Appears in *Proceedings of the Joint SIGDAT Conference on Empirical Methods in Natural Language Processing and Very Large Corpora (EMNLP/VLC-2000),* pp.133-141, Hong Kong, October, 2000

Automated Construction of Database Interfaces: Integrating Statistical and Relational Learning for Semantic Parsing

Lappoon R. Tang and Raymond J. Mooney
Department of Computer Sciences
University of Texas at Austin
Austin, TX 78712-1188
{rupert,mooney}@cs.utexas.edu

Abstract

The development of natural language interfaces (NLI's) for databases has been a challenging problem in natural language processing (NLP) since the 1970's. The need for NLI's has become more pronounced due to the widespread access to complex databases now available through the Internet. A challenging problem for empirical NLP is the automated acquisition of NLI's from training examples. We present a method for integrating statistical and relational learning techniques for this task which exploits the strength of both approaches. Experimental results from three different domains suggest that such an approach is more robust than a previous purely logic-based approach.

1 Introduction

We use the term *semantic parsing* to refer to the process of mapping a natural language sentence to a structured meaning representation. One interesting application of semantic parsing is building natural language interfaces for online databases. The need for such applications is growing since when information is delivered through the Internet, most users do not know the underlying database access language. An example of such an interface that we have developed is shown in Figure 1.

Traditional (*rationalist*) approaches to constructing database interfaces require an expert to hand-craft an appropriate semantic parser (Woods, 1970; Hendrix et al., 1978). However, such hand-crafted parsers are time consuming to develop and suffer from problems with robustness and incompleteness even for domain specific applications. Nevertheless, very little research in empirical NLP has explored the task of automatically acquiring such interfaces from annotated training examples. The only exceptions of which we are aware are a statistical approach to mapping airline-information queries into SQL presented in (Miller et al., 1996), a probabilistic decision-tree method for the same task described in (Kuhn and De Mori, 1995), and an approach using relational learning (a.k.a. *inductive logic programming*, ILP) to learn a logic-based semantic parser described in (Zelle and Mooney, 1996).

The existing empirical systems for this task employ either a purely logical or purely statistical approach. The former uses a deterministic parser, which can suffer from some of the same robustness problems as rationalist methods. The latter constructs a probabilistic grammar, which requires supplying a sytactic parse tree as well as a semantic representation for each training sentence, and requires hand-crafting a small set of contextual features on which to condition the parameters of the model. Combining relational and statistical approaches can overcome the need to supply parse-trees and hand-crafted features while retaining the robustness of statistical parsing. The current work is based on the CHILL logic-based parser-acquisition framework (Zelle and Mooney, 1996), retaining access to the complete parse state for making decisions, but building a probabilistic relational model that allows for statistical parsing.

2 Overview of the Approach

This section reviews our overall approach using an interface developed for a U.S. Geography database (Geoquery) as a sample application (Zelle and Mooney, 1996) which is available on the Web (see `http://www.cs.utexas.edu/users/ml/geo.html`).

2.1 Semantic Representation

First-order logic is used as a semantic representation language. CHILL has also been applied to a restaurant database in which the logical form resembles SQL, and is translated

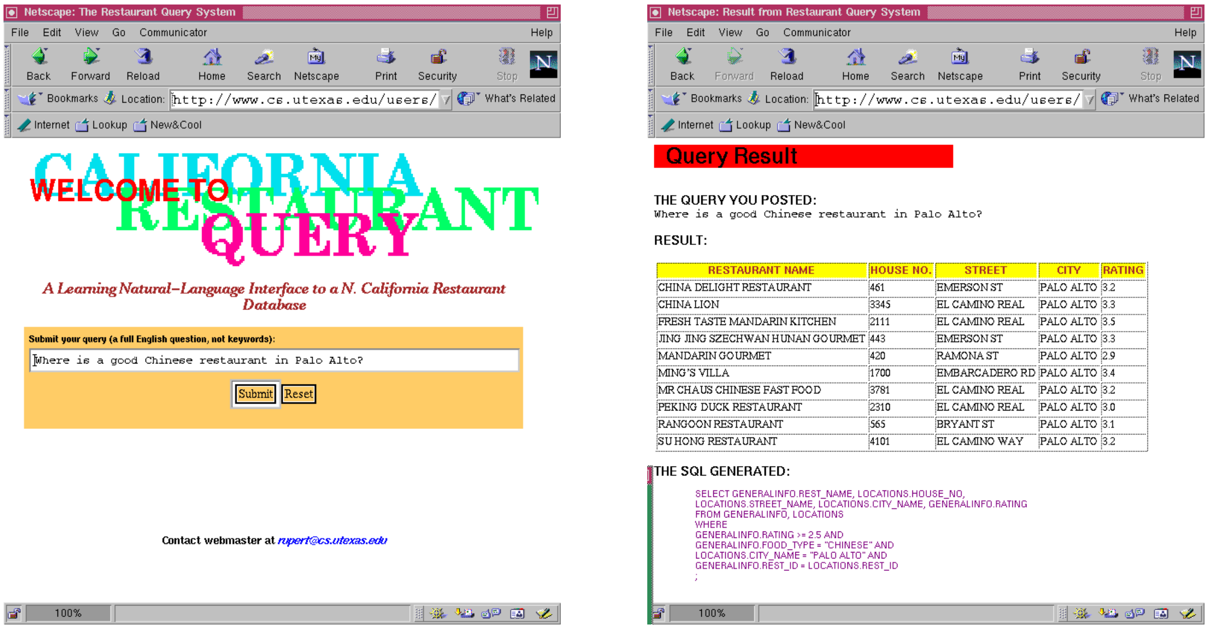


Figure 1: Screenshots of a Learned Web-based NL Database Interface

automatically into SQL (see Figure 1). We explain the features of the Geoquery representation language through a sample query:

Input: "What is the largest city in Texas?"
Query: answer(C,largest(C,(city(C),loc(C,S), const(S,stateid(texas))))).

Objects are represented as logical terms and are typed with a semantic category using logical functions applied to possibly ambiguous English constants (e.g. stateid(Mississippi), riverid(Mississippi)). Relationships between objects are expressed using predicates; for instance, loc(X,Y) states that X is located in Y.

We also need to handle quantifiers such as 'largest'. We represent these using *meta-predicates* for which at least one argument is a conjunction of literals. For example, largest(X, Goal) states that the object X satisfies Goal and is the largest object that does so, using the appropriate measure of size for objects of its type (e.g. area for states, population for cities). Finally, an unspecified object required as an argument to a predicate can appear elsewhere in the sentence, requiring the use of the predicate const(X,C) to bind the variable X to the constant C. Some other database queries (or training examples) for the U.S. Geography domain are shown below:

What is the capital of Texas?
answer(C,(capital(C,S),const(S,stateid(texas)))).

What state has the most rivers running through it?
answer(S,most(S,R,(state(S),river(R),traverse(R,S)))).

2.2 Parsing Actions

Our semantic parser employs a shift-reduce architecture that maintains a stack of previously built semantic constituents and a buffer of remaining words in the input. The parsing actions are automatically generated from templates given the training data. The templates are INTRODUCE, COREF_VARS, DROP_CONJ, LIFT_CONJ, and SHIFT. INTRODUCE pushes a predicate onto the stack based on a word appearing in the input and information about its possible meanings in the lexicon. COREF_VARS binds two arguments of two different predicates on the stack. DROP_CONJ (or LIFT_CONJ) takes a predicate on the stack and puts it into one of the arguments of a meta-predicate on the stack. SHIFT simply pushes a word from the input buffer onto the stack. The parsing actions are tried in exactly this order. The parser also requires a lexicon to map phrases in the input into specific predicates, this lexicon can also be learned automatically from the training data (Thompson and Mooney, 1999).

Let's go through a simple trace of parsing the request "What is the capital of Texas?" A lexicon that maps 'capital' to 'capital(_,_)' and 'Texas' to 'const(_,stateid(texas))' suffices here. Interrogatives like "what" may be mapped to predicates in the lexicon if necessary. The parser begins with an initial stack and a buffer holding the input sentence. Each predicate on the parse stack has an attached buffer to hold the context in which it was introduced; words from the input sentence are shifted onto this buffer during parsing. The initial parse state is shown below:

Parse Stack:	[answer(_,_):[]]
Input Buffer:	[what,is,the,capital,of,texas,?]

Since the first three words in the input buffer do not map to any predicates, three SHIFT actions are performed. The next is an INTRODUCE as 'capital' is at the head of the input buffer:

Parse Stack:	[capital(_,_):[], answer(_,_):[the,is,what]]
Input Buffer:	[capital,of,texas,?]

The next action is a COREF_VARS that binds the first argument of capital(_,_) with the first argument of answer(_,_).

Parse Stack:	[capital(C,_):[], answer(C,_):[the,is,what]]
Input Buffer:	[capital,of,texas,?]

The next sequence of steps are two SHIFT's, an INTRODUCE, and then a COREF_VARS:

Parse Stack:	[const(S,stateid(texas)):[], capital(C,S):[of,capital], answer(C,_):[the,is,what]]
Input Buffer:	[texas,?]

The last four steps are two DROP_CONJ's followed by two SHIFT's:

Parse Stack:	[answer(C, (capital(C,S), const(S,stateid(texas)))): [?,texas,of,capital,the,is,what]]
Input Buffer:	[]

This is the final state and the logical query is extracted from the stack.

2.3 Learning Control Rules

The initially constructed parser has no constraints on when to apply actions, and is therefore overly general and generates numerous spurious parses. Positive and negative examples are collected for each action by parsing each training example and recording the parse states encountered. Parse states to which an action *should* be applied (i.e. the action leads to building the correct semantic representation) are labeled positive examples for that action. Otherwise, a parse state is labeled a negative example for an action if it is a positive example for another action below the current one in the *ordered* list of actions. Control conditions which decide the correct action for a given parse state are learned for each action from these positive and negative examples.

The initial CHILL system used ILP (Lavrac and Dzeroski, 1994) to learn Prolog control rules and employed deterministic parsing, using the learned rules to decide the appropriate parse action for each state. The current approach learns a model for estimating the *probability* that each action should be applied to a given state, and employs statistical parsing (Manning and Schütze, 1999) to try to find the overall most probable parse, using beam search to control the complexity. The advantage of ILP is that it can perform induction over the logical description of the complete parse state without the need to pre-engineer a fixed set of features (which vary greatly from one domain to another) that are relevant to making decisions. We maintain this advantage by using ILP to learn a committee of hypotheses, and basing probability estimates on a weighted vote of them (Ali and Pazzani, 1996). We believe that using such a *probabilistic relational model* (Getoor and Jensen, 2000) combines the advantages of frameworks based on first-order logic and those based on standard statistical techniques.

3 The TABULATE ILP Method

This section discusses the ILP method used to build a committee of logical control hypotheses for each action.

3.1 The Basic TABULATE Algorithm

Most ILP methods use a set-covering method to learn one clause (rule) at a time and construct clauses using either a strictly top-down (general to specific) or bottom-up (specific to general) search through the space of possible rules (Lavrac and Dzeroski, 1994). TABULATE,[1] on the other hand, employs both bottom-up and top-down methods to construct potential clauses and searches through the hypothesis space of complete logic programs (i.e. sets of clauses called *theories*). It uses beam search to find a set of alternative hypotheses guided by a theory evaluation metric discussed below. The search starts with

[1] TABULATE stands for *Top-down And Bottom-Up cLAuse construction with Theory Evaluation.*

```
Procedure Tabulate
Input:
t(X_1,...,X_n):   the target concept to learn
ξ+:               the ⊕ examples
ξ−:               the ⊖ examples
Output:
Q:                a queue of learned theories

Theory_0 := {E ←| E ∈ ξ+}   /* the initial theory */
T(N_0) := Theory_0          /* theory of node N_0 */
C(N_0) := empty             /* the clause being built */
Q := [N_0]                  /* the search queue */
Repeat
  CQ := ∅
  For each search node N_i ∈ Q Do
    If C(N_i) = empty or C(N_i) = fail Then
      Pairs := sampling of S pairs of clauses from T(N_i)
      Find LGG G in Pairs with the greatest cover in ξ+
      R_i := Refine_Clause(t(X_1,...,X_n) ←) ∪
             Refine_Clause(G ←)
    Else
      R_i := Refine_Clause(C(N_i))
    End If
    If R_i = ∅ Then
      CQ_i := {⟨T(N_i), fail⟩}
    Else
      CQ_i := {⟨Complete(T(N_i), G_j, ξ+), next_j⟩ |
               for each G_j ∈ R_i, next_j = empty if G_j
               satisfies the noise criteria; otherwise, G_j}
    End If
    CQ := CQ ∪ CQ_i
  End For
  Q := the B best nodes from Q ∪ CQ
       ranked by metric M
Until termination-criteria-satisfied
Return Q
End Procedure
```

Figure 2: The TABULATE algorithm

the most specific hypothesis (the set of positive examples each represented as a separate clause). Each iteration of the loop attempts to refine each of the hypotheses in the current search queue. There are two cases in each iteration: 1) an existing clause in a theory is refined or 2) a new clause is begun. Clauses are learned using both top-down specialization using a method similar to FOIL (Quinlan, 1990) and bottom-up generalization using Least General Generalizations (LGG's). Advantages of combining both ILP approaches were explored in CHILLIN (Zelle and Mooney, 1994), an ILP method which motivated the design of TABULATE. An outline of the TABULATE algorithm is given in Figure 2.

A *noise-handling criterion* is used to decide when an individual clause in a hypothesis is sufficiently accurate to be permanently retained. There are three possible outcomes in a refinement: 1) the current clause satisfies the noise-handling criterion and is simply returned ($next_j$ is set to *empty*), 2) the current clause does not satisfy the noise-handling criteria and all possible refinements are returned ($next_j$ is set to the refined clause), and 3) the current clause does not satisfy the noise-handling criterion but there are no further refinements ($next_j$ is set to *fail*). If the refinement is a new clause, clauses in the current theory subsumed by it are removed. Otherwise, it is a specialization of an existing clause. Positive examples that are not covered by the resulting theory, due to specializing the clause, are added back into theory as individual clauses. Hence, the theory is always maintained complete (i.e. covering all positive examples). These final steps are performed in the $Complete$ procedure.

The termination criterion checks for two conditions. The first is satisfied if the next search queue does not improve the sum of the metric score over all hypotheses in the queue. Second, there is no clause currently being built for each theory in the search queue and the last finished clause of each theory satisfies the noise-handling criterion. Finally, a committee of hypotheses found by the algorithm is returned.

3.2 Compression and Accuracy

The goal of the search is to find accurate and yet simple hypotheses. We measure accuracy using the *m-estimate* (Cestnik, 1990), a smoothed measure of accuracy on the training data which in the case of a two-class problem is defined as:

$$accuracy(H) = \frac{s + m \cdot p^+}{n + m} \tag{1}$$

where s is the number of positive examples covered by the hypothesis H, n is the total number of examples covered, p^+ is the prior probability of the class $\oplus$, and m is a smoothing parameter.

We measure theory complexity using a metric similar to that introduced in (Muggleton and Buntine, 1988). The size of a $Clause$ having a $Head$ and a $Body$ is defined as follows (ts="term size" and ar="arity"):

$$size(Clause) = 1 + ts(Head) + ts(Body) \tag{2}$$

$$ts(T) = \begin{cases} 1 & \text{T is a variable} \\ 2 & \text{T is a constant} \\ 2 + \sum_{i=1}^{ar(T)} ts(arg_i(T)) & \text{otherwise.} \end{cases} \quad (3)$$

The size of a clause is roughly the number of variables, constants, or predicate symbols it contains. The size of a theory is the sum of the sizes of its clauses. The metric $M(H)$ used as the search heuristic is defined as:

$$M(H) = \frac{accuracy(H) + C}{\log_2 size(H)} \quad (4)$$

where C is a constant used to control the relative weight of accuracy vs. complexity. We assume that the most general hypothesis is as good as the most specific hypothesis; thus, C is determined to be:

$$C = \frac{E_b S_t - E_t S_b}{S_b - S_t} \quad (5)$$

where E_t, E_b are the accuracy estimates of the most general and most specific hypotheses respectively, and S_t, S_b are their sizes.

3.3 Noise Handling

A clause needs no further refinement when it meets the following criterion (as in RIPPER (Cohen, 1995)):

$$\frac{p - n}{p + n} > \beta \quad (6)$$

where p is the number of positive examples covered by the clause, n is the number of negative examples covered and $-1 \leq \beta \leq 1$ is a parameter. The value of β is decreased whenever the sum of the metric over the hypotheses in the queue does not improve although some of them still have unfinished or failed clauses.

4 Statistical Semantic Parsing

4.1 The Parsing Model

A *parser* is a relation $Parser \subseteq Sentences \times Queries$ where $Sentences$ and $Queries$ are the sets of natural language sentences and database queries respectively. Given a sentence $l \in Sentences$, the set $Q(l) = \{q \in Queries \mid \langle l, q\rangle \in Parser\}$ is the set of queries that are correct interpretations of l.

A *parse state* consists of a stack of *lexicalized* predicates and a list of words from the input sentence. S is the set of states reachable by the parser. Suppose our learned parser has n different parsing actions, the ith action a_i is a function $a_i(s) : IS_i \rightarrow OS_i$ where $IS_i \subseteq S$ is the set of states to which the action is applicable and $OS_i \subseteq S$ is the set of states constructed by the action. The function $a_0(l) : Sentences \rightarrow IniS$ maps each sentence l to a corresponding unique *initial* parse state in $IniS \subseteq S$. A state is called a *final* state if there exists no parsing action applicable to it. The *partial* function $a_{n+1}(s) : FS \rightarrow Queries$ is defined as the action that retrieves the query from the final state $s \in FS \subseteq S$ if one exists. Some final states may not "contain" a query (e.g. when the parse stack contains predicates with unbound variables) and therefore it is a partial function. When the parser meets such a final state, it reports a failure.

A *path* is a finite sequence of parsing actions. Given a sentence l, a *good* state s is one such that there exists a path from it to a query $q \in Q(l)$. Otherwise, it is a *bad* state. The set of parse states can be uniquely divided into the set of good states and the set of bad states given l and $Parser$. S^+ and S^- are the sets of good and bad states respectively.

Given a sentence l, the goal is to construct the query $\hat{q}$ such that

$$\hat{q} = \underset{q}{\operatorname{argmax}}\ P(q \in Q(l) \mid l \Rightarrow q) \quad (7)$$

where $l \Rightarrow q$ means a path exists from l to q.

Now, we need to estimate $P(q \in Q(l) \mid l \Rightarrow q)$. First, we notice that:

$$\begin{aligned} &P(q \in Q(l) \mid l \Rightarrow q) = \quad (8) \\ &P(s \in FS^+ \mid l \Rightarrow s \text{ and } a_{n+1}(s) = q) \end{aligned}$$

where $FS^+ = FS \cap S^+$. For notational convenience we drop the conditions and denote the above probabilities as $P(q \in Q(l))$ and $P(s \in FS^+)$ respectively, assuming these conditions in the following discussion. The equation states that the probability that a given query is a correct meaning for l is the same as the probability that the final state (reached by parsing l) is a good state. We need to determine in general the probability of having a good resulting parse state. Given any parse state s_j at the jth step of parsing and an action a_i such that $s_{j+1} = a_i(s_j)$, we have:

$$\begin{aligned} &P(s_{j+1} \in OS_i^+) = \quad (9) \\ &P(s_{j+1} \in OS_i^+ \mid s_j \in IS_i^+) P(s_j \in IS_i^+) + \\ &P(s_{j+1} \in OS_i^+ \mid s_j \notin IS_i^+) P(s_j \notin IS_i^+) \end{aligned}$$

where $IS_i^+ = IS_i \cap S^+$ and $OS_i^+ = OS_i \cap S^+$. Since no parsing action can produce a good parse state from a bad one, the second term is zero. Now, we are ready to derive $P(q \in Q(l))$. Suppose $q = a_{n+1}(s_m)$, we have:

$$\begin{aligned} & P(q \in Q(l)) \qquad (10) \\ & = P(s_m \in FS^+) \\ & \dots \\ & = P(s_m \in FS^+ \mid s_{m-1} \in IS^+_{a_{m-1}}) \dots \\ & \quad P(s_j \in OS^+_{a_{j-1}} \mid s_{j-1} \in IS^+_{a_{j-1}}) \dots \\ & \quad P(s_2 \in OS^+_{a_1} \mid s_1 \in IS^+_{a_1}) P(s_1 \in IS^+_{a_1}) \end{aligned}$$

where a_k denotes the index of which action is applied at the kth step. We assume that $\gamma = P(s_1 \in IS^+_{a_1}) \neq 0$ (which may not be true in general). Now, we have

$$P(q \in Q(l)) = \gamma \prod_{j=1}^{m-1} P(s_{j+1} \in OS^+_{a_j} \mid s_j \in IS^+_{a_j}). \qquad (11)$$

Next we describe how we estimate the probability of the goodness of each action in a given state ($P(a_i(s) \in OS^+_i \mid s \in IS^+_i)$). We need not estimate γ since its value does not affect the outcome of equation (7).

4.2 Estimating Probabilities for Parsing Actions

The committee of hypotheses learned by TABULATE is used to estimate the probability that a particular action is a good one to apply to a given parse state. Some hypotheses are more "important" than others in the sense that they carry more weight in the decision. A weighting parameter is also included to lower the probability estimate of actions that appear further down the decision list. For actions a_i where $1 \leq i \leq n-1$:

$$\begin{aligned} & P(a_i(s) \in OS^+_i \mid s \in IS^+_i) = \qquad (12) \\ & \quad \mu^{pos(i)-1} \sum_{h_k \in H_i} \lambda_k P(a_i(s) \in OS^+_i \mid h_k) \end{aligned}$$

where s is a given parse state, $pos(i)$ is the position of the action a_i in the list of actions applicable to state s, λ_k and $0 < \mu \leq 1$ are weighting parameters,[2] H_i is the set of hypotheses learned for the action a_i, and $\sum_k \lambda_k = 1$.

To estimate the probability for the last action a_n, we devise a simple test that checks if the maximum of the set $A(s)$ of probability estimates for the subset of the actions $\{a_1, \dots, a_{n-1}\}$ applicable to s is less than or equal to a threshold α. If $A(s)$ is empty, we assume the maximum is zero. More precisely,

$$P(a_n(s) \in OS^+_n \mid s \in IS^+_n) = \qquad (13)$$
$$\begin{cases} \frac{c(a_n(s) \in OS^+_n)}{c(s \in IS^+_n)} & \text{if } max(A(s)) \leq \alpha \\ 0 & \text{otherwise} \end{cases}$$

where α is the threshold,[3] $c(a_n(s) \in OS^+_n)$ is the count of the number of good states produced by the last action, and $c(s \in IS^+_n)$ is the count of the number of good states to which the last action is applicable.

Now, let's discuss how $P(a_i(s) \in OS^+_i \mid h_k)$ and λ_k are estimated. If $h_k \models s$ (i.e. h_k covers s), we have

$$P(a_i(s) \in OS^+_i \mid h_k) = \frac{p_c + \theta \cdot n_c}{p_c + n_c} \qquad (14)$$

where p_c and n_c are the number of positive and negative examples covered by h_k respectively. Otherwise, if $h_k \not\models s$ (i.e. h_k does not cover s), we have

$$P(a_i(s) \in OS^+_i \mid h_k) = \frac{p_u + \theta \cdot n_u}{p_u + n_u} \qquad (15)$$

where p_u and n_u are the number of positive and negative examples rejected by h_k respectively. θ is the probability that a negative example is mislabelled and its value can be estimated given β (in equation (6)) and the total number of positive and negative examples.

One could use a variety of linear combination methods to estimate the weights λ_k (e.g. Bayesian combination (Buntine, 1990)). However, we have taken a simple approach and weighted hypotheses based on their relative simplicity:

$$\lambda_k = \frac{size(h_k)^{-1}}{\sum_{j=1}^{|H_i|} size(h_j)^{-1}}. \qquad (16)$$

4.3 Searching for a Parse

To find the most probably correct parse, the parser employs a beam search. At each step, the parser finds all of the parsing actions applicable to each parse state on the queue and calculates the probability of goodness of each of them using equations (12) and (13). It then

[2] μ is set to 0.95 for all the experiments performed.

[3] The threshold is set to 0.5 for all the experiments performed.

computes the probability that the resulting state of each possible action is a good state using equation (11), sorts the queue of possible next states accordingly, and keeps the best B options. The parser stops when a complete parse is found on the top of the parse queue or a failure is reported.

5 Experimental Results

5.1 The Domains

Three different domains are used to demonstrate the performance of the new approach. The first is the U.S. Geography domain. The database contains about 800 facts about U.S. states like population, area, capital city, neighboring states, major rivers, major cities, and so on. A hand-crafted parser, GEOBASE was previously constructed for this domain as a demo product for Turbo Prolog. The second application is the restaurant query system illustrated in Figure 1. The database contains information about thousands of restaurants in Northern California, including the name of the restaurant, its location, its specialty, and a guide-book rating. The third domain consists of a set of 300 computer-related jobs automatically extracted from postings to the USENET newsgroup `austin.jobs`. The database contains the following information: the job title, the company, the recruiter, the location, the salary, the languages and platforms used, and required or desired years of experience and degrees.

5.2 Experimental Design

The geography corpus contains 560 questions. Approximately 100 of these were collected from a log of questions submitted to the web site and the rest were collected in studies involving students in undergraduate classes at our university. We also included results for the subset of 250 sentences originally used in the experiments reported in (Zelle and Mooney, 1996). The remaining questions were specifically collected to be more complex than the original 250, and generally require one or more meta-predicates. The restaurant corpus contains 250 questions automatically generated from a hand-built grammar constructed to reflect typical queries in this domain. The job corpus contains 400 questions automatically generated in a similar fashion. The beam width for TABULATE was set to five for all the domains. The deterministic parser used only the best hypothesis found. The experiments were conducted using 10-fold cross validation.

For each domain, the average recall (a.k.a. accuracy) and precision of the parser on disjoint test data are reported where:

$$
\begin{aligned}
\textit{Recall} &= \frac{\#\text{ of correct queries produced}}{\#\text{ of test sentences}} \\
\textit{Precision} &= \frac{\#\text{ of correct queries produced}}{\#\text{ of complete parses produced}}.
\end{aligned}
$$

A complete parse is one which contains an executable query (which could be incorrect). A query is considered correct if it produces the same answer set as the gold-standard query supplied with the corpus.

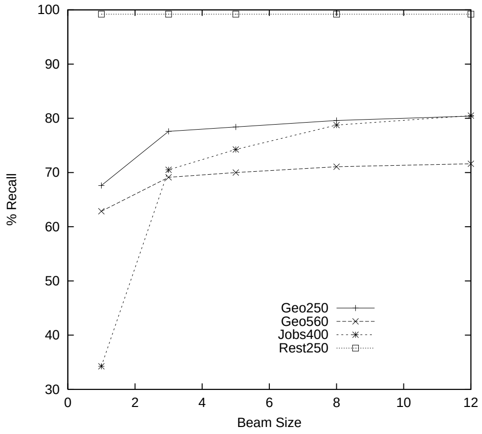
5.3 Results

The results are presented in Table 1 and Figure 3. By switching from deterministic to probabilistic parsing, the system increased the number of correct queries it produced. Recall increases almost monotonically with parsing beam width in most of the domains. Improvement is most apparent in the Jobs domain where probabilistic parsing significantly outperformed the deterministic system (80% vs 68%). However, using a beam width of one (and thus the probabilistic parser picks only the best action) results in worse performance than using the original purely logic-based deterministic parser. This suggests that the probability esitmates could be improved since overall they are not indicating the single best action as well as a non-probabilistic approach. Precision of the system decreased with beam width, but not significantly except for the larger Geography corpus. Since the system conducts a more extensive search for a complete parse, it risks increasing the number of incorrect as well as correct parses. The importance of recall vs. precision depends on the relative cost of providing an incorrect answer versus no answer at all. Individual applications may require emphasizing one or the other.

All of the experiments were run on a 167MHz UltraSparc work station under Sicstus Prolog. Although results on the parsing time of the different systems are not formally reported here, it was noted that the difference between using a beam width of three and the original system was less than two seconds in all domains but increased to around twenty seconds when using a beam width of twelve. However, the current Prolog implementation is not highly optimized.

Parsers \ Corpora	Geo250		Geo560		Jobs400		Rest250	
	R	P	R	P	R	P	R	P
Prob-Parser(12)	80.40	88.16	71.61	78.94	80.50	86.56	99.20	99.60
Prob-Parser(8)	79.60	86.90	71.07	79.76	78.75	86.54	99.20	99.60
Prob-Parser(5)	78.40	87.11	70.00	79.51	74.25	86.59	99.20	99.60
Prob-Parser(3)	77.60	87.39	69.11	79.30	70.50	87.31	99.20	99.60
Prob-Parser(1)	67.60	90.37	62.86	82.05	34.25	85.63	99.20	99.60
TABULATE	75.60	92.65	69.29	89.81	68.50	87.54	99.20	99.60
Original CHILL	68.50	97.65	——		——		——	
Hand-Built Parser	56.00	96.40	——		——		——	

Table 1: Results For All Domains: R = % Recall and P = % Precision. Prob-Parser(B) is the probabilistic parser using a beam width of B. TABULATE is CHILL using the TABULATE induction algorithm with deterministic parsing.



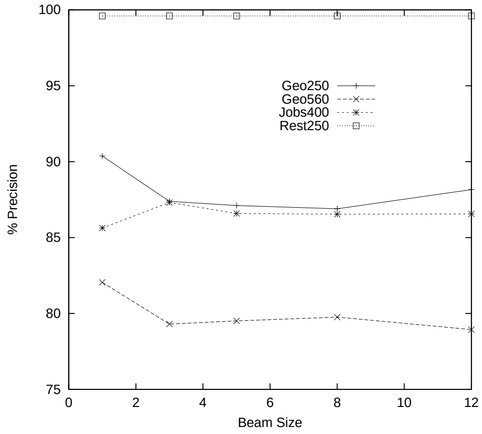


Figure 3: The recall and precision of the parser using various beam widths in the different domains

While there was an overall improvement in recall using the new approach, its performance varied significantly from domain to domain. As a result, the recall did not always improve dramatically by using a larger beam width. Domain factors possibly affecting the performance are the quality of the lexicon, the relative amount of data available for calculating probability estimates, and the problem of "parser incompleteness" with respect to the test data (i.e. there is not a path from a sentence to a correct query which happens when $\gamma = 0$). The performance of all systems were basically equivalent in the restaurant domain, where they were near perfect in both recall and precision. This is because this corpus is relatively easier given the restricted range of possible questions due to the limited information available about each restaurant. The systems achieved $> 90\%$ in recall and precision given only roughly 30% of the training data in this domain. Finally, GEOBASE performed the worst on the original geography queries, since it is difficult to hand-craft a parser that handles a sufficient variety of questions.

6 Conclusion

A probabilistic framework for semantic shift-reduce parsing was presented. A new ILP learning system was also introduced which learns multiple hypotheses. These two techniques were integrated to learn semantic parsers for building NLI's to online databases. Experimental results were presented that demonstrate that such an approach outperforms a purely logical approach in terms of the accuracy of the learned parser.

7 Acknowledgements

This research was supported by a grant from the Daimler-Chrysler Research and Technology Center and by the National Science Foundation under grant IRI-9704943.

References

K. Ali and M. Pazzani. 1996. Error reduction through learning multiple descriptions. *Machine Learning Journal*, 24:3:100–132.

W. Buntine. 1990. *A theory of learning classification rules.* Ph.D. thesis, University of Technology, Sydney, Australia.

B. Cestnik. 1990. Estimating probabilities: A crucial task in machine learning. In *Proceedings of the Ninth European Conference on Artificial Intelligence*, pages 147–149, Stockholm, Sweden.

W. W. Cohen. 1995. Fast effective rule induction. In *Proceedings of the Twelfth International Conference on Machine Learning*, pages 115–123.

L. Getoor and D. Jensen, editors. 2000. *Papers from the AAAI Workshop on Learning Statistical Models from Relational Data*, Austin, TX. AAAI Press.

G. G. Hendrix, E. Sacerdoti, D. Sagalowicz, and J. Slocum. 1978. Developing a natural language interface to complex data. *ACM Transactions on Database Systems*, 3(2):105–147.

R. Kuhn and R. De Mori. 1995. The application of semantic classification trees to natural language understanding. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 17(5):449–460.

N. Lavrac and S. Dzeroski. 1994. *Inductive Logic Programming: Techniques and Applications.* Ellis Horwood.

C. D. Manning and H. Schütze. 1999. *Foundations of Statistical Natural Language Processing.* MIT Press, Cambridge, MA.

Scott Miller, David Stallard, Robert Bobrow, and Richard Schwartz. 1996. A fully statistical approach to natural language interfaces. In *Proceedings of the 34th Annual Meeting of the Association for Computational Linguistics*, pages 55–61, Santa Cruz, CA.

S. Muggleton and W. Buntine. 1988. Machine invention of first-order predicates by inverting resolution. In *Proceedings of the Fifth International Conference on Machine Learning*, pages 339–352, Ann Arbor, MI, June.

J. R. Quinlan. 1990. Learning logical definitions from relations. *Machine Learning*, 5(3):239–266.

C. A. Thompson and R. J. Mooney. 1999. Automatic construction of semantic lexicons for learning natural language interfaces. In *Proceedings of the Sixteenth National Conference on Artificial Intelligence*, pages 487–493, Orlando, FL, July.

W. A. Woods. 1970. Transition network grammars for natural language analysis. *Communications of the Association for Computing Machinery*, 13:591–606.

J. M. Zelle and R. J. Mooney. 1994. Combining top-down and bottom-up methods in inductive logic programming. In *Proceedings of the Eleventh International Conference on Machine Learning*, pages 343–351, New Brunswick, NJ, July.

J. M. Zelle and R. J. Mooney. 1996. Learning to parse database queries using inductive logic programming. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence*, pages 1050–1055, Portland, OR, August.