Interactive Bayesian Probabilistic Programming and Debugging

IBM Programming Languages Day
December 4, 2017

Javier Burroni, Arjun Guha, David Jensen
Definition of (Bayesian) Probabilistic Programming Languages

Regular PL with two special constructs:

• sample
• observe

Definition of (Bayesian) Probabilistic Programming Languages

Regular PL with two special constructs:

• sample
• observe

And a way to access results:

• query

Definition of (Bayesian) Probabilistic Programming Languages

Regular PL with two special constructs:

- sample
- observe

And a way to access results:

- query

Definition of (Bayesian) Probabilistic Programming Languages

Regular PL with two special constructs:

- sample
- observe

And a way to access results:

- query

\[
\text{random Real } \mu() \sim \text{Gaussian}(100.0, 10.0);
\text{random Real } x() \sim \text{Gaussian}(\mu(), 15.0);
\]

\[
\text{obs } x() > 120;
\text{query } \mu() > 100;
\]

Definition of (Bayesian) Probabilistic Programming Languages

Regular PL with two special constructs:

• sample
• observe

And a way to access results:

• query

Definition of (Bayesian) Probabilistic Programming Languages

Regular PL with two special constructs:

• sample
• observe

And a way to access results:

• query

Example of PPL

0: type City;
1: type PrepLevel;
2: type DamageLevel;

3: random City First ~ UniformChoice({c for City c});
4: random City NotFirst ~ UniformChoice({c for City c: c != First});
5: random PrepLevel Prep(City c) ~
6: if (First == c) then Categorical({High -> 0.5, Low -> 0.5})
7: else case Damage(First) in
8:   {Severe -> Categorical({High -> 0.9, Low -> 0.1}),
9:    Mild -> Categorical({High -> 0.1, Low -> 0.9})};
10: random DamageLevel Damage(City c) ~
11: case Prep(c) in {High -> Categorical({Severe -> 0.2, Mild -> 0.8}),
12:    Low -> Categorical({Severe -> 0.8, Mild -> 0.2})};

13: distinct City A, B;
14: distinct PrepLevel Low, High;
15: distinct DamageLevel Severe, Mild;

16: obs Damage(First) = Severe;
17: query Damage(NotFirst);

Milch, B. et al. BLOG: Probabilistic models with unknown objects. in (2005).
Example of PPL

0: type City;
1: type PrepLevel;
2: type DamageLevel;

3: random City First ~ UniformChoice({c for City c});
4: random City NotFirst ~ UniformChoice({c for City c: c != First});
5: random PrepLevel Prep(City c) ~
6:   if (First == c) then Categorical({High -> 0.5, Low -> 0.5})
7:     else case Damage(First) in
8:       {Severe -> Categorical({High -> 0.9, Low -> 0.1}),
9:        Mild -> Categorical({High -> 0.1, Low -> 0.9})};
10: random DamageLevel Damage(City c) ~
11:   case Prep(c) in {High -> Categorical({Severe -> 0.2, Mild -> 0.8}),
12:       Low -> Categorical({Severe -> 0.8, Mild -> 0.2})};

13: distinct City A, B;
14: distinct PrepLevel Low, High;
15: distinct DamageLevel Severe, Mild;

16: obs Damage(First) = Severe;

17: query Damage(NotFirst);

<table>
<thead>
<tr>
<th>query Damage(NotFirst)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mild</td>
</tr>
<tr>
<td>Severe</td>
</tr>
<tr>
<td>dtype: float64</td>
</tr>
</tbody>
</table>

Milch, B. et al. BLOG: Probabilistic models with unknown objects. in (2005).
Interactive limitations

• New queries require re-execution of the entire program
  ✦ Not efficient as neither data nor generative model were changed
Our approach

• Perform backward inference only once
• The result of inference is a posterior distribution over traces — FOL structures.
• New keyword: inspect(expr) \rightarrow value.
  Evaluate the expression in a trace
• query(expr) implemented as application of inspect over a sample of the posterior distribution of traces.
Advantages

• No need to know the queries before running inference
  ✦ Allows interactively querying of the posterior distribution
Dynamically querying

\begin{verbatim}
0: type City;
1: type PrepLevel;
2: type DamageLevel;
3: random City First ~ UniformChoice({c for City c});
4: random City NotFirst ~ UniformChoice({c for City c: c != First});
5: random PrepLevel Prep(City c) -
6: if (First == c) then Categorical({High -> 0.5, Low -> 0.5})
7: else case Damage(First) in
8: (Severe -> Categorical({High -> 0.9, Low -> 0.1}),
9: Mild -> Categorical({High -> 0.1, Low -> 0.9}));
10: random DamageLevel Damage(City c) -
11: case Prep(c) in (High -> Categorical({Severe -> 0.2, Mild -> 0.8}),
12: Low -> Categorical({Severe -> 0.8, Mild -> 0.2}));
13: distinct City A, B;
14: distinct PrepLevel Low, High;
15: distinct DamageLevel Severe, Mild;
16: obs Damage(First) = Severe;
17: query Damage(NotFirst);
\end{verbatim}

\textbf{query} Damage(A)

<table>
<thead>
<tr>
<th>Severity</th>
<th>Probability</th>
</tr>
</thead>
<tbody>
<tr>
<td>Severe</td>
<td>0.641092</td>
</tr>
<tr>
<td>Mild</td>
<td>0.358908</td>
</tr>
</tbody>
</table>

\textbf{query if Damage(A) == Severe then Prep(A) else Prep(B)};

<table>
<thead>
<tr>
<th>Level</th>
<th>Probability</th>
</tr>
</thead>
<tbody>
<tr>
<td>Low</td>
<td>0.741546</td>
</tr>
<tr>
<td>High</td>
<td>0.258454</td>
</tr>
</tbody>
</table>
Advantages

• No need to know the queries before running inference
  ✦ Allows interactively querying of the posterior distribution
• `inspect(expr)` accepts any valid BLOG expression
Inspect one world

0: type City;
1: type PrepLevel;
2: type DamageLevel;

3: random City First ~ UniformChoice({c for City c});
4: random City NotFirst ~ UniformChoice({c for City c: c != First});
5: random PrepLevel Prep(City c) ~
6: if (First == c) then Categorical({High -> 0.5, Low -> 0.5})
7: else case Damage(First) in
8: (Severe -> Categorical({High -> 0.9, Low -> 0.1}),
9: Mild -> Categorical({High -> 0.1, Low -> 0.9}));
10: random DamageLevel Damage(City c) ~
11: case Prep(c) in (High -> Categorical({Severe -> 0.2, Mild -> 0.8}),
12: Low -> Categorical({Severe -> 0.8, Mild -> 0.2}));

13: distinct City A, B;
14: distinct PrepLevel Low, High;
15: distinct DamageLevel Severe, Mild;

16: obs Damage(First) = Severe;
17: query Damage(NotFirst);

inspect if Damage(A) == Severe then Prep(A) else Prep(B)

value: Low
Advantages

• No need to know the queries before running inference
  ✦ Allows interactively querying of the posterior distribution

• `inspect(expr)` accepts any valid BLOG expression
  ✦ The generative model is made of BLOG expressions
  ✦ step-by-step debugging can be implemented by recursively inspecting the generative process
Step debugging

0: type City;
1: type PrepLevel;
2: type DamageLevel;

3: random City First ~ UniformChoice({c for City c});
4: random City NotFirst ~ UniformChoice({c for City c: c != First});
5: random PrepLevel Prep(City c) ~
6:   if (First == c) then Categorical({High -> 0.5, Low -> 0.5})
7:   else case Damage(First) in
8:     (Severe -> Categorical({High -> 0.9, Low -> 0.1}),
9:      Mild -> Categorical({High -> 0.1, Low -> 0.9}));
10: random DamageLevel Damage(City c) ~
11:   case Prep(c) in (High -> Categorical({Severe -> 0.2, Mild -> 0.8}),
12:                    Low -> Categorical({Severe -> 0.8, Mild -> 0.2}));

13: distinct City A, B;
14: distinct PrepLevel Low, High;
15: distinct DamageLevel Severe, Mild;

16: obs Damage(First) = Severe;
17: query Damage(NotFirst);
Step debugging

debugger.step()
Entering: obs Damage(First) = Severe

debugger.step()
Entering: Damage(First)

debugger.step()
Entering: Damage(First)

debugger.step()
Entering: UniformChoice({c for City c})

debugger.runToLine(18)
Entering: case Prep(c) in

0: type City;
1: type PrepLevel;
2: type DamageLevel;
3: random City First ~ UniformChoice({c for City c});
4: random City NotFirst ~ UniformChoice({c for City c: c != First});
5: random PrepLevel Prep(City c) ~
6: if (First == c) then Categorical({High -> 0.5, Low -> 0.5})
7: else case Damage(First) in
8: (Severe -> Categorical({High -> 0.9, Low -> 0.1}),
9: Mild -> Categorical({High -> 0.1, Low -> 0.9}))
10: random DamageLevel Damage(City c) ~
11: case Prep(c) in (High -> Categorical({Severe -> 0.2, Mild -> 0.8}),
12: Low -> Categorical({Severe -> 0.8, Mild -> 0.2}));
13: distinct City A, B;
14: distinct PrepLevel Low, High;
15: distinct DamageLevel Severe, Mild;
16: obs Damage(First) = Severe;
17: query Damage(NotFirst);
debugger.step()
Entering: **obs Damage**(First) = Severe

designer.step()
Entering: **Damage**(First)

```
0: type City;
1: type PrepLevel;
2: type DamageLevel;

3: random City First ~ UniformChoice(\{c for City c\});
4: random City NotFirst ~ UniformChoice(\{c for City c: c != First\});
5: random PrepLevel Prep(City c) ~
6: if (First == c) then Categorical(\{High -> 0.5, Low -> 0.5\})
7: else case Damage(First) in
8:   (Severe -> Categorical(\{High -> 0.9, Low -> 0.1\}),
9:     Mild -> Categorical(\{High -> 0.1, Low -> 0.9\}));
10: random DamageLevel Damage(City c) ~
11:   case Prep(c) in (High -> Categorical(\{Severe -> 0.2, Mild -> 0.8\}),
12:     Low -> Categorical(\{Severe -> 0.8, Mild -> 0.2\}));

13: distinct City A, B;
14: distinct PrepLevel Low, High;
15: distinct DamageLevel Severe, Mild;

16: **obs Damage**(First) = Severe;
17: **query Damage**(NotFirst);
```
Step debugging

```plaintext
debugger.step()
Entering: obs Damage(First) = Severe

debugger.step()
Entering: Damage(First)

debugger.step()
Entering: First
```

0: type City;
1: type PrepLevel;
2: type DamageLevel;
3: random City First ~ UniformChoice({c for City c});
4: random City NotFirst ~ UniformChoice({c for City c: c != First});
5: random PrepLevel Prep(City c) ~
6: if (First = c) then Categorical({High -> 0.5, Low -> 0.5})
7:   else case Damage(First) in
8:     Severe -> Categorical({High -> 0.9, Low -> 0.1}),
9:     Mild -> Categorical({High -> 0.1, Low -> 0.9});
10: random DamageLevel Damage(City c) ~
11:   case Prep(c) in {High -> Categorical({Severe -> 0.2, Mild -> 0.8}),
12:     Low -> Categorical({Severe -> 0.8, Mild -> 0.2});
13: distinct City A, B;
14: distinct PrepLevel Low, High;
15: distinct DamageLevel Severe, Mild;
16: obs Damage(First) = Severe;
17: query Damage(NotFirst);
debugger.step()
Entering: obs Damage(First) = Severe

ddebugger.step()
Entering: Damage(First)

ddebugger.step()
Entering: First

ddebugger.step()
Entering: UniformChoice({c for City c})

0: type City;
1: type PrepLevel;
2: type DamageLevel;
3: random City First ~ UniformChoice({c for City c});
4: random City NotFirst ~ UniformChoice({c for City c: c != First});
5: random PrepLevel Prep(City c) ~
6: if (First == c) then Categorical({High -> 0.5, Low -> 0.5})
7: else case Damage(First) in
8: (Severe -> Categorical({High -> 0.9, Low -> 0.1}),
9: Mild -> Categorical({High -> 0.1, Low -> 0.9}));
10: random DamageLevel Damage(City c) ~
11: case Prep(c) in (High -> Categorical({Severe -> 0.2, Mild -> 0.8}),
12: Low -> Categorical({Severe -> 0.8, Mild -> 0.2}));
13: distinct City A, B;
14: distinct PrepLevel Low, High;
15: distinct DamageLevel Severe, Mild;
16: obs Damage(First) = Severe;
17: query Damage(NotFirst);
Step debugging

debugger.step()
Entering: obs Damage(First) = Severe

debugger.step()
Entering: Damage(First)

debugger.step()
Entering: First

debugger.step()
Entering: UniformChoice({c for City c})

debugger.runToLine(11)
Entering: case Prep(c) in
{
    High -> Categorical({
        Severe -> 0.2,
        Mild -> 0.8}),
    Low -> Categorical({
        Severe -> 0.8,
        Mild -> 0.2})
}

0: type City;
1: type PrepLevel;
2: type DamageLevel;
3: random City First ~ UniformChoice({c for City c});
4: random City NotFirst ~ UniformChoice({c for City c: c != First});
5: random PrepLevel Prep(City c) ~
6: if (First == c) then Categorical({High -> 0.5, Low -> 0.5})
7: else case Damage(First) in
8:   (Severe -> Categorical({High -> 0.9, Low -> 0.1}),
9:     Mild -> Categorical({High -> 0.1, Low -> 0.9}));
10: random DamageLevel Damage(City c) ~
11:   case Prep(c) in (High -> Categorical({Severe -> 0.2, Mild -> 0.8}),
12:     Low -> Categorical({Severe -> 0.8, Mild -> 0.2}));
13: distinct City A, B;
14: distinct PrepLevel Low, High;
15: distinct DamageLevel Severe, Mild;
16: obs Damage(First) = Severe;
17: query Damage(NotFirst);
debugger.inspect('c');
Inspect: c
value: B
debugger.inspect('c');
Inspect: c
value: B

dbgger.inspect('Prep(c)');
Inspect: Prep(c)
value: Low
Inspect local variables

0: type City;
1: type PrepLevel;
2: type DamageLevel;

3: random City First ~ UniformChoice({c for City c});
4: random City NotFirst ~ UniformChoice({c for City c: c != First});
5: random PrepLevel Prep(City c) ~
6:   if (First == c) then Categorical({High -> 0.5, Low -> 0.5})
7:   else case Damage(First) in
8:     (Severe -> Categorical({High -> 0.9, Low -> 0.1}),
9:     Mild -> Categorical({High -> 0.1, Low -> 0.9}));
10: random DamageLevel Damage(City c) ~
11:   case Prep(c) in (High -> Categorical({Severe -> 0.2, Mild -> 0.8}),
12:                      Low -> Categorical({Severe -> 0.8, Mild -> 0.2}));

13: distinct City A, B;
14: distinct PrepLevel Low, High;
15: distinct DamageLevel Severe, Mild;

16: obs Damage(First) = Severe;
17: query Damage(NotFirst);

devger.inspect('c');
Inspect: c
value: B

debugger.inspect('Prep(c)');
Inspect: Prep(c)
value: Low

debugger.switchTo('Prep(B) == High')
switching to trace compatible: #175
Entering: obs Damage(First) = Severe
Advantages

• No need to know the queries before running inference
  ✦ Allows interactively querying of the posterior distribution

• `inspect(expr)` accepts any valid BLOG expression
  ✦ The generative model is made of BLOG expressions
  ✦ step-by-step debugging can be implemented by recursively inspecting the generative process

• Evaluate impact of information for any query
  ✦ Compute the posterior with different subsets of observations, and evaluate the expression in each sample of the posterior.
Impact of data

d debugger.plot_query('Prep(First) == High')
Impact of data

Prior

0: type City;
1: type PrepLevel;
2: type DamageLevel;

3: random City First ~ UniformChoice({c for City c});
4: random City NotFirst ~ UniformChoice({c for City c: c != First});
5: random PrepLevel Prep(City c) ~
6: if (First == c) then Categorical({High -> 0.5, Low -> 0.5})
7: else case Damage(First) in
8: (Severe -> Categorical({High -> 0.9, Low -> 0.1}),
9: Mild -> Categorical({High -> 0.1, Low -> 0.9}));
10: random DamageLevel Damage(City c) ~
11: case Prep(c) in (High -> Categorical({Severe -> 0.2, Mild -> 0.8}),
12: Low -> Categorical({Severe -> 0.8, Mild -> 0.2}));
13: distinct City A, B;
14: distinct PrepLevel Low, High;
15: distinct DamageLevel Severe, Mild;
16: obs Damage(First) = Severe;
17: query Damage(NotFirst);

Posterior

debbuger.plot_query('Prep(First) == High')
Impact of data

Bayesian Skill rating

debugger.plot_query('Skill(A) > 100')

Impact of data

\[ \text{obs Winner}(G1, A, B) = A; \]

Bayesian Skill rating

```
debugger.plot_query('Skill(A) > 100')
```

Impact of data

\[ \text{obs } \text{Winner}(G1, A, B) = A; \]
\[ \text{obs } \text{Winner}(G2, B, C) = B; \]

Bayesian Skill rating

```
debugger.plot_query('Skill(A) > 100')
```

Impact of data

\[
\text{obs } \text{Winner}(G_1, A, B) = A; \\
\text{obs } \text{Winner}(G_2, B, C) = B; \\
\text{obs } \text{Winner}(G_3, A, C) = A;
\]

Bayesian Skill rating

```
debugger.plot_query('Skill(A) > 100')
```

Impact of data

Prior

Impact of data

Posterior

Bayesian Skill rating

debugger.plot_query('Skill(A) > 100')

Final remarks

• This debugger is not meant to find bugs, but as a tool for understanding the program—the probabilistic model.

• It uses information that Bayesian PPL (e.g. anglican) uses:
  ✦ Information regarding sample and observe are stored in *addresses*
  ✦ Taking subsets of observations is a fundamental step for Sequential Monte Carlo sampling

• So it can be implemented in other languages.
query Questions?
Appendix A: Bayesian Skill rating model

type Player;
type Game;
distinct Player A, B, C;
distinct Game G1, G2, G3;

random Real Skill(Player p) ~ Gaussian(100.0, 10.0);
random Real Performance(Player p, Game g) ~ Gaussian(Skill(p), 15.0);
random Player Winner(Game g, Player p1, Player p2) ~
    if (Performance(p1, g) > Performance(p2, g))
        then p1
        else p2;

obs Winner(G1, A, B) = A;
obs Winner(G2, B, C) = B;
obs Winner(G3, A, C) = A;

Appendix B: backward and forward inference

- Region R is learned using backward inference
- Region Q is learned lazily using forward inference