.

Cumby, Chad and Roth, Dan. Kernel Methods for Relational Learning. In *Proceedings of ICML 2003*, pages 107–114, Washington, DC, USA, 2003.

Somers H.L.Dale R., Moisl H., editor. *Database Interfaces*, chapter 9, pages 209–240. Marcel Dekker Inc., 2000. ISBN 0824790006.

Dittenbach M., Merkl W. and Berger, H. A natural language query interface for tourism information. In *Proceedings of the Int'l Conference on Information and Communication Technologies in Tourism (ENTER'03)*, pages 152–162. Springer-Verlag, 2003.

Filipe, Porfirio P. and Mamede, Nuno J. Databases and natural language interfaces. In *IN JISBD 2000*, pages 321–332, 2000.

Garcia-Molina, Hector; Ullman, Jeffrey D., and Widom, Jennifer. *Database Systems: The Complete Book*. Prentice Hall Press, Upper Saddle River, NJ, USA, 2 edition, 2008. ISBN 9780131873254.

Ge, Ruifang and Mooney, Raymond. A statistical semantic parser that integrates syntax and semantics. In *Proceedings of the Ninth Conference on Computational Natural Language Learning (CoNLL-2005)*, pages 9–16, Ann Arbor, Michigan, June 2005. Association for Computational Linguistics. URL `http://www.aclweb.org/anthology/W/W05/W05-0602`.

Giordani, Alessandra and Moschitti, Alessandro. Syntactic structural kernels for natural language interfaces to databases. In *Proceedings of the European Conference on Machine Learning and Knowledge Discovery in Databases: Part I*, ECML PKDD '09, pages 391–406, Berlin, Heidelberg, 2009. Springer-Verlag. ISBN 978-3-642-04179-2. doi: 10.1007/978-3-642-04180-8_43. URL `http://dx.doi.org/10.1007/978-3-642-04180-8_43`.

Giuglea, Ana-Maria and Moschitti, Alessandro. Semantic role labeling via framenet, verbnet and propbank. In *Proceedings of ACL 2006*, Sydney, Australia, 2006.

Hendrix, Gary G. Lifer: A natural language interface facility. In *Berkeley Workshop*, pages 196–, 1977.

Hendrix, Gary G. Bringing natural language processing to the microcomputer market: The story of q&a. In *Proceedings of the 24th Annual Meeting of the Association for Computational Linguistics*, pages 2–2, New York, New York, USA, July 1986. Association for Computational Linguistics. doi: 10.3115/981131.981133. URL `http://www.aclweb.org/anthology/P86-1002`.

Hulgeri, Arvind; Bhalotia, Gaurav; Nakhe, Charuta; Chakrabarti, Soumen, and Sudarshan, S. Keeyword search in databases. *IEEE Data Eng. Bull.*, 24(3):22–32, 2001.

Joachims, T. Making large-scale SVM learning practical. In Schölkopf, B.; Burges, C., and Smola, A., editors, *Advances in Kernel Methods*, 1999.

Kate, Rohit J. and Mooney, Raymond J. Using string-kernels for learning semantic parsers. In *Proceedings of the 21st ICCL and 44th Annual Meeting of the ACL*, pages 913–920, Sydney, Australia, July 2006. Association for Computational Linguistics. doi: 10.3115/1220175.1220290. URL `http://www.aclweb.org/anthology/P06-1115`.

Kazama, Jun'ichi and Torisawa, Kentaro. Speeding up Training with Tree Kernels for Node Relation Labeling. In *Proceedings of EMNLP 2005*, pages 137–144, Toronto, Canada, 2005.

Koutrika, Georgia; Simitsis, Alkis, and Ioannidis, Yannis E. Précis: The essence of a query answer. In Liu, Ling; Reuter, Andreas; Whang, Kyu-Young, and Zhang, Jianjun, editors, *ICDE*, pages 69–78. IEEE Computer Society, 2006.

Kudo, Taku and Matsumoto, Yuji. Fast Methods for Kernel-Based Text Analysis. In Hinrichs, Erhard and Roth, Dan, editors, *Proceedings of ACL*, pages 24–31, 2003. URL `http://www.aclweb.org/anthology/P03-1004.pdf`.

Kudo, Taku; Suzuki, Jun, and Isozaki, Hideki. Boosting-based parse reranking with subtree features. In *Proceedings of ACL'05*, US, 2005. URL `http://www.aclweb.org/anthology/P/P05/P05-1024`.

Kwiatkowski, Tom; Zettlemoyer, Luke S.; Goldwater, Sharon, and Steedman, Mark. Inducing probabilistic ccg grammars from logical form with higher-order unification. In *EMNLP*, pages 1223–1233. ACL, 2010. ISBN 978-1-932432-86-2.

Li, Yunyao; Yang, Huahai, and Jagadish, H. V. Nalix: an interactive natural language interface for querying xml. In Özcan, Fatma, editor, *SIGMOD Conference*, pages 900–902. ACM, 2005. ISBN 1-59593-060-4.

Liang, P.; Jordan, M. I., and Klein, D. Learning dependency-based compositional semantics. In *Association for Computational Linguistics (ACL)*, pages 590–599, 2011.

Lodhi, Huma; Taylor, John S.; Cristianini, Nello, and Watkins, Christopher J. C. H. Text classification using string kernels. In *NIPS*, pages 563–569, 2000. URL `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.19.7025`.

Lu, Wei; Ng, Hwee Tou; Lee, Wee Sun, and Zettlemoyer, Luke S. A generative model for parsing natural language to meaning representations. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*, EMNLP '08, pages 783–792, Stroudsburg, PA, USA, 2008. Association for Computational Linguistics. URL `http://dl.acm.org/citation.cfm?id=1613715.1613815`.

Marie-Catherine de Marneffe, Bill MacCartney and Manning, Christopher D. Generating typed dependency parses from phrase structure parses. In *Proceedings LREC 2006*, 2006.

Miller, George A. WordNet: A lexical database for English. *Communications of the ACM*, 1995.

Minock, Michael; Olofsson, Peter, and Näslund, Alexander. 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.

Moschitti, A. Efficient convolution kernels for dependency and constituent syntactic trees. In *Proceedings of ECML'06*, 2006.

Moschitti, A.; Quarteroni, S.; Basili, R., and Manandhar, S. Exploiting syntactic and shallow semantic kernels for question/answer classification. In *Proceedings of ACL'07*, Prague, Czech Republic, 2007.

Moschitti, Alessandro and Quarteroni, Silvia. Kernels on linguistic structures for answer extraction. In *Proceedings of ACL-08: HLT, Short Papers*, Columbus, Ohio, 2008.

Pallett, David S.; Fiscus, Jonathan G.; Fisher, William M.; Garofolo, John S.; Lund, Bruce A., and Przybocki, Mark A. benchmark tests for the arpa spoken language program. In *In Proceedings of the Spoken Language Systems Technology Workshop*, pages 5–36. Morgan Kaufmann, 1994.

Pieraccini, Roberto; Tzoukermann, Evelyne; Gorelov, Zakhar; Gauvain, Jean-Luc; Levin, Esther; Lee, Chin-Hui, and Wilpon, Jay G. A speech understanding system based on statistical representation of semantics. In *Proceedings of the 1992 IEEE international conference on Acoustics, speech and signal processing - Volume 1*, ICASSP'92, pages 193–196, Washington, DC, USA, 1992. IEEE Computer Society. ISBN 0-7803-0532-9. URL `http://dl.acm.org/citation.cfm?id=1895550.1895604`.

Poon, Hoifung and Domingos, Pedro. Unsupervised semantic parsing. In *EMNLP*, pages 1–10. ACL, 2009. ISBN 978-1-932432-63-3. URL `http://www.cs.washington.edu/homes/hoifung/papers/poon09.pdf`.

Popescu, Ana-Maria; A Etzioni, Oren, and A Kautz, Henry. Towards a theory of natural language interfaces to databases. In *Proceedings of the 2003 International Conference on Intelligent User Interfaces*, pages 149–157, Miami, 2003. Association for Computational Linguistics.

R., Boohtra. Natural language interfaces: Comparing english language front end and english query. Virginia Commonwealth University, 2004. Thesis (M.S.).

Ruwanpura, S. Sq-hal: Natural language to sql translator. URL `http://www.csse.monash.edu.au/hons/projects/2000/Supun.Ruwanpura`.

Salton, Gerard. Recent trends in automatic information retrieval. In *SIGIR'86, Proceedings of the 9th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval, Pisa, Italy, September 8-10, 1986*, pages 1–10. ACM, 1986.

Shawe-Taylor, J. and Cristianini, N. *Kernel Methods for Pattern Analysis.* Cambridge University Press, 2004.

Shen, D. and Lapata, M. Using semantic roles to improve question answering. In *Proceedings of EMNLP-CoNLL*, 2007. URL `http://www.aclweb.org/anthology/D/D07/D07-1002`.

Shen, Libin and Joshi, Aravind 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*, pages 9–16, 2003.

Shen, Libin; Sarkar, Anoop, and Joshi, Aravind k. Using LTAG Based Features in Parse Reranking. In *EMNLP*, Sapporo, Japan, 2003.

Surdeanu, Mihai; Ciaramita, Massimiliano, and Zaragoza, Hugo. Learning to rank answers on large online QA collections. In *Proceedings of ACL-08: HLT*, Columbus, Ohio, 2008. URL `http://www.aclweb.org/anthology/P/P08/P08-1082`.

Tang, L. R. and Mooney, Raymond J. Using multiple clause constructors in inductive logic programming for semantic parsing. In *Proceedings of the 12th European Conference on Machine Learning*, pages 466–477, Freiburg, Germany, 2001. URL `http://www.cs.utexas.edu/users/ml/publication/paper.cgi?paper=cocktail-ecml-01.ps.gz`.

Toutanova, Kristina; Markova, Penka, and Manning, Christopher. The Leaf Path Projection View of Parse Trees: Exploring String Kernels for HPSG Parse Selection. In *Proceedings of EMNLP 2004*, Barcelona, Spain, 2004.

Vishwanathan, S. V. N. and Smola, Alexander J. Fast kernels for string and tree matching. In *Advances in Neural Information Processing Systems 15*, pages 569–576. MIT Press, 2003.

Waltz, David L. An english language question answering system for a large relational database. *Commun. ACM*, 21(7):526–539, July 1978. ISSN 0001-0782. doi: 10.1145/359545.359550. URL `http://doi.acm.org/10.1145/359545.359550`.

Winograd, T. *Understanding Natural Language.* Academic Press, New York, 1972.

Wong, Yuk Wah and Mooney, Raymond. Learning for semantic parsing with statistical machine translation. In *Proceedings of the Human Language Technology Conference of the NAACL, Main Conference*, pages 439–446, New York City, USA, June 2006. Association for Computational Linguistics. URL `http://www.aclweb.org/anthology/N/N06/N06-1056`.

Wong, Yuk Wah and Mooney, Raymond. Learning synchronous grammars for semantic parsing with lambda calculus. In *Proceedings of the 45th Annual Meeting of the Association of Computational Linguistics*, pages 960–967, Prague, Czech Republic, June 2007. Association for Computational Linguistics. URL `http://www.aclweb.org/anthology/P07-1121`.

Zettlemoyer, Luke S. and Collins, Michael. Learning to map sentences to logical form: Structured classification with probabilistic categorial grammars. In *UAI*, pages 658–666, 2005.

Zhang, D. and Lee, W. Question classification using support vector machines. In *Proceedings of SIGIR'03*, Toronto, Canada, 2003a. ACM. doi: http://doi.acm.org/10.1145/860435.860443.

Zhang, Dell and Lee, Wee Sun. Question classification using support vector machines. In *Proceedings of the 26th annual international ACM SIGIR conference on Research and development in informaion retrieval*, pages 26–32. ACM Press, 2003b. ISBN 1-58113-646-3. doi: http://doi.acm.org/10.1145/860435.860443.

Zhang, Min; Zhang, Jie, and Su, Jian. Exploring Syntactic Features for Relation Extraction using a Convolution tree kernel. In *Proceedings of NAACL*, pages 288–295, New York City, USA, 2006. URL `http://www.aclweb.org/anthology/N/N06/N06-1037`.