PSL supports two primary types of rules: Logical and Arithmetic. Each of these types of rules support weights and squaring.

Logical Rules

Logical rules in PSL are implications joined with logical operators (with the exception of negative priors). Since PSL uses soft logic, hard logic operators are replaced with Lukasiewicz operators.

Operators

`& (&&)` - Logical And
The `and` operator is binary and functions as a `Lukasiewicz t-norm` operator: `A & B = MAX(0, A + B - 1)`

`| (||)` - Logical Or
The `or` operator is binary and functions as a `Lukasiewicz t-conorm` operator: `A | B = MIN(1, A + B)`

`>> (->) / << (<-)` - Implication
The `implication` operator acts similar to the standard logical implication where the truth of the body implies the truth of the head. Note that the head is always the side the that arrow is pointing at and both directions are supported. It is most common to see rules where the body is on the left and the head is on the right. The body of an implication must be a conjunctive clause (contain only `and` operators) while the head must be a disjunctive clause (contain only `or` operators).

`~ (!)` - Negation
The `negation` operator is unary and functions as a `Lukasiewicz negation` operator: `~A = 1 - A`

Examples

```
// The same rule written in two different ways.
Nice(A) & Nice(B) -> Friends(A, B)
Friends(A, B) << Nice(A) && Nice(B)

// Using a disjunction in the head instead of a conjunction in the body.
// Also written two different ways.
Friends(A, B) >> Nice(A) || Nice(B)
Nice(A) | Nice(B) <- Friends(A, B)
```

Arithmetic Rules

Arithmetic rules are relations of two linear combinations.

Operators

The following operators are used in arithmetic rules:

- `+`
- `-`
- `*`
- `/`

Note that each side of an arithmetic rule must be a linear combination, so `+/-` is only allowed between terms and `*//` is only allowed for coefficients.

Relational Operator

The following relational operators are allowed between the two linear combinations:

- `=`
- `<=`
- `>=`

Summation

A summation can be used when you want to aggregate over a variable. You turn a variable into a summation variable by prefixing it with a `+`. Each sum variable can only be used once per expression, but you may have multiple different summation variables.

Filter Clauses

A filter clause appears at the end of a rule and decides what values the summation variable can take. There can be multiple filter clauses for each rule, but each summation variable can have at most one filter clause. The filter clause is a logical expression, but uses hard logic rather than Lukasiewicz. All non-zero truth values are considered true in a filter expression. If this expression evaluates to zero for a value, then that value is not used in the summation. Valid things that may appear in the filter clause are:

- Constants
- Variables appearing in the associated arithmetic expression
- Closed predicates
- The single summation variable that was defined as the argument

```
// Only sum up friends of A that are nice.
Friends(A, +B) <= 1 {B: Nice(B)}

// Only sum up friends of A that are similar to A.
// Note that Similar must be a closed predicate.
Friends(A, +B) <= 1 {B: Similar(A, B)}

// Only sum up friends of A that are not similar to A.
// Note that Similar must be a closed predicate.
Friends(A, +B) <= 1 {B: !Similar(A, B)}

// Only sum up friends of A where both A and B are nice and similar.
// Note that Similar must be a closed predicate.
```

```
Friends(A, +B) <= 1 {B: Nice(A) & Nice(B) & Similar(A, B)}

// Only sum up combinations for friends where A is nice and B is not nice.
Friends(+A, +B) <= 1 {A: Nice(A)} {B: !Nice(B)}
```

Coefficients

Each term in an arithmetic rule can take an optional coefficient. The coefficient may be any real number and can either appear before the term and act as a multiplier:

```
2.5 * Similar(A, B) >= 1
```

or, appear after the term and act as a divisor:

```
Similar(A, B) / 2.5 <= 1
```

Coefficient Operators

Special coefficients (called **Coefficient Operators**) may be used:

`|A|` - Cardinality
A cardinality coefficient may only be used on a summation variable. It becomes the count of the number of terms substituted for a summation variable.

`@Min[A, B]` - Max
Returns the maximum of `A` and `B`. May be used with summation variables.

`@Max[A, B]` - Min
Returns the minimum of `A` and `B`. May be used with summation variables.

Examples

(Note that these rules are meant to show the semantics of arithmetic rules and may not make logical sense.)

```
Friends(A, B) = 0.5
Friends(A, +B) <= 1
Friends(A, +B) / |B| <= 1
@Min[2, |B|] * Friends(A, +B) <= 1
Friends(A, +B) <= 1 {B: Nice(B)}
Friends(A, B) <= Nice(A) + Nice(B)
Friends(A, B) >= 3.0 * Nice(A) - 2.0 * Nice(B)
```

Weights

Every rule must be either weighted or unweighted. Unweighted rules are also called constraints since they are strictly enforced.

Weighted

Weighted rules are prefixed with the weight and a colon:

```
<weight>: <rule>
```

For example:

```
2.5: Nice(A) & Nice(B) & (A != B) -> Friends(A, B)
5.0: Friends(A, +B) <= 1
10.0: Friends(A, +B) <= 1 {B: Nice(B)}
```

Unweighted

Unweighted rules (constraints) are suffixed with a period:

```
<rule> .
```

For example:

```
Nice(A) & Nice(B) & (A != B) -> Friends(A, B) .
Friends(A, +B) <= 1 .
Friends(A, +B) <= 1 . {B: Nice(B)}
```

Squaring

Any weighted rule can choose to square their hinge-loss functions. Squaring the hinge-loss (or "squared potentials") may result in better performance. Non-squared potentials tend to encourage a "winner take all" optimization, while squared potentials encourage more trading off. To square a rule, just suffix a `^2` to it:

```
2.5: Nice(A) & Nice(B) (A != B) -> Friends(A, B) ^2
5.0: Friends(A, +B) <= 1 ^2
10.0: Friends(A, +B) <= 1 ^2 {B: Nice(B)}
```

Priors

You may specify priors for a predicate in PSL. A prior is specified on a specific predicate and affects all open ground atoms of that predicate. Priors in PSL must be weighted and may be squared. Priors tend to have low weights, since they are supposed to get overpowered by evidence.

Note that priors are not the same as specifying an initial value for your open predicate in a data file. Once optimization starts, the initial value specified in the data file will quickly get changed and have little/no impact on the final optimization. A prior however, is a ground rule that becomes a full fledged potential function that actively participates in optimization.

Negative Priors

Negative priors are the most common type of prior in PSL. It assumes that all ground atoms for the predicate are zero.

Negative priors may be specified using logical rules:

```
1.0: ~Friends(A, B) ^2
```

This prior can be interpreted as “By default, people are not friends”.

Arithmetic rules may also be used to specify a negative prior:

```
1.0: Friends(A, B) = 0 ^2
```

Positive Priors

Positive priors can be a little more tricky than negative priors.

If you want all the ground atoms to take the same positive prior, then you can just use an arithmetic rule:

```
1.0: Friends(A, B) = 0.75 ^2
```

If you want different ground atoms to have different positive priors, then you will need to use a surrogate predicate. First, create a new closed predicate that corresponds 1-to-1 with your open predicate you wish to put the prior on. Then add observations for the surrogate predicate with the truth value being the prior you wish to put on that ground atom. Now just create a rule that directly ties together the surrogate predicate to the open predicate. See the example below.

Non-Uniform Prior Example

Consider a PSL program where we are trying to infer friendship (the `Friends` predicate). We may have a prior belief on the friendship quality between all people in our data. To encode this prior belief, we will first construct a surrogate predicate called `FriendsPrior` (the name does not matter). Now we will load `FriendsPrior` with observations where the truth value of the observation is our prior. Our data file for `FriendsPrior` may look something like:

```
Alice   Bob     1.0
Alice   Charlie 0.75
Bob     Charlie 0.33
```

Now we will add this rule that acts as our prior:

```
1.0: FriendsPrior(A, b) -> Friends(A, B) ^2
```

Special Operators

Both logical and arithmetic rules support some special operators.

`== (=)` - Equals
Ensure that that the left and right side are equal. Note that this is not the same as a similarity function evaluating to 1. Two variables may be 100% similar, but equals will only evaluate to 1 unless they refer to the same value.

`!= (~=)` - Not Equals
Evaluates to 1 when both side are not the same. This is a very common operator to use in most rules.

For example, consider the following two rules:

```
Nice(A) && Nice(B) -> Friends(A, B)
Nice(A) && Nice(B) && (A != B) -> Friends(A, B)
```

If A and B can both take the values “Alice” and “Bob”, then the first rule would generate the following groundings:

```
Nice("Alice") && Nice("Alice") -> Friends("Alice", "Alice")
Nice("Alice") && Nice("Bob") -> Friends("Alice", "Bob")
Nice("Bob") && Nice("Bob") -> Friends("Bob", "Bob")
Nice("Bob") && Nice("Alice") -> Friends("Bob", "Alice")
```

While the second rule would only generate two groundings:

```
Nice("Alice") && Nice("Bob") -> Friends("Alice", "Bob")
Nice("Bob") && Nice("Alice") -> Friends("Bob", "Alice")
```

`% (^)` - Non-Symmetric
Ensure that the reverse (or equal) paring of the two operands is not grounded. Essentially, this enforces a less than relationship between the two operands. For example, consider the following two rules:

```
SimilarNames(A, B) -> SamePerson(A, B)
SimilarNames(A, B) && (A % B) -> SamePerson(A, B)
```

If A and B can both take the values “Alice” and “Bob”, then the first rule would generate the following groundings:

```
SimilarNames("Alice", "Alice") && ("Alice" % "Alice") -> SamePerson("Alice", "Ali
SimilarNames("Alice", "Bob") && ("Alice" % "Bob") -> SamePerson("Alice", "Bob")
SimilarNames("Bob", "Alice") && ("Bob" % "Alice") -> SamePerson("Bob", "Alice")
SimilarNames("Bob", "Bob") && ("Bob" % "Bob") -> SamePerson("Bob", "Bob")
```

While the second rule would only generate one grounding:

```
SimilarNames("Alice", "Bob") && ("Alice" % "Bob") -> SamePerson("Alice", "Bob")
```

Edit This Page