SLD-Resolution and Fuzzy Logic

Olaeriu Vlad Mihai

January 2025

1 SLD-Resolution

Task: define your own rules and questions such that the chaining is at least 3 size long. A program interface should address the questions to the user. The knowledge must be expressed as *positive Horn clauses*. The knowledge base, expressed as a list of lists, will be read from a file.

1.1 Rules

1. If the dog was more than 20kg, it is big.
2. If the dog was less than 20kg, it is small.
3. If the dog is intelligent and big, then it is also protective.
4. If the dog is protective and trainable, then it is German Sheperd.
5. If the dog is big and playful, then it Rottweiler.
6. If it is small and playful, then it is a Beagle.

1.2 Clauses

1. intelligent $\land$ big $\rightarrow$ protective
2. protective $\land$ trainable $\rightarrow$ germanSheperd
3. big $\land$ playful $\rightarrow$ rottweiler
4. small $\land$ playful $\rightarrow$ beagle

1.3 Horn clauses

1. $\neg$intelligent $\lor$ $\neg$big $\lor$ protective
2. $\neg$protective $\lor$ $\neg$trainable $\lor$ germanSheperd
3. $\neg$big $\lor$ $\neg$playful $\lor$ rottweiler
4. $\neg$small $\lor$ $\neg$playful $\lor$ beagle

1.4 Interface

The scope is to recommend a dog breed depending on the preferences of the user. In the interface, the following fields need to be completed by the user:

1. Weight: from 0 to 35 kg
2. Intelligent: yes / no
3. Playful: yes / no
4. Trainable: yes / no

After the user completed these fields, the weight will be transformed into the *Big* or *Small* predicate if matching, and together with the positive answers from the rest of the questions will be used in the knowledge base to find which dog is a good fit for the user. Three different questions will be solved by the algorithm: German Sheperd, Rottweiler, Beagle.

Based on the *answers* and the *knowledge base* is the X breed good for the user?

1.5 Example

In the case when the answers received from the users are:

- Weight: 30kg
- Intelligent: yes
- Playful: no
- Trainable: yes,

which would be represented by 3 different predicates: *big*, *intelligent*, *trainable*, the knowledge base would look like:

1. $\neg$intelligent $\vee$ $\neg$big $\vee$ protective
2. $\neg$protective $\vee$ $\neg$trainable $\vee$ germanSheperd
3. $\neg$big $\vee$ $\neg$playful $\vee$ rottweiler
4. $\neg$small $\vee$ $\neg$playful $\vee$ beagle
5. big
6. intelligent
7. trainable

For the question *germanShepherd*, using the forward chaining for the SLD-resolution, we will have:

- [big]: has no negative atoms - *big* solved
- [intelligent]: has no negative atoms - *intelligent* solved
- [trainable]: has no negative atoms - *trainable* solved
- [$\neg$ intelligent, $\neg$ big, protective]: intelligent is solved, big is solved - *protective* solved
- [$\neg$ protective, $\neg$ trainable, germanShepherd]: protective is solved, trainable is solved - *germanShepherd* solved
- Returns **YES**

For the question *germanShepherd*, using the backward chaining for the SLD-resolution, we will have:

- [$\neg$germanShepherd] initial goals, negated
- [$\neg$protective, $\neg$trainable]: resolution with [$\neg$protective, $\neg$trainable, germanShepherd]
- [$\neg$intelligent, $\neg$big, $\neg$trainable]: resolution with [$\neg$intelligent, $\neg$big, protective]
- [$\neg$big, $\neg$trainable]: resolution with [intelligent]
- [$\neg$trainable]: resolution with [big]
- []: resolution with [trainable]
- Returns **YES**

For the question *rottweiler*, using the backward chaining for the SLD-resolution, we will have:

- [¬rottweiler] initial goals, negated
- [¬big, ¬playful]: resolution with [¬big, ¬playful, rottweiler]
- [¬playful]: resolution with [big]
- [¬playful]: algorithm stop because no clause can resolve with atom ¬playful
- Returns **NO**

2 Fuzzy Logic

Task: define your own rules and degree curves for each predicate involved in the rules. The program interface should ask the use what are his/her ratings for each attribute that will be defining the predicates.

For my knowledge base, I have chosen multiple rules that aim to predict the price of a car depending on some of its properties, which will be completed by the user in the interface.

2.1 Rules

1. If the car age is old and the consumption is high, then its price is low.
2. If the car age is new and the mileage is high, then its price is low.
3. If the car age is old and the mileage is low, then its price is medium.
4. If the car age is new and the consumption is high, then its price is medium.
5. If the car age is middle and the consumption is low, then the price is high.
6. If the car mileage is low or the consumption is low, then the price is high.

2.2 Degree curves

From the rules above, there are 4 main variables **age**, **consumption**, **mileage** and **price**. Each variables has its own 3 predicates:

- age: old, middle, new
- mileage: low, medium, high
- consumption: low, medium, high
- price: low, medium, high

The degree curves for each predicate can be examined in Figure 1.

2.3 Interface

In the interface, the users needs to complete the inputs for the following variables: *age*, *mileage*, *consumption*. Based on the answers, a price estimation will be given to the car using the rules and the degree curves defined previously.

Possible values:

- age: 0 to 15 years
- mileage: 0 to 200,000km
- consumption: 0 to 15L/100KM
- price: 0 to 15 thousands of dollars

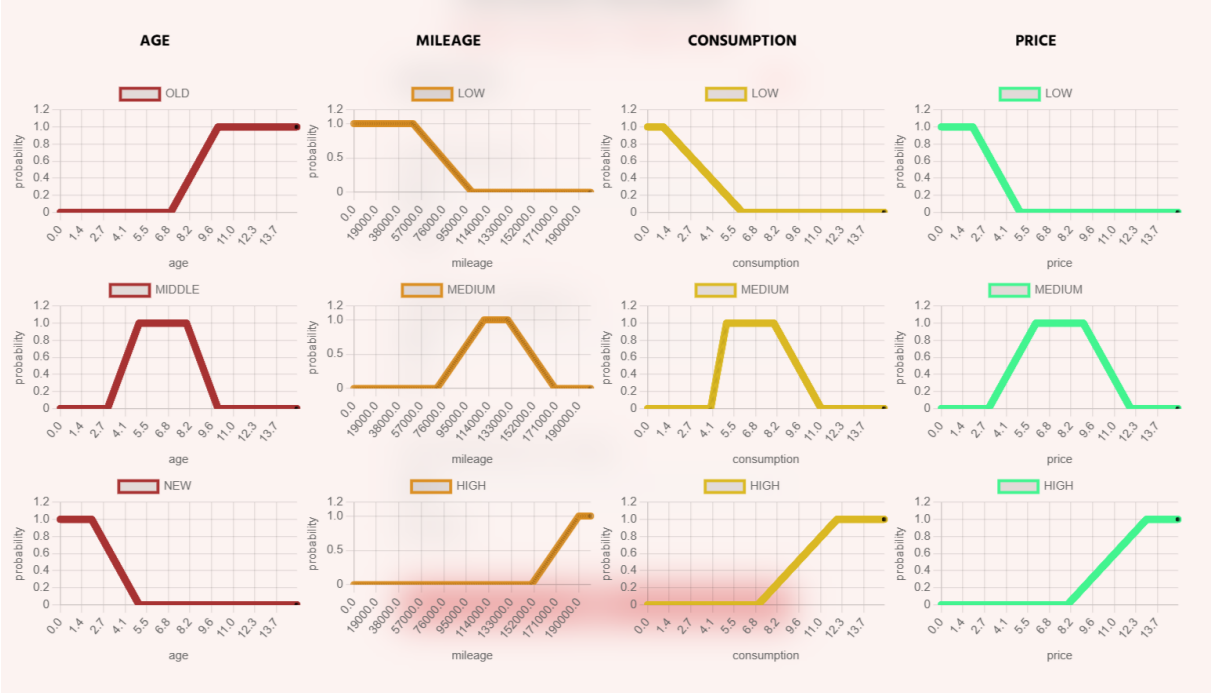

Figure 1: Degree curves for each predicate

2.4 Algorithm

After the input is received from the user, the antecedents need to be computed and aggregated: for conjunctions, a **minimum** is applied between the degree of all antecedents of a rule, respectively a **maximum** for disjunctions.

Each rule has a consequence. The degree of the rule was computed using a **minimum** between the antecedent's degree and the consequence's degree. (the result needs to take into account the applicability of each rule, meaning that it is constrained by the antecedents)

This process is repeated for every rule, and then all the results are aggregated through a **maximum** operator, and the outcome is the degree curve of the price of the car, given its age, consumption, and mileage.

The final price predicted was extracted using the centroid value of the degree curve. Because the degree curve is a continuous function, only a few values from the function, integers from 0 to 15, were used to compute the centroid.

$$\text{price} = \frac{\sum_{i=0}^{15} x_i * y_i}{\sum_{i=0}^{15} y_i}$$

$$x_i = \text{price, equal to i}$$

$$y_i = \text{degree of the price i}$$

3 Code

utils.pl

```
neg(n(X), X) :- !.
neg(X, n(X)).

neg_list([], []).
neg_list([H|T], [HNeg|TNeg]) :- neg(H, HNeg), neg_list(T, TNeg).

is_neg(n(_)).
is_pos(X) :- \+ is_neg(X).
is_opposite(Atom1, Atom2) :- neg(Atom1, NAtom1), NAtom1 = Atom2.
get_positive_atom(Clause, Atom) :- member(Atom, Clause), is_pos(Atom), !.
```

```
eliminate(Target, [Target|T], T).
eliminate(Target, [H|T], [H|Result]) :- eliminate(Target, T, Result).

merge([], L, L).
merge([H|T], L, Result) :- member(H, L), !, merge(T, L, Result).
merge([H|T], L, [H|Result]) :- merge(T, L, Result).

unpack_kb([KB], KB).

interval(Max, Max, [Max]).
interval(Min, Max, [Min|Rest]) :-
    Min < Max,
    NewMin is Min + 1,
    interval(NewMin, Max, Rest).

min(A, B, A) :- A < B, !.
min(_, B, B).

max(A, B, A) :- A > B, !.
max(_, B, B).

max_between_lists([], [], []).
max_between_lists([A|As], [], [A|Rs]) :- max_between_lists(As, [], Rs).
max_between_lists([], [B|Bs], [B|Rs]) :- max_between_lists(Bs, [], Rs).
max_between_lists([A|As], [B|Bs], [R|Rs]) :- max(A, B, R), max_between_lists(As, Bs, Rs).

sum([], 0).
sum([H|T], Sum) :- sum(T, SumRec), Sum is H + SumRec.

sum_between_lists_product([], [], 0).
sum_between_lists_product([A|As], [B|Bs], Sum) :-
    sum_between_lists_product(As, Bs, SumRec),
    Sum is SumRec + A * B.
```

read.pl

```
read_file(Filename, Lines) :-
    open(Filename, read, Stream),
    read_lines(Stream, Lines), !,
    close(Stream).

line_to_atom(Line, Term) :-
    atom_string(Atom, Line),
    atom_to_term(Atom, Term, _).

read_lines(Stream, []) :-
    at_end_of_stream(Stream).
read_lines(Stream, [Atom|Rest]) :-
    \+ at_end_of_stream(Stream),
    read_line_to_string(Stream, Line),
    line_to_atom(Line, Atom),
    read_lines(Stream, Rest).
```

parse.pl

```
process_sentence(Sentece, Result) :- translate(Result, Sentece, []).
```

```
process_sentences([], []).
process_sentences([Sentece|Sentences], [Result|Results]) :-
    process_sentence(Sentece, Result),
    process_sentences(Sentences, Results).

translate([A|B]) --> disjunction(A), [and], translate(B).
translate([A])   --> disjunction(A).

disjunction(A) --> ['('], list_of_options(A).

list_of_options([Element])   --> [Element, ')'].
list_of_options([Element|T]) --> [Element], [or], list_of_options(T).
```

sld-backward.pl

```
:- ["./utils/utils.pl"].

make_new_goals(Clause, Goals, Atom, GoalsNew) :-
    eliminate(Atom, Clause, ClauseUpdated),
    merge(ClauseUpdated, Goals, GoalsNew).

is_clause_resolving(Clause, [Goal|Goals], GoalsNew) :-
    member(CAtom, Clause),
    is_opposite(CAtom, Goal),
    make_new_goals(Clause, Goals, CAtom, GoalsNew).

resolution_backward_helper(_, []) :- !.
resolution_backward_helper(KB, Goals) :-
    member(Clause, KB),
    is_clause_resolving(Clause, Goals, GoalsNew),
    resolution_backward_helper(KB, GoalsNew).
resolution_backward_helper(_, _) :- false.

resolution_backward(KB, Goals, Result) :-
    neg_list(Goals, GoalsNeg),
    (
     resolution_backward_helper(KB, GoalsNeg) ->
         Result = "YES";
         Result = "NO"
    ).
```

sld-forward.pl

```
:- ["./utils/utils.pl", "./utils/parse.pl", "./utils/read.pl"].

get_negative_atoms_as_positive(Clause, Atoms) :-
    findall(NAtom, (
  member(Atom, Clause),
  is_neg(Atom),
  neg(Atom, NAtom)
 ), Atoms).

is_clause_eligible(Clause, Solved, SolvedNew) :-
    get_negative_atoms_as_positive(Clause, NAtoms),
    subset(NAtoms, Solved),
```

```
    get_positive_atom(Clause, PAtom),
    \+ member(PAtom, Solved),
    SolvedNew = [PAtom|Solved].

is_goal_solved([], _).
is_goal_solved([Goal|Goals], Solved) :-
    member(Goal, Solved), is_goal_solved(Goals, Solved).

resolution_forward_helper(_, Goals, Solved) :-
    is_goal_solved(Goals, Solved), !.
resolution_forward_helper(KB, Goals, Solved) :-
    member(Clause, KB),
    is_clause_eligible(Clause, Solved, SolvedNew),
    resolution_forward_helper(KB, Goals, SolvedNew), !.
resolution_forward_helper(_, _, _) :- false.

resolution_forward(KB, Goals, Result) :-
    (
        resolution_forward_helper(KB, Goals, []) ->
            Result = "YES";
            Result = "NO"
    ).
```

degree-curves.pl

```
:- ["./utils/read.pl", "./utils/parse.pl", "./utils/utils.pl"].

new_aged(X, Degree) :- X >= 0, X =< 2, Degree is 1.
new_aged(X, Degree) :- X > 2, X =< 5, Degree is (1 - (X - 2) / 3).
new_aged(X, Degree) :- X > 5, X =< 15, Degree is 0.

middle_aged(X, Degree) :- X >= 0, X =< 3, Degree is 0.
middle_aged(X, Degree) :- X > 3, X < 5, Degree is (3 / 2 - X / 2).
middle_aged(X, Degree) :- X >= 5, X =< 8, Degree is 1.
middle_aged(X, Degree) :- X > 8, X =< 10, Degree is (5 - X / 2).
middle_aged(X, Degree) :- X >= 10, X =< 15, Degree is 0.

old_aged(X, Degree) :- X >= 0, X =< 7, Degree is 0.
old_aged(X, Degree) :- X > 7, X < 10, Degree is (X / 3 - 7 / 3).
old_aged(X, Degree) :- X >= 10, X =< 15, Degree is 1.


low_consum(X, Degree) :- X >= 0, X =< 1, Degree is 1.
low_consum(X, Degree) :- X > 1, X =< 6, Degree is (-1 / 5) * (X - 1) + 1.
low_consum(X, Degree) :- X > 6, X =< 15, Degree is 0.

medium_consum(X, Degree) :- X >= 0, X =< 4, Degree is 0.
medium_consum(X, Degree) :- X > 4, X < 5, Degree is X - 4.
medium_consum(X, Degree) :- X >= 5, X =< 8, Degree is 1.
medium_consum(X, Degree) :- X > 8, X < 11, Degree is (-1 / 3) * (X - 8) + 1.
medium_consum(X, Degree) :- X >= 11, X =< 15, Degree is 0.

high_consum(X, Degree) :- X >= 0, X =< 7, Degree is 0.
high_consum(X, Degree) :- X > 7, X < 12, Degree is (1 / 5) * (X - 7).
high_consum(X, Degree) :- X >= 12, X =< 15, Degree is 1.
```

```
low_mileage(X, Degree) :- X >= 0, X =< 50000, Degree is 1.
low_mileage(X, Degree) :- X > 50000, X =< 100000, Degree is 1 - (X - 50000) / 50000.
low_mileage(X, Degree) :- X > 100000, X =< 200000, Degree is 0.

medium_mileage(X, Degree) :- X >= 0, X =< 70000, Degree is 0.
medium_mileage(X, Degree) :- X > 70000, X =< 110000, Degree is (X - 70000) / 40000.
medium_mileage(X, Degree) :- X >= 110000, X =< 130000, Degree is 1.
medium_mileage(X, Degree) :- X > 130000, X < 170000, Degree is 1 - (X - 130000) / 40000.
medium_mileage(X, Degree) :- X >= 170000, X =< 200000, Degree is 0.

high_mileage(X, Degree) :- X >= 0, X =< 150000, Degree is 0.
high_mileage(X, Degree) :- X > 150000, X < 190000, Degree is (X - 150000) / 40000.
high_mileage(X, Degree) :- X >= 190000, X =< 200000, Degree is 1.


low_price(X, Degree) :- X >= 0, X =< 2, Degree is 1.
low_price(X, Degree) :- X > 2, X =< 5, Degree is 1 - (X - 2) / 3.
low_price(X, Degree) :- X > 5, X =< 15, Degree is 0.

medium_price(X, Degree) :- X >= 0, X =< 3, Degree is 0.
medium_price(X, Degree) :- X > 3, X < 6, Degree is (X - 3) / 3.
medium_price(X, Degree) :- X >= 6, X =< 9, Degree is 1.
medium_price(X, Degree) :- X > 9, X < 12, Degree is 1 - (X - 9) / 3.
medium_price(X, Degree) :- X >= 12, X =< 15, Degree is 0.

high_price(X, Degree) :- X >= 0, X =< 8, Degree is 0.
high_price(X, Degree) :- X > 8, X < 13, Degree is (X - 8) / 5.
high_price(X, Degree) :- X >= 13, X =< 15, Degree is 1.
```

fuzzy-logic.pl

```
:- ["./degree-curves.pl", "./utils/read.pl", "./utils/utils.pl"].

apply_predicates([], _, []).
apply_predicates([Attribute/Predicate | Predicates], AttributeValues, [Degree | Degrees]) :-
    member(Attribute/Value, AttributeValues),
    call(Predicate, Value, Degree),
    apply_predicates(Predicates, AttributeValues, Degrees).

apply_conjuction([], 1).
apply_conjuction([Degree | Degrees], MinDegree) :-
    apply_conjuction(Degrees, MinDegreeRecursive),
    min(Degree, MinDegreeRecursive, MinDegree).

apply_disjunction([], 0).
apply_disjunction([Degree | Degrees], MaxDegree) :-
    apply_disjunction(Degrees, MaxDegreeRecursive),
    max(Degree, MaxDegreeRecursive, MaxDegree).

apply_premises(Operator, Premises, Inputs, Result) :-
    Operator = and,
    apply_predicates(Premises, Inputs, Degrees),
    apply_conjuction(Degrees, Result).
apply_premises(Operator, Premises, Inputs, Result) :-
    Operator = or,
    apply_predicates(Premises, Inputs, Degrees),
    apply_disjunction(Degrees, Result).
```

```
apply_min_premises_conseq([], _, []).
apply_min_premises_conseq([Premise|Premises], Consequence, [Min|Mins]) :-
    min(Premise, Consequence, Min),
    apply_min_premises_conseq(Premises, Consequence, Mins).

apply_rules([], _, []).
apply_rules([[Operator, Premises, [_/Consequence]]|KB], Inputs, Result) :-
    apply_premises(Operator, Premises, Inputs, ResultPremises),
    findall(
        Y,
        (between(0, 15, X), call(Consequence, X, Y)),
        ResultConsequence
    ),
    apply_min_premises_conseq(ResultConsequence, ResultPremises, ResultRule),
    apply_rules(KB, Inputs, ResultRuleRecursive),
 max_between_lists(ResultRule, ResultRuleRecursive, Result).

get_centroid_price(Ys, Price) :-
 interval(0, 15, Xs),
    sum(Ys, YSum),
    sum_between_lists_product(Xs, Ys, XYSum),
    Price is XYSum / YSum.

get_price(KB, Inputs, Ys, Price) :-
 apply_rules(KB, Inputs, Ys),
    get_centroid_price(Ys, Price).

solve :-
    read_file("./inputs/car-price.txt", KB),
    get_price(KB, [age/3, consumption/8, mileage/100000], _, Price),
    writeln(Price).
```

server-sld.pl

```
:- ["./utils/read.pl", "./utils/parse.pl", "./utils/utils.pl", "./sld-backward.pl", "./sld-forward.pl"].
:- use_module(library(socket)).

dogs([germanShepherd, rottweiler, beagle]).

start_server :-
    tcp_socket(Socket),
    tcp_bind(Socket, 5000),
    tcp_listen(Socket, 1),
    load_dogs_kb(KB),
    write("Server started on localhost:5000."), nl,
    write("Waiting for connections..."), nl,
    accept_connections(Socket, KB).

accept_connections(Socket, KB) :-
    tcp_accept(Socket, ClientSocket, _),
    write('Client connected!'), nl,
    handle_client(ClientSocket, KB),
    accept_connections(Socket, KB).

handle_client(Socket, KB) :-
    setup_call_cleanup(
        tcp_open_socket(Socket, InStream, OutStream),
```

```
        communicate_with_client(InStream, OutStream, KB),
        close_connection(InStream, OutStream)
    ).

    communicate_with_client(InStream, OutStream, KB) :-
        read_line_to_string(InStream, Answers),
        (   Answers == end_of_file
        ->  true
        ;   (   Answers == "stop"
            ->  write("Stop command received. Shutting down server..."), nl,
                format(OutStream, "Server shutting down.~n", []),
                flush_output(OutStream),
                halt
            ;   write("Processing the answers from the client..."), nl,
                process_answers(Answers, AnswersProcessed),
                merge(KB, AnswersProcessed, KBComplete),
                write("Started searching for the right dog..."), nl,
                dogs(Dogs),
                recommend_dogs(KBComplete, Dogs, DogsRecommended),
                format(OutStream, '~w~n', [DogsRecommended]),
                flush_output(OutStream)
            )
        ).

close_connection(InStream, OutStream) :-
    close(InStream),
    close(OutStream),
    write("Client disconnected."), nl.

process_answers(Answers, AnswersParsed) :-
    atom_string(Atom, Answers),
    atom_to_term(Atom, AnswersTerm, _),
    process_sentence(AnswersTerm, AnswersParsed).

load_dogs_kb(KB) :-
    read_file("./inputs/dogs.txt", KBUnpacked),
    process_sentences(KBUnpacked, KBParsed),
    unpack_kb(KBParsed, KB).

recommend_dogs(_, [], []).
recommend_dogs(KB, [Dog|Dogs], [Result|Results]) :-
    recommend_dogs(KB, Dogs, Results),
    resolution_forward(KB, [Dog], Result).

:- initialization(start_server).
```

server-fuzzy.pl

```
:- ["./utils/read.pl", "./utils/parse.pl", "./utils/utils.pl", "./fuzzy-logic.pl"].
:- use_module(library(socket)).

start_server :-
    tcp_socket(Socket),
    tcp_bind(Socket, 5000),
    tcp_listen(Socket, 1),
    read_file("./inputs/car-price.txt", KB),
    write("Server started on localhost:5000."), nl,
    write("Waiting for connections..."), nl,
```

```
    accept_connections(Socket, KB).

accept_connections(Socket, KB) :-
    tcp_accept(Socket, ClientSocket, _),
    write("Client connected!"), nl,
    handle_client(ClientSocket, KB),
    accept_connections(Socket, KB).

handle_client(Socket, KB) :-
    setup_call_cleanup(
        tcp_open_socket(Socket, InStream, OutStream),
        communicate_with_client(InStream, OutStream, KB),
        close_connection(InStream, OutStream)
    ).

communicate_with_client(InStream, OutStream, KB) :-
    read_line_to_string(InStream, Answers),
    (   Answers == end_of_file
    -> true
    ;   ( Answers == "stop"
        ->  write("Stop command received. Shutting down server..."), nl,
            format(OutStream, "Server shutting down.~n", []),
            flush_output(OutStream),
            halt
        ;   writeln("Processing the question from the client..."),
            process_answers(Answers, AnswersProcessed),
            writeln("Started computing the price..."),
            get_price(KB, AnswersProcessed, Degrees, Price),
            writeln("Price computed!"),
            format(OutStream, '{"degrees": ~w, "price": ~w}', [Degrees, Price]),
            flush_output(OutStream)
        )
    ).

close_connection(InStream, OutStream) :-
    close(InStream),
    close(OutStream),
    write("Client disconnected."), nl.

process_answers(Answers, AnswersProcessed) :-
    atom_string(Atom, Answers),
    atom_to_term(Atom, AnswersProcessed, _).

:- initialization(start_server).
```