KNOWLEDGE REPRESENTATION AND REASONING

PROJECT 2

February 5, 2023

Rares Patrascu

Contents

Chapter 1

Backward-Forward-Chaining

1.1 Knowledge base

Rules:

- If a car has a sunroof and also has a premium sound system is considered a luxury car.
- If a car cost more that 30000 the car is expensive.
- If a car is expensive it has an interface.
- If a luxury car has an interface is a high end car.

Questions:

1. What is the car's price?
2. Does the car have a sunroof?
3. Does the car have a premium sound system?
4. Does the car have an interface?

Goal: Determine whether a car is high-end or not.

Horn clauses in CNF:

1. [n(sunroof), n(premiumSoundSystem), luxuryCar]
2. [n(cost),expensive]
3. [n(expensive), interface]
4. [n(luxuryCar), n(interface),highEndCar]

Where x represents a car, Sunroof(x) is a predicate indicating whether the car has a sunroof, PremiumSoundSystem(x) is a predicate indicating whether the car has a premium sound system, Cost(x) represents the price of the car, and Interface(x) is a predicate indicating whether the car has an interface.

1.2 Implemented Predicates

The following Predicates we're implemented in order to apply forward chaining and backward chaining on a knowledge-based written as Horn clauses based on some rules and answers of some questions related to the kb.

- negate: returns the negated value of an element.
- read_knowledge _base: reads the kb from a file as a list.
- ask_q: asks the questions to the user.
- to_pred: converts the user answer to the necessary predicate
- append_kb: used to concatenate the kb based on the rules with the kb based on user answers.
- contained: verify if all elements of a list are contained in another list.

- get_negated _elements: return the non-negated values of all the negated predicates from a list.
- backward: algorithm for backward inference on a kb with a goal.
- forward: algorithm for forward inference on a kb with a goal.
- back_forw _chaining: calls the program.

1.3 Code

```
negate(n(A),A).
negate(A,n(A)).

read_knowledge_base(S,[]) :-
    at_end_of_stream(S).
read_knowledge_base(S,[L|R]) :-
    not(at_end_of_stream(S)),
    read(S,L),
    read_knowledge_base(S,R).

ask_q([[P1],[P2],[P3],[P4]]):-
    has_premium_sound_sistem(P3),
    car_price(P1),
    has_sunroof(P2),
    has_interface(P4).


to_pred(yes,SPredicate,SPredicate):-!.
```

```
to_pred(y,SPredicate,SPredicate):-!.
to_pred(no,SPredicate,n(SPredicate)):-!.
to_pred(n,SPredicate,n(SPredicate)):-!.
to_pred(Price,SPredicate,SPredicate):-
    Price >= 15000,
    !.
to_pred(_,SPredicate,n(SPredicate)):-!.

car_price(Predicate):-
        repeat,
        writeln('What is the car price?'),
        read(Price),
    nl,
        (
    number(Price),
    Price >= 1
    ->
        to_pred(Price, cost, Predicate),
    !;
        writeln('The input should be a non null number.'), fail
    ).

has_sunroof(Predicate):-
        repeat,
        writeln('Does the car have a sunroof?'),
        read(USER_ANS),
```

```
    nl,
        (
    member(USER_ANS, [yes, no, y, n])
    ->
        to_pred(USER_ANS, sunroof, Predicate),!
    ;
        writeln('The input should be yes,y,no or n.'), fail
    ).

has_premium_sound_sistem(Predicate):-
        repeat,
        writeln('Does the car have a premium sound system?'),
        read(USER_ANS),
    nl,
        (
    member(USER_ANS, [yes, no, y, n])
    ->
    to_pred(USER_ANS, premiumSoundSystem, Predicate),!
    ;
        writeln('The input should be yes,y,no or n.'), fail
    ).

has_interface(Predicate):-
        repeat,
        writeln('Does the car have an interface?'),
        read(USER_ANS),
```

```
    nl,
        (
    member(USER_ANS, [yes, no, y, n])
    ->
    to_pred(USER_ANS, interface, Predicate),!
    ;
        writeln('The input should be yes,y,no or n.'), fail
    ).

append_KB([HEAD|_], KB_ANS, R):-
    append(HEAD,KB_ANS,R).

continue_or_end :-
    writeln('Write end if you want to stop the program else continue.'),
    read(USER_ANS), nl,
    (
    USER_ANS = end
    ->
    writef('%w typed. Program stoped.', [USER_ANS]), !
    ;
    USER_ANS = continue
    ->
    writef('%w typed. Program will continue.', [USER_ANS]), nl, fail
    ;
    writef('%w typed. Something else than "end" or "continue" was typed. Anyway the p
    ).
```

```
back_forw_chaining:-
    open('kb.txt', read, S),
    read_knowledge_base(S, KB_RULES), close(S),
    writeln('Knowledge based on rules.'),
    writeln(KB_RULES),nl,
    repeat,
    ask_q(KB_ANS),
    writeln('Knowledge based on user answers.'),
    writeln(KB_ANS),nl,
    writeln('ALL knowledge.'),
    append_KB(KB_RULES,KB_ANS, KB),
    writeln(KB),
    backward([n(highEndCar)], KB, R_back),nl,
    writef('Backward Chaining --> Determine whether a car is high-end or not : %w ' ,
    forward([highEndCar], KB,[], R_forw),
    writef('Forward Chaining --> Determine whether a car is high-end or not : %w ', [
    continue_or_end.

backward([], _, 'yes'):-!.
backward([H|Goals], KB, R):-
    member(Clause, KB),
    negate(H,H_negated),
    member(H_negated, Clause),
        delete(Clause, H_negated, Clause_withouth_H_negated),
        append(Clause_withouth_H_negated, Goals, Concatenate),
```

```
        backward(Concatenate, KB, R), !.
backward(_, _, 'no'):- !.


contained(List1, List2):-
    forall(member(X, List1), member(X, List2)).

get_negated_elements([], []).
get_negated_elements([n(H)|T], [H|NegT]):-
    get_negated_elements(T, NegT),!.
get_negated_elements([H|T], NegT):-
    H \= n(_),
    get_negated_elements(T, NegT),!.


forward(Goals, _, Solved, 'yes'):-
    contained(Goals, Solved), !.
forward(Goals, KB, Solved, R):-
    member(Clause, KB),
        member(Positive_atom, Clause),
    not(Positive_atom=n(_)),
        get_negated_elements(Clause, Clause_negated_atoms),
    contained(Clause_negated_atoms,Solved),
    not(contained([Positive_atom],Solved)),
    forward(Goals,KB,[Positive_atom|Solved],R), !.
forward(_, _, _, 'no'):- !.
```

1.4 Output

```
2 ?- back_forw_chaining.
Knowledge based on rules.
[[[n(sunroof),n(premiumSoundSystem),luxuryCar],[n(cost),expensive],[
,interface],[n(luxuryCar),n(interface),highEndCar]]]

Does the car have a premium sound system?
|: y.

What is the car price?
|: 17000.

Does the car have a sunroof?
|: y.

Does the car have an interface?
|: y.

Knowledge based on user answers.
[[cost],[sunroof],[premiumSoundSystem],[interface]]


Backward Chaining --> Determine whether a car is high-end or not : y
es
Forward Chaining --> Determine whether a car is high-end or not : ye
s
Write end if you want to stop the program else continue.
|: continue.

continue typed. Program will continue.
Does the car have a premium sound system?
|: n.

What is the car price?
|: 8000.

Does the car have a sunroof?
|: y.

Does the car have an interface?
|: y.

Knowledge based on user answers.
[[n(cost)],[sunroof],[n(premiumSoundSystem)],[interface]]

ALL knowledge.
[[n(sunroof),n(premiumSoundSystem),luxuryCar],[n(cost),expensive],[n
(expensive),interface],[n(luxuryCar),n(interface),highEndCar],[n(cos
t)],[sunroof],[n(premiumSoundSystem)],[interface]]

Backward Chaining --> Determine whether a car is high-end or not : n
o
Forward Chaining --> Determine whether a car is high-end or not : no

Write end if you want to stop the program else continue.
|: end.
```

Figure 1.1: FINAL RESULTS

Chapter 2

Vaguness

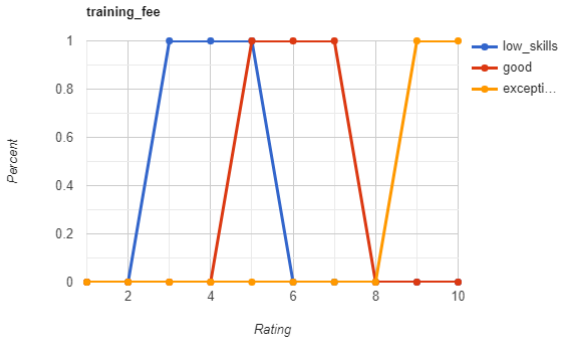
2.1 Rules

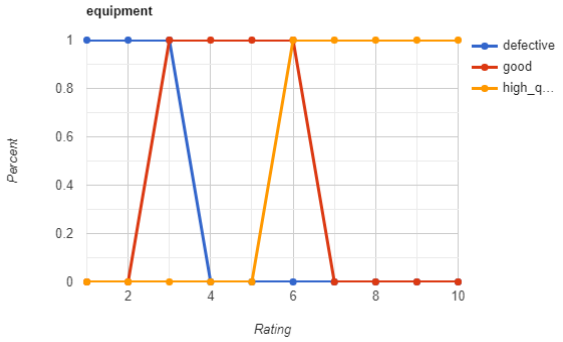
- If the tennis instructor has low skills or the tennis equipment is defective, then the training fee is minimal.
- If the tennis instructor is good or the tennis equipment is good, then the training fee is average.
- If the tennis instructor is exceptional or the tennis equipment is of high quality, then the training fee is generous.

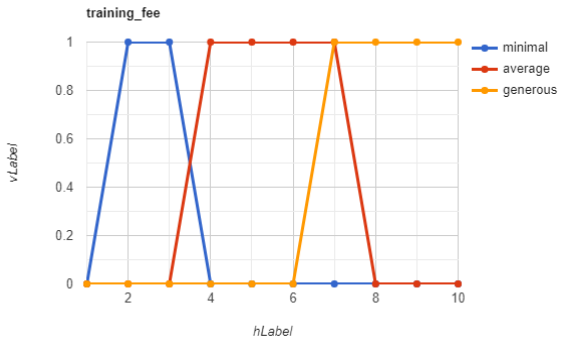
Rules represented in the proposed representation format from lab6:

- $[\text{or}, [\text{tennis\_instructor/low\_skills}, \text{equipment/defective}], [\text{training\_fee/minimal}]]$
- $[\text{or}, [\text{tennis\_instructor/good}, \text{equipment/good}], [\text{training\_fee/average}]]$
- $[\text{or}, [\text{tennis\_instructor/exceptional}, \text{equipment/high\_quality}], [\text{training\_fee/generous}]]$

2.2 Degree curves







2.3 Code

```
read_rules(S,[]) :-
    at_end_of_stream(S).
```

```
read_rules(S,[L|R]) :-
    not(at_end_of_stream(S)),
    read(S,L),
    read_rules(S,R).

map_rating(SERVICE,RATING,[SERVICE,RATING]).

rating_tennis_instructor(RATING_MAPPED):-
        repeat,
        writeln('What is the rating of the tennis instructor?'),
        read(RATING),
        (
    number(RATING),
    RATING >= 1,
    10 >= RATING
    ->
    map_rating(tennis_instructor,RATING,RATING_MAPPED),
    writef('Rating of the tenis instructor is %w', [RATING]),nl,nl, !
    ;
        writeln('The input should be a number between 1 and 10.'), fail
    ).

rating_equipment(RATING_MAPPED):-
        repeat,
        writeln('What is the rating of the equipment?'),
        read(RATING),
```

```
        (
    number(RATING),
    RATING >= 1,
    10 >= RATING
    ->
    map_rating(equipment,RATING,RATING_MAPPED),
    writef('Rating of the equipment is %w', [RATING]),nl,nl, !
    ;
        writeln('The input should be a number between 1 and 10.'), fail
    ).

ratings([R1,R2]):-
    rating_tennis_instructor(R1),
    rating_equipment(R2).

continue_or_end :-
    writeln('Write end if you want to stop the program else continue.'),
    read(USER_ANS), nl,
    (
    USER_ANS = end
    ->
    writef('%w typed. Program stoped.', [USER_ANS]), !
    ;
    USER_ANS = continue
    ->
    writef('%w typed. Program will continue.', [USER_ANS]), nl, fail
```

```
    ;
    writef('%w typed. Something else than "end" or "continue" was typed. Anyway the p
    ).



vaguness:-
    open('rules.txt', read, R),
    read_rules(R, RULES),
    close(R),
    writeln('The created rules are:'),
    write(RULES),nl,
    open('curves.txt', read, C),
    read_rules(C, CURVES),
    close(C),
    writeln('The degree curves are:'),
    writeln(CURVES),nl,
    repeat,
    ratings(RATINGS),
    writef('Mapped ratings --> %w.', [RATINGS]),nl,nl,
    continue_or_end.
```