Unpublished Technical Note, Apr 2004

Learning Transformation Rules for Semantic Parsing

Rohit J. Kate Yuk Wah Wong Ruifang Ge Raymond J. Mooney
Department of Computer Sciences
University of Texas, Austin
TX 78712, USA
{rjkate,ywwong,grf,mooney}@cs.utexas.edu

Abstract

This paper presents an approach for inducing transformation rules that map natural-language sentences into a formal semantic representation language. The approach assumes a formal grammar for the target representation language and learns transformation rules that exploit the non-terminal symbols in this grammar. Patterns for the transformation rules are learned using an induction algorithm based on longest-common-subsequences previously developed for an information extraction system. Experimental results are presented on learning to map English coaching instructions for Robocup soccer into an existing formal language for coaching simulated robotic agents.

1 Introduction

The ability to map natural language to a pre-existing formal query or command language is critical to developing more usable interfaces to many computing applications (e.g. databases (Woods, 1977)). However, relatively little research in empirical NLP has addressed the problem of learning such semantic parsers from a corpus of sentences paired with their formal-language equivalents.

As a part of a larger project on advice-taking reinforcement-learning agents, we are developing a system for simulated robotic soccer that can take coaching instructions in natural language. In the extant RoboCup Coach Competition, teams compete to provide effective instructions to a *coachable team* in a simulated soccer domain. Coaching advice is provided in a formal language called CLANG (Coach Language) comprised of if-then rules for influencing players' behavior (Chen et al., 2003). Therefore, in this paper, we consider the problem of learning to translate English coaching advice into formal CLANG instructions (see Section 2 for an example).

The few previous systems for directly learning semantic parsers have employed complex inductive-logic-programming methods to acquire parsers for mapping sentences to a logical query language (Zelle and Mooney, 1996; Tang and Mooney, 2000). In this paper, we introduce a much simpler approach based on learning string-to-tree transformation rules. The approach assumes that a deterministically-parsable formal grammar for the target representation language is available. Transformation rules are learned that map substrings in natural-language to sub-trees of the formal-language parse tree. The non-terminal symbols in the target-language grammar provide convenient intermediate representations that enable the construction of general, effective transformation rules. Transformation rules for introducing each of of the productions in the target-language grammar are induced using a pattern-learning algorithm based on longest common subsequences previously developed for an information extraction task (Bunescu et al., 2004).

Our approach has been implemented in a system called SILT (Semantic Interpretation by Learning Transformations). Using an assembled corpus of 300 sentences, we present experimental results on learning to translate natural-language RoboCup coaching instructions into formal CLANG expressions. SILT is able to produce completely correct parses for more than a third of novel sentences, and mostly correct parses for most others.

2 CLANG: The RoboCup Coach Language

RoboCup[1] is an international AI research initiative using robotic soccer as its primary domain. In the Coach Competition, teams of agents compete on a simulated soccer field, each player controlled by a separate client that makes intelligent decisions based on its limited perception of the field. During the game, players may receive advice from a human or automated coach. In particular, a human observer may act as a coach and change the behavior of individual players by giving advice. Since the goal of the competition is to test how well coaches work with teams developed by different research groups, a standard, formal coaching language called CLANG (Chen et al., 2003) has been developed to facilitate communication between coaches and players.

CLANG is a simple declarative programming language with a set of domain-specific terms composed using a primarily prefix notation like LISP. Tactics and behaviors are described as if-then *rules*, which consist of a *condition* description and a list of *directives* that are applicable when the condition is true. Directives are lists of *actions* that individual sets of players should or should not take. There are expressions for soccer-specific entities such as *regions* and *points*. Coreference among expressions is possible using variables, but is restricted to player numbers and is currently infrequently used.

We have supplemented CLANG with additional expression types for those concepts that are easily expressible in natural language but not in CLANG, such as "all our players except for player n" and "left half of region r". These new expression types can be automatically converted into original CLANG.

[1] `http://www.robocup.org/`

CLANG has a deterministic context free grammar. Below are some of its productions:

- RULE $\rightarrow$ `(`CONDITION DIRECTIVE`)`
- CONDITION $\rightarrow$ `(bpos` REGION`)`
- DIRECTIVE $\rightarrow$ `(do` PLAYER ACTION`)`
- ACTION $\rightarrow$ `(pos` REGION`)`
- REGION $\rightarrow$ `(penalty-area` TEAM`)`

Below is a sample pair of an English statement with its corresponding CLANG:

- "If the ball is in our penalty area, the goalie should stay in front of our goal."

```
((bpos (penalty-area our))
 (do (player our {1})
     (pos (front-of-goal our))))
```

Conditions and regions, among other entities, can be pre-defined in separate statements, and the coach may refer to them later using identifiers, making the rules more concise. Below is a sample definition of a region, and its subsequent use in a rule:

- "Define REG1 to be our penalty area."

```
(definer "REG1" (penalty-area our))
```

- "If the ball is in REG1, the goalie should stay in front of our goal."

```
((bpos "REG1")
 (do (player our {1})
     (pos (front-of-goal our))))
```

CLANG is a useful test-bed for semantic parsing because although it is relatively simple, it was not constructed specifically for natural-language interpretation and allows expressing a wide range of instructions for a realistic application. The language has been in use for over 2 years by the worldwide RoboCup community, and has been actively maintained since its inception.

3 Semantic Parsing by String-to-Tree Transformation

We have developed a new approach called SILT (Semantic Interpretation by Learning Transformations) for mapping natural-language sentences to their formal representations. Approaches based on transformation rules have previously been used for other tasks (Brill, 1995). SILT uses pattern-based transformation rules which map phrases in natural language directly to the productions in the CLANG grammar. These transformation rules are repeatedly applied to construct the parse tree for the corresponding CLANG representation in a bottom-up manner.

3.1 Rule Representation

SILT's transformation rules consist of a pattern and a CLANG production to be introduced when this pattern is matched. The pattern is a sequence of words and CLANG non-terminal symbols. Every pair of adjacent words or non-terminals is separated by a number which stands for the maximum number of words allowed between the two (called a *word gap*). A sample rule is:

> "If **player N has**<1> **ball**" ⇒
> CONDITION → `(bowner our {N})`

here "N" and "CONDITION" are non-terminals, the "<1>" between "has" and "ball" is a word gap, all other word gaps are zero and have been omitted for clarity. A rule matches a sentence if the sentence satisfies all the word and non-terminal constraints in the given order and respects all the respective word gaps.

A rule also has a special contiguous portion of the pattern called the *replacement subpattern*, shown in bold in the example. When the rule is applied to a sentence, it replaces the portion of the sentence that matches the replacement subpattern with the production's left-hand-side (LHS) non-terminal (i.e. "CONDITION" in the example, an example trace is given in the next subsection). The purpose of replacement subpatterns is to allow the rest of the pattern to specify the context for the rule's application without actually contributing to the semantic concept introduced by the rule. Since this context may contribute to another semantic concept, it should be preserved when the rule is applied. In the example, "If" represents the additional context in which the rule should be applied.

Finally, a rule's replacement subpattern must contain all of the non-terminals present in the right-hand-side (RHS) of the production (N in the example). This condition is needed for successful semantic parsing as described in the next subsection.

3.2 Semantic Parsing

SILT builds the semantic parse tree bottom-up by repeatedly applying the transformation rules to the given natural-language sentence. When a rule is applied to a sentence, the matched portion of its replacement subpattern is replaced by the LHS non-terminal of the production. This non-terminal is expanded into a subtree according to the production. In this expansion, RHS non-terminals required by the production are provided by the non-terminals (sub-trees) present in the sentence that matched the non-terminals of the replacement subpattern (hence the requirement that all RHS non-terminals should be present in it). We illustrate this process through an example, assuming we have already acquired the transformation rules. In the next section, we present how transformations are learned. Consider the sentence:
"If player 2 has the ball, player 2 should pass to player 10." A pre-processing step replaces every number in the sentence by the non-terminal N expanded into that number. In this running example we show expansions by square brackets. After this the sentence becomes:

> "If player N[2] has the ball, player N[2] should pass to player N[10]."

Next, the transformation rule:

> "If **player N has** <1> **ball**" ⇒
> CONDITION → `(bowner our {N})`

rewrites the sentence to:

> "If CONDITION[(bowner our {N[2]})], player N[2] should pass to player N[10]."

Here the replacement subpattern "**player N has** <1> **ball**" replaced "player 2 has the ball" in the sentence by the non-terminal CONDITION. This non-terminal is then expanded into "`(bowner our {N})`". The RHS non-terminal N in the expansion is taken from the sentence where the N of the rule's pattern matched the sentence.
Next:

> "should **pass to player N**" ⇒ ACTION
> → `(pass {N})`

rewrites the sentence to:

"If CONDITION[(bowner our {N[2]})], player 2 should ACTION[(pass {N[10]})]."

Next:

"**player N should ACTION**" ⇒
DIRECTIVE → (do our {N} ACTION)

rewrites the result to:

"If CONDITION[(bowner our {N[2]})], DIRECTIVE[do our {N[2]} ACTION[(pass {N[10]})]]."

Finally:

"**If CONDITION <2> DIRECTIVE.**" ⇒
RULE → (CONDITION DIRECTIVE)

produces the final CLANG parse:

RULE[CONDITION[(bowner our {N[2]})]
DIRECTIVE[do our {N[2]} ACTION[(pass {N[10]})]]]

SILT's transformation procedure can be looked upon as compositionally doing "information extraction" of the non-terminals of the semantic language.

4 Learning Transformation Rules

SILT induces transformation rules from a training set of natural-language sentences paired with their formal representations and the grammar of the formal language. First, all of the formal representations in the training data are parsed using the grammar of the formal language. Since the language is deterministic, parsing is unambiguous. These parses are then used to build positive and negative example sets for each of the productions. If production π is used in a parse of a formal representation then the corresponding natural language sentence is included in π's positive example set $\mathcal{P}(\pi)$ otherwise it is included in its negative example set $\mathcal{N}(\pi)$. It is possible for a production to have multiple applications in a parse. For this reason, each natural language sentence p in the positive example set $\mathcal{P}(\pi)$ is also given a count p_c equal to the number of times π was used in the parse of p's semantic representation.

Using a bottom-up (specific-to-general) learning method, SILT induces transformation rules for every production from its positive and negative example sentences (similar to rule learning algorithm GOLEM (Muggleton and Feng, 1990)). Rule induction starts with separate, maximally-specific rules for each positive sentence which contains all of the words in the sentence with zero-length word gaps. These rules are then repeatedly generalized to form more general rules until the rules become overly general and start matching negative sentences. Hence, a key operation in SILT's rule learning algorithm is generalization of two existing rules which is described in the next subsection. The bottom-up algorithm for learning a single transformation rule for a production is described next. The last subsection describes the complete algorithm for learning transformation rules for all the productions.

4.1 Generalization of Rules

Given two rules with patterns $r = r_1 r_2 ... r_l$ and $r' = r'_1 r'_2 ... r'_m$, Figure 1 gives the pseudo-code for finding their generalization. First, the set $\mathcal{C}(r, r')$ of all the non-contiguous common subsequences between r and r' is computed. For every common subsequence $c = c_1 c_2 ... c_n$ in this set, the word gap, $g(c_i, c_{i+1})$, between every two of its adjacent words c_i and c_{i+1}, is computed as follows. Let s and e be the places where c_i and c_{i+1} appear in r. Similarly, let s' and e' be the places where c_i and c_{i+1} appear in r', then the word gap $g(c_i, c_{i+1})$ is:

$$max(e-s+\sum_{j=s}^{e-1} g(r_j, r_{j+1}), e'-s'+\sum_{j=s'}^{e'-1} g(r'_j, r'_{j+1}))$$

This is the larger of the number of words plus the sum of existing word gaps between the two words where c_i and c_{i+1} were present in r and r'. Since the pattern of a generalized rule is required to have all the RHS non-terminals of its production, those common subsequences which do not have all the required non-terminals are deleted from $\mathcal{C}(r, r')$. Since the subpattern having the required non-terminals always gets replaced (by the LHS non-terminal) when the rule is applied, it is necessary not to lose any other non-terminal of the sentence in this process (since it does not appear in the expansion of the LHS non-terminal of the rule's production, it will never appear in the final result). For this reason, those common subsequences which have some other non-terminal present between the required non-terminals are also deleted. The remaining common subsequences are scored for their generalization quality. Longer common subsequences

Input: Two rules r and r'
Production π for the rule being learned
Output: Generalization of r and r'
Parameter: η - negative weight for the word gaps
Function generalize(r, r')
Let $\mathcal{C}(r, r')$ ={all common sequences between r and r' with their word gaps}
Delete from $\mathcal{C}(r, r')$ those sequences which don't have all the RHS non-terminals of π
Delete from $\mathcal{C}(r, r')$ those sequences which have any other non-terminals between the RHS non-terminals of π
Return $argmax_{c \in \mathcal{C}(r,r')}(score(c))$, where
$score(c) = length(c) - \eta * (sum\, of\, c's\, word\, gaps)$

Figure 1: Method for generalizing two rules or examples.

Sentence 1: During a penalty kick position player N at REGION.
Sentence 2: Whenever the ball is in REGION the position of player N should be at REGION.
Generalization: position <1> player N <2> at REGION.

Figure 2: Generalization of two sentences for learning pattern for ACTION $\rightarrow$ (pos REGION).

are better generalizations since they are conservative generalizations and, hence, less likely to match negative examples. But since the pattern of the rule typically represents a natural-language phrase, it is also desirable that the words in generalizations are close-by, therefore large word gaps should be avoided. Considering these two factors, we define the score of a common subsequence $c = c_1 c_2 ... c_n$ as:

$$score(c) = length(c) - \eta * \sum_{i=0}^{n-1}(g(c_i, c_{i+1}))$$

where η is a parameter that determines the size of the word-gap penalty. The common sequence with the highest score is returned as the desired generalization. Figure 2 shows a sample generalization.

4.2 Rule Learning

Using the generalization scheme described in previous subsection, SILT employs a greedy-covering, bottom-up rule-induction method to learn a rule that covers as many positive sentences as possible while covering few or no negative sentences. This pattern learning algorithm is similar to the ELCS system, used for an information extraction task (Bunescu et al., 2004). Figure 3 gives the pseudo-code for the algorithm. It uses beam search and considers only the β best rules for generalization at any time. It starts with β randomly

Input: $\mathcal{P}(\pi)$ and $\mathcal{N}(\pi)$ - set of positive and negative examples for production π
Output: A rule which covers many positive examples and a very few or no negative example
Parameter: β - beam width for search
Function learnRule($\mathcal{P}(\pi)$, $\mathcal{N}(\pi)$)
Randomly select β positive examples
$P = \{p_1, p_2 ... p_\beta\}$ from $\mathcal{P}(\pi)$
For each of the remaining positive example p
  For $i = 1..\beta$
    Add $generalize(p, p_i)$ to P and
    compute its accuracy using $\mathcal{P}(\pi)$ and $\mathcal{N}(\pi)$
  Sort P by the accuracies of its elements
  Retain the top β rules, delete the remaining
Return the first element (i.e. the best rule) in P.

Figure 3: Algorithm for learning one rule

selected positive examples and their initial rules are generalized with one of the remaining positive examples to obtain β more rules. From the resulting 2β rules, it selects the β most accurate rules and attempts to generalize them further with the remaining positive examples. The accuracy of a rule r is measured as:

$$accuracy(r) = \frac{pos(r)}{pos(r) + neg(r) + \epsilon}$$

where $pos(r)$ and $neg(r)$ are the number of positive and negative examples the rule r matches. A small positive value ϵ is added so that when $neg(r)$ is zero the rule with larger $pos(r)$ is considered more accurate. After iterating over the remaining positive examples, the best rule is returned. We describe later how SILT determines the replacement subpattern for a learned rule.

4.3 Complete Learning Algorithm

SILT learns transformation rules for all productions by using the algorithm described in the previous subsection for learning a single transformation rule for a given production. Figure 4 gives the pseudo-code for the complete learning algorithm.

First, the productions of the formal language are divided into levels. This fixes the order in which the transformation rules for various productions will be learned. Productions that have *no* RHS non-terminals (e.g. REGION $\rightarrow$ `(midfield)`, POINT $\rightarrow$ `(pt ball)`) are included in the first level. Each subsequent higher level then includes productions whose RHS non-terminals are all present as LHS non-terminals at lower levels. For example, the production ACTION $\rightarrow$ `(pass`

REGION) is at one level above the highest-level production with REGION as its LHS non-terminal. If a production is recursive, then it is added to the level immediately above the levels of all of its base cases.

Since the productions at a higher level depend on the productions at lower levels for their RHS non-terminals, the transformation rules for the productions are learned in the order of their levels. The learned transformation rules are applied to the sentences, and then the algorithm proceeds to the next level. At any level, transformation rules for all its productions can be learned simultaneously because these productions do not depend on each other for their RHS non-terminals. Simultaneous learning is beneficial because it allows the transformation rules to compete with each other when determining their patterns from the same set of sentences. This idea of learning multiple rules for different concepts simultaneously is similar to Multiple Predicate Learning (De Raedt et al., 1993).

At each level, the algorithm obtains the best rule for each of the productions at that level by invoking the algorithm in Figure 3. From the resulting rules, the rule with the best accuracy is selected for inclusion in the set of learned transformation rules. The replacement subpattern for a selected rule is determined as follows. As argued earlier, it must contain all of the non-terminals in the RHS of its production, and no other non-terminals. All possible subpatterns of the rule's pattern satisfying this criteria are considered, starting with the largest one. A desirable property of a replacement subpattern is that it not include any terms that act only as context for the rule's application and therefore could be part of another rule's replacement subpattern that will be learned subsequently. If a subpattern matches some positive sentences of a production whose rules are yet to be learned, but none of its negative sentences, then this is a strong indication that the subpattern could become a part of a rule's pattern when rules are later learned for that production. The largest subpattern which does not so indicate that it could become part of another rule's pattern, is marked as the replacement subpattern.

Next, the resulting rule is applied to all the positive sentences for its production. As described in section 3.1, a rule application replaces the portion of the sentence matched by the replacement pattern, introduces the LHS non-terminal of its production, and expands this non-terminal according to the production. It is possible for a rule to be applicable to a sentence multiple times. Hence, the rule is applied to a sentence, p, as many times as possible while decrementing its count p_c every time. If the count p_c becomes zero, it means that the algorithm has introduced and expanded as many LHS non-terminals of the rule's production in the sentence as there were instances of that production used in the parse of the corresponding formal representation. At this point, the sentence is deleted from the production's positive set and is included in its negative set to prevent any further introduction of the LHS non-terminal and its expansion by that production. The modifications to the sentences caused by the rule applications are propagated to all the instances of sentences in the example sets of other productions. The rule is then included in the set of learned transformation rules.

This rule learning process is repeated until all positives of all production rules are covered at that level, then the algorithm moves to the next level.

During testing, transformation rules are applied to a sentence in the order in which they were learned.

5 Experiments

5.1 Methodology

A corpus of formal-language advice paired with natural-language annotation was constructed as follows. From the log files of the RoboCup Coach Competition 2003, we extracted all CLANG advice given by coaches. The extracted advice consists of definitions of rules, conditions, and regions. 560 of these were picked at random and translated into English by one of four annotators. The annotators were instructed to replace identifiers with their definitions as long as the resulting sentence did not become too convoluted. In the same spirit, we encouraged the annotators to use natural phrases for regions whenever possible (e.g. "our half"), rather than coordinate-based representations.

```
Input: (L, F) - Natural language sentences paired with
         their formal representations
         Π - all the productions of the formal language
Output: Transformation rules T for all the productions in Π
Function learnAllRules((L, F), Π)
Parse all the formal representations of F using Π
/* Find positive and negative examples */
For all productions π in Π
  Find sets P(π) and N(π) of positive and
  negtive examples:
    For every natural language sentence s in L
      If parse of s's formal representation uses π then
        include s in P(π), set its count s_c to the
         number of times π is used in the parse
      else include s in N(π)
Divide productions of Π into levels.
/* Learn the rules */
For each level l of the productions
  Let Π_l be the set of all productions at level l
  while not all P(π) are empty for π ∈ Π_l
    R = {learnRule(P(π), N(π)) : π ∈ Π_l}
    Let r be the best rule of R and a rule for production π*
    Compute r's replacement subpattern.
    For all elements p ∈ P(π*)
      While (applicable and p_c > 0)
        apply r to the natural language sentence p,
        decrement p_c by 1
      If (p_c = 0) delete p from P(π*)
      and include it in N(π*)
    Include r in T
return T
```

Figure 4: The complete learning algorithm.

Since long CLANG expressions led to unnatural, confusing English glosses, the 300 shortest CLANG statements were used in the final corpus. This corpus contains definitions of 202 rules, 38 conditions, and 60 regions. The average length of a natural-language gloss is 18.77 words.

To evaluate SILT, we performed 10-fold cross validation. The corpus was split into 10 equal-sized disjoint segments and results were averaged over ten trials. In each trial, a different segment was used as independent test data and the system was trained on the remaining data. In the experiments, the beam width parameter β was set to 5 and the parameter η, for penalizing word gaps, was set to 0.4 based on pilot studies.

A sentence is said to be *completely parsed* if the system transforms it into a single top-level CLANG expression. Learned transformation rules may be unable to completely parse some test sentences. A sentence is said to be *correctly parsed* if it is completely parsed and the CLANG statement produced is semantically equivalent to the correct representation (i.e. an exact match up to

	Precision	Recall
Complete parses	91.1%	37.6%
Parse nodes	98.4%	76.1%

Table 1: Performance on test data

reordering of the arguments of commutative and associative logical operators). With respect to this strict measure of correctness, we measured precision (percentage of completely parsed sentences that are correctly parsed) and recall (percentage of sentences that are correctly parsed).

Since complete correctness gives no credit to parses which are close to the correct parses, we also use a measure which gives credit to such parses. This measure evaluates the overlap between the nodes in the output parse and the correct parse. A node of a parse is said to match a node in another parse if both the nodes and their children have the same labels. A node match thus indicates that same production was used for expanding them in both the parses. By counting the number of matched nodes, we measured precision (percentage of nodes in the output parse that matched some node in the correct parse) and recall (percentage of nodes in the correct parse that matched some node in the output parse).

5.2 Results

Table 1 shows the complete as well as partially correct versions of precision and recall for 10-fold cross validation. As can be seen from precision of completely generated parses, if the system completely parses a sentence, the representation is perfectly correct over 90% of the time. This is important for our application because if the system generates an incorrect complete parse, then incorrect advice will be passed to the coachable players, which may worsen their performance. It is better to fail to produce a complete parse, in which case the players will not receive any advice. The high scores for node matches indicate that, even when the system fails to produce a completely correct parse, it usually produces a representation that is close to the correct one.

Table 2 shows the system's performance when tested on the *training* data. The system generates complete and correct parses for only about half of the training sentences. This shows that there

	Precision	Recall
Complete parses	91.3%	49.3%
Parse nodes	99.0%	79.4%

Table 2: Performance on training data

is still significant room for improving the training algorithm. SILT fails to completely parse training sentences mostly because, frequently, after applying an imperfect rule at one level, the resulting sentence does not match the higher level rule that will lead to a complete parse. This requires coordination between rule learning for productions at different levels, possibly by back-tracking when reaching an impasse at higher levels and learning different rules at lower levels to avoid the impasse.

6 Future Work

As noted in the Results section, SILT's performance could improve by having a better coordination between rule learning for productions at different levels. We also plan to test this system on other domains (Zelle and Mooney, 1996; Tang and Mooney, 2000). SILT does not use any knowledge of the English language. It would be interesting to incorporate knowledge about phrasal structures in the natural language when inducing patterns. One way to do this would be using an external syntactic parser to obtain the phrasal structure of sentences. A better way would be integrating syntactic and semantic analysis by pairing each production in the syntactic grammar with a compositional semantic function that produces a semantic form for a phrase given semantic forms for its subphrases (Norvig, 1992). These are left for future work.

7 Conclusions

We presented a new approach, SILT, for mapping natural language sentences to their semantic representations using transformation rules. The system learns these transformation rules from the training data using a bottom-up rule induction algorithm based on the generalization of sentences. SILT was evaluated for the task of mapping natural-language RoboCup coaching instructions into formal CLANG expressions. The system could output complete semantic parses for a reasonable fraction of novel sentences. These complete semantic parses were correct over 90% of the time. For the cases where the system failed to output complete parses, the partial parses were mostly close to the correct parses.

Acknowledgements

We would like to thank Gregory Kuhlmann for his help in annotating the corpus. This research was supported by Defense Advanced Research Projects Agency under grant HR0011-04-1-007. The second author was supported by an MCD Fellowship from the University of Texas at Austin.

References

Eric Brill. 1995. Transformation-based error-driven learning and natural language processing: A case study in part-of-speech tagging. *Computational Linguistics*, 21(4):543–565.

Razvan Bunescu, Ruifang Ge, Rohit J. Kate, Edward M. Marcotte, Raymond J. Mooney, Arun Kumar Ramani, and Yuk Wah Wong. 2004. Comparative experiments on learning information extractors for proteins and their interactions. *Artificial Intelligence in Medicine*. To appear.

Mao Chen, Ehsan Foroughi, Fredrik Heintz, Spiros Kapetanakis, Kostas Kostiadis, Johan Kummeneje, Itsuki Noda, Oliver Obst, Patrick Riley, Timo Steffens, Yi Wang, and Xiang Yin. 2003. Users manual: RoboCup soccer server manual for soccer server version 7.07 and later. Available at `http://sourceforge.net/projects/sserver/`.

Luc De Raedt, Nada Lavrac, and Saso Dzeroski. 1993. Multiple predicate learning. In *Proc. of 13th Intl. Joint Conf. on Artificial Intelligence (IJCAI-93)*, pages 1037–1042, Chambery, France.

Stephen Muggleton and C. Feng. 1990. Efficient induction of logic programs. In *Proc. of 1st Conf. on Algorithmic Learning Theory*, Tokyo, Japan. Ohmsha.

Peter Norvig. 1992. *Paradigms of Artificial Intelligence Programming: Case Studies in Common Lisp*. Morgan Kaufmann, San Mateo, CA.

Lappoon R. Tang and Raymond J. Mooney. 2000. Automated construction of database interfaces: Integrating statistical and relational learning for semantic parsing. In *Proc. of the Joint SIGDAT Conf. on Empirical Methods in Natural Language Processing and Very Large Corpora(EMNLP/VLC-2000)*, pages 133–141, Hong Kong, October.

William A. Woods. 1977. Lunar rocks in natural English: Explorations in natural language question answering. In Antonio Zampoli, editor, *Linguistic Structures Processing*. Elsevier North-Holland, New York.

John M. Zelle and Raymond J. Mooney. 1996. Learning to parse database queries using inductive logic programming. In *Proc. of 13th Natl. Conf. on Artificial Intelligence (AAAI-96)*, pages 1050–1055, Portland, OR, August.