Leaning to Parse Database Queries Using Inductive Logic Programming

John M. Zelle and Raymond J. Mooney

Presented by Lena Dankin
Plan

• Task definition
• General background:
  – Prolog
  – Shift Reduce Parsing
• CHILL parser
• ILP (Inductive logic programming)
  – CHILLIN algorithm
• Experiments and Results
Plan

• Task definition
• General background:
  – Prolog
  – Shift Reduce Parsing
• CHILL parser
• ILP (Inductive logic programming)
  – CHILLIN algorithm
• Experiments and Results
Task Definition

• Executable semantic parsing:
  – Natural language $\rightarrow$ Executable DB query

How many people live in Iowa?
  $\Rightarrow$ answer(P, (population(S,P), equal(S,stateid(iowa))))).

What is the capital of the state with the largest population?
  $\Rightarrow$ answer(C, (capital(S,C), largest(P, (state(S),
population(S,P))))).
Task Definition

• Development of the database application requires two components:
  – A framework for parsing the natural language query into the logical query representations
  – A specific query language for the our database (domain specific)
Geobase – in short

• A database in USA geography
• contains about 800 **Prolog** facts asserting relational tables for basic information about U.S. states, including:
  – Population
  – Area
  – capital city
  – neighboring states
  – major rivers
  – etc
Plan

• Task definition
• General background:
  – Prolog
  – Shift Reduce Parsing
• CHILL parser
• ILP (Inductive logic programming)
  – CHILLIN algorithm
• Experiments and Results
Prolog

• Prolog is a logic programming language.
• Prolog consists of a series of rules and facts.
• A program is run by presenting some query and checking if it can be proved against these known rules and facts.
Prolog: hands on

• A prolog rule base example:

```
facts

mother_child(trude, sally).
father_child(tom, sally).
father_child(tom, erica).
father_child(mike, tom).

rules

sibling(X, Y) :- parent_child(Z, X), parent_child(Z, Y).
parent_child(X, Y) :- father_child(X, Y).
parent_child(X, Y) :- mother_child(X, Y).
```

From: https://en.wikipedia.org/wiki/Prolog#Rules_and_facts
Prolog: hands on

• Prolog queries:

?- mother_child(trude, tom).
false.

?- sibling(sally, erica).
true.

?- mother_child(X, sally).
X = trude.

?- sibling(sally, X).
X = sally;
X = erica;
X = sally.

Rule base:

mother_child(trude, sally).
father_child(tom, sally).
father_child(tom, erica).
father_child(mike, tom).

sibling(X, Y) :- parent_child(Z, X),
              parent_child(Z, Y).

parent_child(X, Y) :- father_child(X, Y).
parent_child(X, Y) :- mother_child(X, Y).
Prolog: hands on

- List notation with *head* and *tail*

```prolog
?- [1,2|X] = [1,2,3,4,5].
X = [3, 4, 5]
```

Will be used later!
• Now that we know Prolog...

**Geobase predicates:**

<table>
<thead>
<tr>
<th>Predicate</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>state</td>
<td>(name, abbreviation, capital, population, area, state_number, city1, city2, city3, city4)</td>
</tr>
<tr>
<td>city</td>
<td>(state, state_abbreviation, name, population)</td>
</tr>
<tr>
<td>river</td>
<td>(name, length, [states through which it flows])</td>
</tr>
<tr>
<td>border</td>
<td>(state, state_abbreviation, [states that border it])</td>
</tr>
<tr>
<td>highlow</td>
<td>(state, state_abbreviation, highest_point, highest_elevation, lowest_point, lowest_elevation)</td>
</tr>
<tr>
<td>mountain</td>
<td>(state, state_abbreviation, name, height)</td>
</tr>
<tr>
<td>road</td>
<td>(number, [states it passes through])</td>
</tr>
<tr>
<td>lake</td>
<td>(name, area, [states it is in])</td>
</tr>
</tbody>
</table>

Database available at: https://www.cs.utexas.edu/users/ml/nldata/geoquery.html
• In order to express interesting questions about geography, we need a query language having a vocabulary sufficient for expressing interesting questions about geography

• GeoQuery – a query language used for our task (all predicates are, naturally, implemented in Prolog)
GeoQuery

• Predicated for basic objects:

<table>
<thead>
<tr>
<th>Type</th>
<th>Form</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>country</td>
<td>countryid(Name)</td>
<td>countryid(usa)</td>
</tr>
<tr>
<td>city</td>
<td>cityid(Name, State)</td>
<td>cityid(austin, tx)</td>
</tr>
<tr>
<td>state</td>
<td>stateid(Name)</td>
<td>stateid(texas)</td>
</tr>
<tr>
<td>river</td>
<td>riverid(Name)</td>
<td>riverid(colorado)</td>
</tr>
<tr>
<td>place</td>
<td>placeid(Name)</td>
<td>placeid(pacific)</td>
</tr>
</tbody>
</table>
GeoQuery

• Predicated for basic relations:

<table>
<thead>
<tr>
<th>Form</th>
<th>Predicate</th>
</tr>
</thead>
<tbody>
<tr>
<td>capital(C)</td>
<td>C is a capital (city).</td>
</tr>
<tr>
<td>city(C)</td>
<td>C is a city.</td>
</tr>
<tr>
<td>major(X)</td>
<td>X is major.</td>
</tr>
<tr>
<td>place(P)</td>
<td>P is a place.</td>
</tr>
<tr>
<td>river(R)</td>
<td>R is a river.</td>
</tr>
<tr>
<td>state(S)</td>
<td>S is a state.</td>
</tr>
<tr>
<td>capital(C)</td>
<td>C is a capital (city).</td>
</tr>
<tr>
<td>area(S,A)</td>
<td>The area of S is A.</td>
</tr>
<tr>
<td>capital(S,C)</td>
<td>The capital of S is C.</td>
</tr>
<tr>
<td>equal(V,C)</td>
<td>variable V is ground term C.</td>
</tr>
<tr>
<td>density(S,D)</td>
<td>The (population) density of S is P</td>
</tr>
<tr>
<td>elevation(P,E)</td>
<td>The elevation of P is E.</td>
</tr>
<tr>
<td>high_point(S,P)</td>
<td>The highest point of S is P</td>
</tr>
<tr>
<td>higher(P1,P2)</td>
<td>P1’s elevation is greater than P2’s.</td>
</tr>
<tr>
<td>loc(X,Y)</td>
<td>X is located in Y.</td>
</tr>
<tr>
<td>low_point(S,P)</td>
<td>The lowest point of S is P</td>
</tr>
<tr>
<td>len(R,L)</td>
<td>The length of R is L.</td>
</tr>
<tr>
<td>next_to(S1,S2)</td>
<td>S1 is next to S2.</td>
</tr>
<tr>
<td>size(X,Y)</td>
<td>The size of X is Y.</td>
</tr>
<tr>
<td>traverse(R,S)</td>
<td>R traverses S.</td>
</tr>
</tbody>
</table>
Meta predicates - distinguished in that they take completely-formed conjunctive goals as one of their arguments

<table>
<thead>
<tr>
<th>Form</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>answer(V,Goal)</td>
<td>$V$ is the variable of interest in $Goal$.</td>
</tr>
<tr>
<td>largest(V,Goal)</td>
<td>$Goal$ produces only the solution that maximizes the size of $V$.</td>
</tr>
<tr>
<td>smallest(V,Goal)</td>
<td>Analogous to largest.</td>
</tr>
<tr>
<td>highest(V,Goal)</td>
<td>Like largest (with elevation).</td>
</tr>
<tr>
<td>lowest(V,Goal)</td>
<td>Analogous to highest.</td>
</tr>
<tr>
<td>longest(V,Goal)</td>
<td>Like largest (with length).</td>
</tr>
<tr>
<td>shortest(V,Goal)</td>
<td>Analogous to longest.</td>
</tr>
<tr>
<td>count(D,Goal,C)</td>
<td>$C$ is count of unique bindings for $D$ that satisfy $Goal$.</td>
</tr>
<tr>
<td>most(X,D,Goal)</td>
<td>$Goal$ produces only the $X$ that maximizes the count of $D$.</td>
</tr>
<tr>
<td>fewest(X,D,Goal)</td>
<td>Analogous to most.</td>
</tr>
</tbody>
</table>
GeoQuery

• How many people live in Iowa?
• => answer(P, (population(S,P), equal(S,stateid(iowa))))).

A GeoQuery query to be executed on Geobase. The variable P holds the answer
Shift Reduce Parser

• **Goal of parser:**
  
  **Input:**
  Grammar, linear input text

  **Output:**
  The grammatical structure of linear input text

• **Bottom-up parser:** build a derivation by working from the input back toward the start symbol
  
  • Builds parse tree from leaves to root
  • Builds reverse rightmost derivation
A shift-reduce parser uses two data structures:
- An input buffer to store words of a sentence that have not yet been examined.
- A stack which stores information concerning sentence constituents that have been recognized so far.

Initially, the stack is empty, and the input buffer contains all of the words of a sentence to be processed.

The process of parsing a sentence is a search problem. The parser must find a sequence of operators that transforms the initial state into a final representation.
Shift-Reduce parsing

• For the given grammar we define:
  – Terminal: num, id, +, *
  – Non terminals: S, E

A handle:
  – matches the rhs (right hand side) of some rule
  – allows further reductions back to the start symbol

1. S → E
2. E → E + E
3. E → E * E
4. E → num
5. E → id
Shift-reduce parser

• Two questions
  1. Have we reached the end of handles and how long is the handle?
  2. Which non-terminal does the handle reduce to?

• We use tables to answer the questions
  – ACTION table
  – GOTO table
Shift-Reduce parsing

• A shift-reduce parser has 4 actions:
  – **Shift** -- next input token is shifted onto the stack
  – **Reduce** -- handle is at top of stack
    • pop handle
    • push appropriate Lhs
  – **Accept** -- stop parsing & report success
  – **Error** -- call error reporting/recovery routine

from https://www.cs.northwestern.edu/academics/courses/322/notes/05.ppt
Example: Shift-reduce parsing

Grammar:
1. S → E
2. E → E + E
3. E → E * E
4. E → num
5. E → id

Input to parse: id₁ + num * id₂

Handles: underlined

<table>
<thead>
<tr>
<th>STACK</th>
<th>ACTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>$</td>
<td>Shift</td>
</tr>
<tr>
<td>$ id₁</td>
<td>Reduce (rule 5)</td>
</tr>
<tr>
<td>$ E</td>
<td>Shift</td>
</tr>
<tr>
<td>$ E +</td>
<td>Shift</td>
</tr>
<tr>
<td>$ E + num</td>
<td>Reduce (rule 4)</td>
</tr>
<tr>
<td>$ E + E</td>
<td>Shift</td>
</tr>
<tr>
<td>$ E + E *</td>
<td>Shift</td>
</tr>
<tr>
<td>$ E + E * id₂</td>
<td>Reduce (rule 5)</td>
</tr>
<tr>
<td>$ E + E * E</td>
<td>Reduce (rule 3)</td>
</tr>
<tr>
<td>$ E + E</td>
<td>Reduce (rule 2)</td>
</tr>
<tr>
<td>$ E</td>
<td>Reduce (rule 1)</td>
</tr>
<tr>
<td>$ S</td>
<td>Accept</td>
</tr>
</tbody>
</table>

from https://www.cs.northwestern.edu/academics/courses/322/notes/05.ppt
Plan

• Task definition
• General background:
  – Prolog
  – Shift Reduce Parsing
• CHILL parser
• ILP (Inductive logic programming)
  – CHILLIN algorithm
• Experiments and Results
CHILL

• CHILL (Constructive Heuristics Induction for Language Learning):

  Input:
  a set of training instances consisting of sentences paired with the desired parses

  Output:
  a deterministic shift-reduce parser in Prolog which maps sentences into parses
Training Examples

• \(<\text{Sentence, Parse}>: \)
  – For example:
    <The man ate the pasta,
    \[[\text{ate}, \text{obj:}[[\text{pasta, det:the}], \text{agt:}[[\text{man, det:the}]]]] >
CHILL

<Sentence, Representation> → Training Examples → Parsing Operator Generator → Overly-General Parser

Example Analysis → Control Examples

Control Rule Induction → Control Rules

Program Specialization → Final Parser

Prolog
The training examples are analyzed to formulate an overly-general shift-reduce parser that is capable of producing parses from sentences.

```
parse(S, Parse) :- parse([], S, [Parse], []).
parse(Stack, Input, Stack, Input).
parse(Stack0, In0, Stack, In) :-
    op(Stack0, In0, Stack1, In1), parse(Stack1, In1, Stack, In).
    
    op([[Top,Second|Rest],In,[NewTop|Rest],In]) :-
        reduce(Top,agt,Second,NewTop).
    op([[Top,Second|Rest],In,[NewTop|Rest],In]) :-
        reduce(Top,det,Second,NewTop).
    op([[Top,Second|Rest],In,[NewTop|Rest],In]) :-
        reduce(Second,obj,Top,NewTop).
    op(Stack,[Word|Words],[Word|Stack],Words).  % Shift operation.
reduce([[Head|Slots],Role,Filler,[Head,Role:Filler|Slots]]) :- !.
reduce(BareHead,Role,Filler,Result) :-
    reduce([BareHead],Role,Filler,Result).
```
CHILL
Example Analysis

• For Each operator we want to generate two kind of examples:
  – Correct control examples: in what situation we should apply this operator
  – Incorrect control examples: in what situation we shouldn’t apply this operator.
Example Analysis

Positive control examples:
for each example of the training instance, parse states to which the operator should be applied.

for example: the operation that reduced to \textit{agt}:

\[
\text{op}(\text{[ate, [man, det:the]], [the, pasta], A, B}).
\]

- Current stack
- Input buffer
- New stack
- New input buffer
**Example Analysis**

Negative control examples:
all contexts where this operator was not applied.

We assume that the set of training examples includes a pair for every correct parsing for each unique sentence appearing in the set.

For example (*agt* reduction operator):

\[
\text{op}([\text{man}, \text{the}], [\text{ate}, \text{the}, \text{pasta}], A, B)
\]

\[
\text{op}([\text{the}, [\text{ate}, \text{agt: [man, det: the]]}], [\text{pasta}], A, B)
\]
CHILL

Later!
Program Specialization

• “fold” the control information back into the overly-general parser. Each operator clause in the overly-general parser is modified by adding the learned control knowledge so that attempts to use the operator inappropriately fail immediately.
CHILL for GeoQuery

• **introduce** action:
  – The word *capital* might cause the *capital/2* predicate to be pushed on the stack

• **co-reference** action:
  – variables may be unified with variables appearing in other stack items.
  – For example, the first argument of the *capital/2* structure may be unified with the argument of a previously introduced *state/1* predicate

• **conjoin** action:
  – a stack item may be embedded into the argument of another stack item to form conjunctive goals inside of meta-predicates
CHILL for GeoQuery

• Parsing example for the query:
  What is the capital of Texas?

Assume:
• A lexicon that maps:
  ‘capital’ to capital(_)
  ‘of’ to loc(_)
  ‘Texas’ to const(_, stateid(texas))
• Each predicate on the parse stack has an attached buffer to hold the context in which it was introduced
## CHILL for GeoQuery

- **What is the capital of Texas?**

<table>
<thead>
<tr>
<th>Parse Stack</th>
<th>Input Buffer</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>[answer(_, _):[]]</code></td>
<td><code>[what, is, the, capital, of, texas, ?]</code></td>
<td>3* SHIFT</td>
</tr>
<tr>
<td><code>[answer(_, _):[the, is, what]]</code></td>
<td><code>[capital, of, texas, ?]</code></td>
<td>INTRODUCE</td>
</tr>
<tr>
<td><code>[capital(_):[], answer(_, _):[the, is, what]]</code></td>
<td><code>[capital, of, texas, ?]</code></td>
<td>COREF</td>
</tr>
<tr>
<td><code>[capital(C):[], answer(C, _):[the, is, what]]</code></td>
<td><code>[capital, of, texas, ?]</code></td>
<td>SHIFT</td>
</tr>
<tr>
<td><code>[capital(C):[capital], answer(C, _):[the, is, what]]</code></td>
<td><code>[of, texas, ?]</code></td>
<td>INTRODUCE</td>
</tr>
<tr>
<td><code>[loc(, _):[], capital(C):[capital], answer(C, _):[the, is, what]]</code></td>
<td><code>[of, texas, ?]</code></td>
<td>COREF</td>
</tr>
<tr>
<td><code>[loc(C, _):[], capital(C):[capital], answer(C, _):[the, is, what]]</code></td>
<td><code>[of, texas, ?]</code></td>
<td>SHIFT</td>
</tr>
</tbody>
</table>
CHILL for GeoQuery

• What is the capital of Texas?

<table>
<thead>
<tr>
<th>Parse Stack</th>
<th>Input Buffer</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>[const(S,stateid(texas))]:[],</td>
<td>[texas, ?]</td>
<td>CONJ</td>
</tr>
<tr>
<td>loc(C,S):[of], capital(C):[capital], answer(C,_):[the,is,what]]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>[loc(C,S):[of], capital(C):[capital], answer(C, const(S,stateid(texas)))]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>:[the,is,what]]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>[capital(C):[capital],</td>
<td></td>
<td></td>
</tr>
<tr>
<td>answer(C, (loc(C,S),const(S,stateid(texas)))))]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>:[the,is,what]]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>[answer(C, (capital(C),loc(C,S),const(S,stateid(texas))))]:[the,is,what]]</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Plan

• Task definition
• General background:
  – Prolog
  – Shift Reduce Parsing
• CHILL parser
• ILP (Inductive logic programming)
  – CHILLIN algorithm
• Experiments and Results
First order logic

Consist of:

1. The quantifier symbols $\forall$ and $\exists$
2. $\land$ for conjunction, $\lor$ for disjunction, $\rightarrow$ for implication, $\leftrightarrow$ for biconditional, $\neg$ for negation
3. Variables
4. An equality symbol (sometimes, identity symbol) =
Inductive logic programming

Positive examples + Negative examples + Background knowledge

⇒

hypothesis that entail all positive examples and none of the negatives
ILP – formal definitions

• Given
  – a logic program $B$ representing background knowledge
  – a set of positive examples $E^+$
  – a set of negative examples $E^-$

• Find hypothesis $H$ such that:

  1. $B \cup H \models e$ for every $e \in E^+$. (complete)
  2. $B \cup H \not\models f$ for every $f \in E^-$.
  3. $B \cup H$ is consistent.

Assume that $B \not\models e$ for some $e \in E^+$. 
ILP – List membership example

• For Example: consider learning the concept of list membership.
  – Positive examples:
    member(1, [1,2]),
    member(2, [1,2]),
    member(1, [3,1]), etc
  – Negative examples:
    member(1, []),
    member(2, [1,3]), etc
  – Background:
    The predicate *components/3* which decomposes a list into its component head and tail.
ILP – List membership example

• Given the examples + background, we hope to learn the correct definition of member, namely:

\[
\text{member}(X, \text{List}) :- \text{components}(\text{List}, X, \text{Tail}).
\]

\[
\text{member}(X, \text{List}) :- \text{components}(\text{List}, \text{Head}, \text{Tail}), \text{member}(X, \text{Tail}).
\]
CHILLIN

CHILLIN (CHILL inductive algorithm)

**Input:**
- A set of positive and negative examples of a concept, expressed as a ground facts
- A set of background predicates, expresses as definite clauses

**Output:**
definite clause concept definition which entails the positives examples, but not the negative.
CHILLIN

1. The algorithm starts with a most specific definition (the set of all positive examples)
2. Then, it introduces a generalization which make the definition more compact
   – Compactness: measured by the sum of the program’s clauses
1. We start with a most specific definition: a set of all the positive examples

CHILLIN

DEF := \{E : \text{true} \mid E \in \text{Pos}\}

Repeat

PAIRS := a sampling of pairs of clauses from DEF

GENS := \{G \mid G = \text{build\_gen}(C_i, C_j, \text{DEF}, \text{Pos}, \text{Neg}) for \langle C_i, C_j \rangle \in \text{PAIRS}\}

G := Clause in GENS yielding most compaction

DEF := (DEF - (Clauses subsumed by G)) \cup G

Until no further compaction
CHILLIN

DEF := \{ E :- true \mid E \in Pos \}

Repeat
  PAIRS := a sampling of pairs of clauses from DEF
  GENS := \{ G \mid G = build\_gen(C_i, C_j, DEF, Pos, Neg) \}
    for \langle C_i, C_j \rangle \in PAIRS \}

G := Clause in GENS yielding most compaction
DEF := (DEF - (Clauses subsumed by G)) \cup G

Until no further compaction

2. A search for more general definition is carried out
3. We sample 15 pairs of clauses from DEF

\[
\begin{align*}
\text{DEF} & := \{ E : \text{true} \mid E \in \text{Pos}\} \\
\text{Repeat} \\
\text{PAIRS} & := \text{a sampling of pairs of clauses from DEF} \\
\text{GENS} & := \{ G \mid G = \text{build\_gen}(C_i, C_j, \text{DEF, Pos, Neg}) \} \\
& \quad \text{for } \langle C_i, C_j \rangle \in \text{PAIRS} \\
G & := \text{Clause in GENS yielding most compaction} \\
\text{DEF} & := (\text{DEF} - (\text{Clauses subsumed by } G)) \cup G \\
\text{Until} & \text{ no further compaction}
\end{align*}
\]
DEF := \{ E :: true \mid E \in \text{Pos} \}

Repeat

PAIRS := a sampling of pairs of clauses from DEF

GENS := \{ G \mid G = \text{build}\_\text{gen}(C_i,C_j,\text{DEF},\text{Pos},\text{Neg}) \}

for \( \langle C_i, C_j \rangle \in \text{PAIRS} \}

G := \text{Clause in GENS yielding most compaction}

DEF := (DEF \backslash (\text{Clauses subsumed by } G)) \cup G

Until no further compaction

4. For each pair we build a generalization

details in a few slides!
CHILLIN

DEF := \{ E : true \mid E \in Pos \}

Repeat

PAIRS := a sampling of pairs of clauses from DEF
GENS := \{ G \mid G = \text{build}\_\text{gen}(C_i,C_j,DEF,Pos,Neg) for \langle C_i, C_j \rangle \in PAIRS \}

G := Clause in GENS yielding most compaction
DEF := (DEF - (Clauses subsumed by G)) \cup G

Until no further compaction

5. Choose the best generalization (using the compaction measure)
CHILLIN

\[
\text{DEF} := \{E : \text{true} \mid E \in \text{Pos}\}
\]

Repeat

PAIRS := a sampling of pairs of clauses from DEF
GENS := \{G \mid G = \text{build-gen}(C_i, C_j, \text{DEF}, \text{Pos}, \text{Neg})
\text{for } \langle C_i, C_j \rangle \in \text{PAIRS}\}

G := \text{Clause in GENS yielding most compaction}
DEF := (\text{DEF} - (\text{Clauses subsumed by G})) \cup G

Until no further compaction

5. The reduction:
   - add G to DEF
   - Prove all positive examples with DEF, remove all clauses that were not used in any of the proofs.
CHILLIN - Generalizations

• The generalization process consists of three steps:
  1. Introduce a simple generalization of the input clauses
  2. If this generalization covers no negative examples, it is returned
  3. Else (the generalization is too general) try:
     a. Adding literals to the generalization
     b. Call a routine which invents a new predicate, so that no negative examples are covered.
CHILLIN - Generalizations

• For example:

  given the positive clauses:
  • member(1, [1, 2, 3])
  • member(3, [3])

  The least general generalization would be:
  
  member(A, [A|B])

  Which is a valid generalization (no need for stage 3)
Plan

• Task definition
• General background:
  – Prolog
  – Shift Reduce Parsing
• CHILL parser
• ILP (Inductive logic programming)
  – CHILLIN algorithm
• Experiments and Results
Experiment setting

• A corpus of 250 sentences was gathered by submitting a questionnaire to 50 uninformed subjects.

• For evaluation purposes, the corpus was split into:
  -- training sets of 225 examples
  -- with the remaining 25 held-out for testing.

• Overall 10 folds
Experiment setting

Baseline:
- Geobase, uses a semantic-base parser which scans for words corresponding to the entities and relationships encoded in the database.
- The system attempts to match sequences of entities and associations in sentences with an entity-association network describing the schemas present in the database.
Experiment result
Questions?
References

• Using Inductive Logic Programming to Automate the Construction of Natural Language Parsers, John M. Zelle, PHD Thesis

• Learning to Parse Database Queries using Inductive Logic Programming, John M. Zelle and Raymond J. Mooney

• Learning Semantic Grammars with Constructive Inductive Logic Programming, John M. Zelle and Raymond J. Mooney

• Using Multiple Clause Constructors in Inductive Logic Programming for Semantic Parsing, Lappoon R. Tang and Raymond J. Mooney
References

• https://en.wikipedia.org/wiki/Inductive_logic_programming

• https://en.wikipedia.org/wiki/Prolog

• lyle.smu.edu/~mhd/2353sp09/prolog.pptx

• https://www.cs.northwestern.edu/academics/courses/322/notes/05.ppt