# Probabilistic Logic Programming

## Lecture 1: Crash course in logic programming

Ronald de Haan
me@ronalddehaan.eu

University of Amsterdam

June 4, 2024

# Logic programming ←

# Main ingredients of logic programming

- In logic programming, you can express facts and if-then rules.
- For example, think of Prolog.
- The goal is to get a computationally useful language to represent knowledge and reason with it.

# Positive logic programs, without variables

- Positive logic programs consist of rules and facts.
  - Rules are of the form:

    ```
    a :- b, c, d, ..., z.
    ```

    which represents the implication $(b \wedge c \wedge d \wedge \cdots \wedge z) \rightarrow a$

    - `a` is called the head of the rule.
    - `b,c,d,...,z` is called the body of the rule.
  - Facts are of the form:

    ```
    a.
    ```

    which represents the positive literal $a$.

# Positive logic programs, without variables (ct'd)

- For example, this is a positive logic program.

```
wet :- rainy.
rainy.
cloudy :- rainy.
windy :- cloudy, rainy.
warm :- sunny.
```

# Positive logic programs, with variables

- We can also use variables (and function symbols).
  - The convention is to write variables starting with capital letters, and relation and function symbols starting with small letters.
  - Variables are always universally quantified.
- For example, in:

```
1 rainy(amsterdam).
2 rainy(vienna).
3 wet(X) :- rainy(X).
```

the rule on the third line is interpreted as:

$$\forall x.\ (\mathrm{Rainy}(x) \rightarrow \mathrm{Wet}(x))$$

(and `rainy` and `wet` are unary predicate symbols,
and `amsterdam` and `vienna` are constant symbols).

# Database semantics

- The notion of interpretations in first-order logic is very general, and complicated to work with.

  For example:
  - there can be any number of objects;
  - different constants can refer to the same objects;
  - statements that are not mentioned can (in general) be true or false.

- In logic programming, typically what is called database semantics is used:
  - "the objects mentioned are the only objects" (**domain closure**)
  - "objects with different names are different objects" (**unique-names assumption**)
  - "statements for which we have no reason to conclude that they are true, are false" (**closed-world assumption**)

# Database semantics (ct'd)

- These three assumptions mean that:
  - we do not have to (explicitly) say what function symbols mean;
  - we can represent objects simply by the terms that point to them;
  - we can represent the meaning of a relation $R$ by the set of atoms (over $R$) that are true—and then all atoms that are not in this set are false.
- In the context of logic programming, interpretations are sets of atoms:
  - all atoms in the set are true
  - all atoms not in the set are false

# Models

- Models are interpretations that make all rules of a program true.
- For example, this is a positive logic program.

```
wet :- rainy.
rainy.
cloudy :- rainy.
windy :- cloudy, rainy.
warm :- sunny.
```

- Then the following are (some of the) interpretations for this program:
  - $I_1 = \{\texttt{wet}, \texttt{rainy}, \texttt{cloudy}, \texttt{windy}\}$
  - $I_2 = \{\texttt{wet}, \texttt{rainy}, \texttt{cloudy}, \texttt{windy}, \texttt{sunny}\}$
  - $I_3 = \{\texttt{wet}, \texttt{rainy}, \texttt{cloudy}, \texttt{windy}, \texttt{sunny}, \texttt{warm}\}$
- Of these, $I_1$ and $I_3$ are models, and $I_2$ is not.

# Connection to FOL: Herbrand interpretations

- Let $\varphi$ be a first-order logic sentence.
- Then a Herbrand interpretation for $\varphi$ is an interpretation $(I, \cdot^I)$ such that:
  - $I$ is the set of all terms that can be built using constant and function symbols appearing in $\varphi$.
  - For each term $t$, $t^I = t$.

- *For example, take* $\varphi = \forall x.(R(x,x) \to R(mother(alex), x))$.
  - *Then for each Herbrand interpretation for* $\varphi$*:*

    $I = \{alex, mother(alex), mother(mother(alex)), mother(mother(mother(alex))), \ldots\}$,

    $alex^I = alex$,

    $mother(alex)^I = mother(alex)$,

    $mother(mother(alex))^I = mother(mother(alex))$,

    $\ldots$

# Implementing the closed-world assumption

- Remember (our version of) the closed-world assumption:

  > *"statements for which we have no reason to conclude that they are true, are false"*

- This is open to multiple interpretations
- Two ways to interpret this, that are commonly used in logic programming, are:
  - consider only minimal models of a logic program
  - consider only supported models of a logic program

# Minimal models

- For positive logic programs, there is a unique minimal model.
    - Minimal in terms of subset-inclusion.
- For example, for:

```
wet :- rainy.
rainy.
cloudy :- rainy.
windy :- cloudy, rainy.
warm :- sunny.
```

the minimal model is $\{\texttt{wet}, \texttt{rainy}, \texttt{cloudy}, \texttt{windy}\}$.

- We can find this minimal model $M$ with the following procedure:
    1. Start by putting all facts of the program into $M$.
    2. Repeat until $M$ does not change anymore:
        - If there is a rule $b \leftarrow c_1, \ldots, c_n$ where $c_1, \ldots, c_n \in M$, put $b$ in $M$ too.

# Supported models

- A model $M$ is a supported model of a logic program $P$ if:
    - for each atom a $\in M$, there is some rule $\mathsf{a} \leftarrow \beta.$ in $P$ such that $M$ makes $\beta$ true.
- In other words, $M$ is a supported model of $P$ if each atom a is “forced” to be in $M$ by some rule $\mathsf{a} \leftarrow \beta$.
- For positive logic programs, the unique minimal model is also a supported model.

# Negation

- It is often useful to use negation in the body of rules: `not` (meaning ¬)
- For example:

```
wet(X) :- rainy(X), not sunny(X).
rainy(amsterdam).
rainbow(X) :- rainy(X), sunny(X).
warm(X) :- sunny(X).
```

# Normal logic programs

- In normal logic programs, rules are of the following form:

```
a :- b1, ..., bn, not c1, ..., not cm.
```

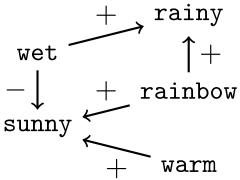
- When allowing negation, we lose the property that there is a unique minimal (and supported) model

# Stratified negation

- Negation is stratified in a program $P$ if the following holds.
  - Draw a directed graph, where the nodes are the relation symbols appearing in $P$.
  - For every rule `a :- b`$_1$`, ..., b`$_n$`, not c`$_1$`, ..., not c`$_m$`.` in $P$:
    - Draw an edge labelled with $-$ (a negative edge) from the relation symbol in `a` to the relation symbol in `c`$_i$, for each $1 \leq i \leq m$.
    - Draw an edge labelled with $+$ (a positive edge) from the relation symbol in `a` to the relation symbol in `b`$_i$, for each $1 \leq i \leq n$.
  - If this graph has no cycles involving negative edges, then negation is stratified.

*For example:*

```
wet(X) :- rainy(X), not sunny(X).
rainy(amsterdam).
rainbow(X) :- rainy(X), sunny(X).
warm(X) :- sunny(X).
```



# Iterated fixpoint model

- For programs with stratified negation, we define iterated fixpoint models (IFMs)
  - *(Details of the definition on the next slide..)*
  - Every program with stratified negation has an IFM, and only one
  - The IFM is a minimal model, and it is a supported model
  - (But there might be models that are minimal and/or supported that are not the IFM)

- This model serves as "intended model"
  - We consider the unique IFM as the (only) meaning of the program
  - (Just like the unique minimal model for positive programs..)

# Iterated fixpoint model (ct'd)

- We can compute the IFM of a program with stratified negation as follows:
  *(At the same time, this procedure defines the IFM..)*

1. Assign a positive integer to each relation symbol such that:
    - When there is a negative edge from a to b, then the number assigned to a is strictly larger than that assigned to b.
    - When there is a positive edge from a to b, then the number assigned to a is at least as large as that assigned to b.

    *(We can do this because there are no cycles with negated edges.)*
2. Start by putting all facts of the program into $M$.
3. Proceed in stages, one stage for each assigned integer $i$, going from low to high.

    In each stage $i$:

    - Repeat until $M$ does not change anymore:

      If there is a rule $b \leftarrow c_1, \ldots, c_n, \text{not } d_1, \ldots, \text{not } d_m.$ in $P$ where $c_1, \ldots, c_n \in M$ and $d_1, \ldots, d_m \notin M$, and $b$ is assigned the number $i$, then put $b$ in $M$ too.

# Iterated fixpoint model (ct'd)

- Take this example again:

```
wet(X) :- rainy(X), not sunny(X).
rainy(amsterdam).
rainbow(X) :- rainy(X), sunny(X).
warm(X) :- sunny(X).
```

- Assign 1 to `rainy`, `rainbow`, `sunny`, `warm`, and 2 to `wet`.
- After stage 1, we end up with the interpretation $M_1 = \{\texttt{rainy(amsterdam)}\}$
- After stage 2, we end up with the interpretation $M_2 = \{\texttt{rainy(amsterdam)}, \texttt{wet(amsterdam)}\}$
- Then $M_2$ is the iterated fixpoint model

# Iterated fixpoint model (ct'd)

- On the previous slides, we assigned numbers to predicate symbols
  (for programs with variables)
- Instead, we could assign numbers to atoms
  (for ground programs, i.e., programs without variables)
- The procedure for computing the IFM works entirely similarly in that case

# What if we want unstratified negation?

- What if we want to use negation, but it is not stratified in our knowledge base?
- For example:

```
rainy :- not sunny.
sunny :- not rainy.
```

- As mentioned, then there is not always a unique minimal model.
  - In this example, the following are minimal models: $\{$`rainy`$\}$ and $\{$`sunny`$\}$.

? *What to do?*

# Minimality and supportedness does not capture what we want

- Not all minimal and supported models match our intuitions for what desirable models are (under the closed-world assumption)

- Consider for example the following normal logic program with unstratified negation:

```
rainy :- windy, not sunny.
windy :- rainy, not sunny.
sunny :- not rainy.
```

- Then the model $M = \{\texttt{rainy}, \texttt{windy}\}$ is both minimal and supported

- But this model can be argued not to be in line with the closed-world assumption:
    - `rainy` and `windy` provide each other a reason for being true, but there is no “external” reason why they should be true

# Answer Set Programming </>

# Syntax of ASP

- Answer Set Programming (ASP) assigns the so-called answer set semantics to logic programs with negations.

- In the basic language of ASP (normal logic programs), programs may contain facts and rules of the form:

```
a :- b_1, ..., b_n, not c_1, ..., not c_m.
```

  - Negation does not have to be stratified.
  - It may contain variables, that must appear safely:
    - Every variable that appears in the head of a rule, must also appear in a non-negated atom in the body.
    - Every variable that appears in a negated atom in the body of a rule, must also appear in a non-negated atom in the body.

# Semantics of ASP (intuitions)

1. Consider an interpretation $M$ as an assumption for which atoms are true and which are false.
2. Construct a (positive) variant $P^M$ of the program $P$ that takes into account this assumption.
3. Check whether $M$ is the (unique) minimal model of $P^M$.

# Semantics of ASP

- Take a normal logic program $P$ and an interpretation $M$.
- The reduct $P^M$ of $P$ w.r.t. $M$ is obtained from $P$ by:
  1. removing rules with `not a` in the body, for a $\in M$
  2. removing literals `not b` from all rules, for b $\notin M$
- An answer set of $P$ is a set $M$ that is the minimal model of $P^M$
- For example, take $P$ to be:

```
rainy :- not sunny.
sunny :- not rainy.
```

and $M = \{\ \texttt{sunny}\ \}$. Then $P^M$ is:

```
sunny.
```

and $M$ is the minimal model of $P^M$.

# Another example

**Logic program** *P*:

```
a :- not b.
b :- not a.
c :- b.
b :- a.
```

What are the answer sets of this program?

**Answer sets:**

```
b c
```

# Another example (ct'd)

**Logic program** $P^{\{\texttt{b},\texttt{c}\}}$**:**

```
~~a :- not b.~~
b :- ~~not a~~.
c :- b.
b :- a.
```

$\{\texttt{b},\texttt{c}\}$ is the minimal model of $P^{\{\texttt{b},\texttt{c}\}}$,
and thus $\{\texttt{b},\texttt{c}\}$ is an answer set of $P$.

# Another example (ct'd)

**Logic program** $P^{\{\texttt{a},\texttt{b},\texttt{c}\}}$:

```
~~a :- not b.~~
~~b :- not a.~~
c :- b.
b :- a.
```

$\{\texttt{a},\texttt{b},\texttt{c}\}$ is not the minimal model of $P^{\{\texttt{a},\texttt{b},\texttt{c}\}}$,
and thus $\{\texttt{a},\texttt{b},\texttt{c}\}$ is not an answer set of $P$.

# Yet another example

**Logic program:**

```
low :- not high.
high :- not low.
```

```
left :- not right.
right :- not left.
```

**Answer sets:**

```
low left
```

```
low right
```

```
high left
```

```
high right
```

# Using variables

**Logic program:**

```
num(1).
num(2).

left(X) :- not right(X), num(X).
right(X) :- not left(X), num(X).
```

**Answer sets:**

```
num(1) num(2) right(1) left(2)
num(1) num(2) right(1) right(2)
num(1) num(2) left(1) left(2)
num(1) num(2) left(1) right(2)
```

# Constraints

- Constraints are rules without head, meaning that the body must be false.

**Logic program:**

```
num(1).
num(2).
```

```
left(X) :- not right(X), num(X).
right(X) :- not left(X), num(X).
```

```
:- left(1). % it cannot be the case that left(1) is true
```

**Answer sets:**

```
num(1) num(2) right(1) left(2)
```

```
num(1) num(2) right(1) right(2)
```

# Answer sets vs. models

```
a :- not b.
b :- not a.
```