Wolfe

www.wolfe.ml

Strength Reduction and Approximate Programming for Probabilistic Programming

Sebastian Riedel
University College London

Sameer Singh
University of Washington

Vivek Srikumar
Stanford University

Tim Rocktäschel
University College London

Larysa Visengeryeva
Technische Universität Berlin

Jan Noessner
University of Mannheim

Yet another PPL?

Existing PPLs pick a "representation":
- Undirected Graphical Models
- Bayesian Models
- Markov Logic Networks
- Other Logic-based formulations

Advantages:
+ Precisely defines the semantics
+ Easy to compile/optimize for efficiency

But it can be restrictive:
- Practical models may not be possible
- Cannot be future-proofed
- May not be concise for all applications
- Cannot easily combine with other PPLs

"Bring probabilistic programming as close to the underlying math as possible."

- Math is concise, precise, universal
- Can represent current & future models
- Allows combination of different paradigms in the same framework

Wolfe

Akin to machine learning math, a Wolfe **probabilistic program** consists of a set of **scalar functions** (for the model and loss), and a small **set of operators** that are applied to them to define inference/ learning. Given such a **mathematical description** in a **functional language**, Wolfe converts the operator applications to **efficient runtime code**.

Components | Linear Chains | Wolfe Code

Search Space

Define *all* possible values.

$$c = \{\ldots(x_i, y_i)\ldots\}, c \in \mathcal{C}$$

```
case class Chain(x:Seq[String],y:Seq[String])
def chains = seqs(strings) x seqs(strings)
```

Scalar Functions

Define real-value functions over the search space to define models (energy or density) and objectives.

$$\phi : \mathcal{C} \to \mathcal{R}^d$$

$$\phi(c) = \sum_{i=0}^{n} e_{\text{obs},x_i^c,y_i^c} + \sum_{i=0}^{n} e_{\text{trans},y_i^c,y_{i+1}^c}$$

$$m_w(c) = w \cdot \phi(c)$$

$$\text{where, } w : \mathcal{R}^d, m : \mathcal{R}^d \times \mathcal{C} \to \mathcal{R}$$

```
def features(c: Chain) = {
  val n = s.x.size
  sum(0 until n) {
    i=>oneHot('obs->s.x(i)->s.y(i))} +
  sum(0 until n-1) {
    i=>oneHot('trans->s.y(i)->s.y(i+1))}
}
def m(w: Vector)(s: Chain) = w dot features(s)
```

Operators

Combine model and objectives with search space to define inference and learning.
Operators are: **`argmax`**, **`argmin`**, **`sum`**, **`map`**, **`logZ`**, and **`expect`**

$$h_w(x) = \underset{\forall c \in \mathcal{C}, x^c = x}{\arg\max}\, m_w(c)$$

$$L(\mathbf{C}, w) = \sum_{c \in \mathbf{C}} m_w\left(h_w(x^c)\right) - m_w(c)$$

$$w^* = \underset{\forall w \in \mathcal{R}^d}{\arg\min}\, L(\mathbf{C}, w)$$

$$\hat{\mathbf{C}} = \forall_{x \in \mathbf{X}_t} h_{w^*}(x)$$

```
def h(w: Vector)(x: Seq[String]) =
    argmax(chains st _.x==x){m(w)}
def loss(data: Seq[Chain])(w: Vector) =
    sum(data) { s => m(w)(h(w)(s.x)) - m(w)(s) }

val (train,test) = NLP.conll2000Data()
val w = argmin(vectors) { loss(train) }
val predicted = map(test) {h(w)}
```

Efficiency

Wolfe maintains efficiency due to:
- Analyzes code during **compile time**
  - no overhead at runtime
- Generated code is natively compiled
  - enables Scala compiler optimizations
- Allows users to **inject customizations**
  - using Scala **`@Annotations`**
- Uses efficient implementations
  - Gurobi for ILP, Factorie for learning
  - can be multi-core, GPUs, etc.

Current Status

Currently, compiles to a factor graph.
Inference:
- Sum/Max-Product BP
- Junction Tree Inference
- Gibbs Sampling
- Integer Linear Programming

Learning:
- Structure perceptron
- Batch Methods (LBFGS)
- Stochastic Approaches: SGD, AdaGrad, AROW, etc.

Future Work

- More inference & learning methods
  - generative, matrix factorization
- Deeper code analysis
  - more sophisticated pattern matching
- Use even more existing packages
  - efficient inference implementations
- Automatic derivatives
  - compute gradients automatically
- Interactive Debugging
  - browser-based visualization