Assigning Probabilities to Boolean Functions

by Sven Nilsen, 2017

Here I represent a method to assign a probability to all boolean functions. The probability of a function returning "true" is calculated from independently randomized inputs.

All boolean functions can be constructed using a higher order function `if`:

```
if := \(a, b) = \(c) = if c { a } else { b }
```

For example:

if(false, false) <=> false_1
if(false, true) <=> not
if(true, false) <=> id
if(true, true) <=> true_1
if(id, false_1) <=> and

This construction can be used to define a unique probability that a boolean function returns "true" from input bias. An input bias is a probability of the value being `true`.

First we define the probability for "false" and "true":

$$P(\text{false}) = 0$$
$$P(\text{true}) = 1$$

To define a probability for all boolean functions, it is sufficient to define it for `if`:

$$P(\text{if}(A, B)(x)) = P(x)\cdot P(A) + (1-P(x))\cdot P(B)$$

The formula above assumes probabilities are independent `P(A ∩ B) = P(A) · P(B)`.

Talking about `if` in path semantics makes only sense when the function identity is unique. It turns out that when probabilities are dependent, this automatically adds domain constraints to the input and changes the path set of the function. The path set is changed when going from a total function to a partial function. This means that if probabilities are dependent, one is actually talking about a partial function of `if` and not the total function `if`. Therefore, in order to make sense the probabilities must be independent. One can talk about probabilities of partial functions in general if the domain constraints are defined, but not in this specific case because domain constraints are not allowed. Probability of Boolean functions with domain constraints is another topic.

Proofs of all functions of type `bool → bool`:

$\underline{P(false_1(x)) = 0:}$

$P(false_1(x))$
$P(if(false, false)(x))$
$P(x)\cdot P(false) + (1-P(x))\cdot P(false)$
$P(x)\cdot 0 + (1-P(x))\cdot 0$
0

$\underline{P(not(x)) = 1-P(x):}$

$P(not(x))$
$P(if(false, true)(x))$
$P(x)\cdot P(false) + (1-P(x))\cdot P(true)$
$P(x)\cdot 0 + (1-P(x))\cdot 1$
$1-P(x)$

$\underline{P(id(x)) = P(x):}$

$P(id(x))$
$P(if(true, false)(x))$
$P(x)\cdot P(true) + (1-P(x))\cdot P(false)$
$P(x)\cdot 1 + (1-P(x))\cdot 0$
$P(x)$

$\underline{P(true_1(x)) = 1:}$

$P(true_1(x))$
$P(if(true, true)(x))$
$P(x)\cdot P(true) + (1-P(x))\cdot P(true)$
$P(x)\cdot 1 + (1-P(x))\cdot 1$
$P(x) + 1 – P(x)$
1

The function `and` is important in probability theory because it takes the product of input biases:

$\underline{P(and(x_0, x_1)) = P(x_0)\cdot P(x_1):}$

$P(and(x_0, x_1))$
$P(if(id, false_1)(x_0, x_1))$
$P(x_0)\cdot P(id(x_1)) + (1-P(x_0))\cdot P(false_1(x_1))$
$P(x_0)\cdot P(x_1) + (1-P(x_0))\cdot 0$
$P(x_0)\cdot P(x_1)$

The function `or` has two commonly used forms, one that takes the sum:

$\underline{P(or(x_0, x1)) = P(x_0) + P(x_1) – P(x_0)\cdot P(x_1):}$

$P(or(x_0, x1))$
$P(if(true_1, id)(x_0, x_1))$
$P(x_0)\cdot P(true_1(x_1)) + (1-P(x_0))\cdot P(id(x_1))$
$P(x_0)\cdot 1 + (1-P(x_0))\cdot P(x_1)$
$P(x_0) + P(x_1) – P(x_0)\cdot P(x_1)$

One that uses the symmetric path `and[not] <=> or`:

$\underline{P(or(x_0, x1)) = 1 – (1-P(x_0))·(1-P(x_1)):}$

$P(or(x_0, x_1))$
$P(not(and(not(x_0), not(x_1))))$
$1 – P(and(not(x_0), not(x_1))$
$1 – P(not(x_0))·P(not(x_1))$
$1 – (1-P(x_0))·(1-P(x_1))$

The second form for `or`, the one that uses the symmetric path, has the advantage that it is easier to generalize for N arguments, such as a list. This is because `and` takes the product of the input biases to calculate the probability of returning `true` and this property carries over to `or` through the symmetric path.