Practical Machine Learning Using Probabilistic Programming and Optimization

Sameer Singh
University of Washington

Sebastian Riedel   Tim Rocktäschel   Luke Hewitt
University College London
Limitations of existing PPLs (Why NLP people don’t really use them)

- “Differently Expressive”, or “I also want to get the awesome results from ICML 2014!”
- Inefficient, or lack of support for existing (or latest/greatest) inference approaches
- Black-boxness, or “Fine, results suck. Should I (can I) change anything?”
Practical Probabilistic Programming

Probabilistic Program

Inference Results

Inference and Learning
Practical Probabilistic Programming

Expressive Models used in Machine Learning
Bayesian Networks, Markov Random Fields, Conditional Random Fields,
Matrix Factorization, Word Embeddings, Deep Neural Networks
Practical Probabilistic Programming

- Probabilistic Program
- Inference Results

Interface for Existing (and Future) Approaches

Inference and Learning
Practical Probabilistic Programming

Comprehend Results and Debug Models
Visualizing Distributions, Plate Notation, Inference Progress, Feature engineering, Hyper-parameter optimization, …
Factor Graph Models

- Models where distribution is specified by an undirected graphical model over variables and “factors”

\[
P(Y) = \frac{1}{Z} \exp \sum_c \phi_c(Y_c)
\]

- Often conditional and parameterized

\[
P_\theta(Y|X) = \frac{1}{Z_\theta(x)} \exp \sum_c \theta \cdot \psi_c(Y, X)
\]

- Partial support: Figaro, Church, MLNs, ProbLog...

- Factorie: orders of magnitude faster MCMC on big graphs
Practical Probabilistic Programming
Wolfe

Functional Probabilistic Programming for Declarative Machine Learning
Wolfe

Philip Wolfe

• founder of convex optimization and mathematical programming

Sriram: Optimization is more important than PPLs

• “We want to give users what they use!”
Wolfe Overview

1) Functional programs
   • scalar functions for density, loss/objective,…
   • special operators for inference/learning
     - argmax, logZ, expect

2) Wolfe Interpreter
   • find factorizations in expression trees
   • Replaces calls with efficient code

3) Native Language Compiler
   Compiles to efficient code

User Code
import wolfe._
def domain = ...
def model = prod ...
def mu = expect ...

Wolfe Interpreter
import wolfe._
def domain = ...
def model = factorGraph
def map = beliefProp ...

Actual Code
import wolfe._
def domain = ...
def model = factorGraph
def map = beliefProp ...

Scala Compiler
Wolfe: Language

• Inspired by “math” (for example Jason’s Dyna)
  • make programs look like equations in paper
  • universal: allow impossible things (but fail at compile time)
• But not a DSL!
  • Integrate with existing tools, codebases, and libraries
  • Don’t expect users to learn another language
  • Make use of existing compiler optimizations
Wolfe: Universe

- Space of elements of the universe: \( \text{Set}[T] \)
  - Booleans/Categories: \( \text{bools} = \text{Set}(\text{true}, \text{false}) \)
  - Infinite and Uncountable sets: \( \text{ints}, \text{doubles} \)
  - Iterables and Functions: \( \text{seqs}(\text{bools}), \text{maps}(\text{bools}, \text{ints}) \)
- Abstract Data Types (Structures)
  - All possible tuples: \( \text{all}(\text{bools}, \text{bools}) \)
  - Person(name: String, age: Int)
    - cases[Person](strings, ints)
- Conditioning: \( \text{space where cond} \) (same as “filter”)
  - persons where \_\.name==“Andy Gordon”
Wolfe: Functions

• Define the density function: $T \Rightarrow Double$
  
  • def flip(x:Boolean) = 0.5

• Easier: Unnormalized, log-probability
  
  • def uniform(x: Double) = 0.0 // or 1.0

• Parameterized distributions
  
  • def bernoulli(p)(x) = if(x) log p else log (1-p)
  
  • def coin(x) = bernoulli(0.5)(x)

• Model Compositions: def $f(x)(z) = g(x) + h(x)(z)$
Wolfe: Operators

- **sample**: \((\text{Set}[T])(T \rightarrow \text{Double}) \rightarrow T\)
  - \text{sample}(\text{bools})(\text{bernoulli}(0.2))
- **argmax**: \((\text{Set}[T])(T \rightarrow \text{Double}) \rightarrow T\)
- **expect**: \((\text{Set}[T])(T \rightarrow \text{Double})(T \rightarrow \text{Vec}) \rightarrow \text{Vec}\)
  - \text{expect}(\text{doubles st } _{>0})(\text{norm})(x \rightarrow x**2)
- **logZ**: \((\text{Set}[T])(T \rightarrow \text{Double}) \rightarrow \text{Double}\)
Wolfe: Inference

• Sampling and MAP Inference are straightforward

• Marginal Inference: \( T = \text{Seq}[(\text{Int, Double})] \)

• `expect(seqs)(model) { seq => oneHot(\'0 \to seq(\theta)) }`

• Discriminative learning: `model(w)(xy)`
  
  • Conditional Likelihood: `def cll(data)(w)

  • `sum(data){ d=> model(w)(d) - logZ(_.x==d.x)(model(w))}`

  • Maximize: `argmax(doubles) { w => cll(data)(w) }`
case class Token(word: String, topic: Int)

case class Doc(tokens: Seq[Token], theta: Map[Int, Double])

case class World(docs: Seq[Doc], phi: Seq[Map[String, Double]])

val alpha = 50.0, beta = 0.01

def lda(world: World) = {
  import world._
  prod(phi) { dir(_, beta) } *
  prod(docs) { d =>
    dir(d.theta, alpha) *
    prod(d.tokens) { t =>
      cat(t.topic, d.theta) *
      cat(t.word, phi(t.topic)) }
  }
}

\[
P_{\alpha,\beta}(W, Z, \theta, \phi) = \prod_{i=1}^{K} \text{Dir}_{\beta}(\phi_i) \prod_{d=1}^{M} \text{Dir}_{\alpha}(\theta_d) \prod_{d=1}^{N_d} \text{Cat}(Z_{d,t}|\theta_d) \text{Cat}(W_{d,t}|\phi_{Z_{d,t}})\]
Relational Model

case class World(smokes:Pred[Symbol], cancer:Pred[Symbol],
    friends: Pred[(Symbol, Symbol)])

def persons = List('anna, 'bob)
def worlds =
    cross[World](preds(persons), preds(persons), preds(friends))

def mln(world: World) = {
    sum(persons) { p => 1.5*I(smokes(p) --> cancer(p)) }
    + sum(persons) { p1 => sum(persons) { p2 =>
        1.1*I(friends(p1, p2) --> (smokes(p1) == smokers(p2)))
    }
}

def evidence(world: World) = world.smokes('anna) && world.friends('anna, 'bob)
def query(world: World) = oneHot(world.cancer('bob))

val mu = expect(worlds where evidence) {
    mln
} {
    query
}

Friends(person, person)
Smokes(person)
Cancer(person)
Smokes(x) => Cancer(x) 1.5
Friends(x,y) => (Smokes(x) <=> Smokes(y)) 1.1
def mln(world: World) = 
  sum(persons) { p => 1.5 * I(smokes(p) --> cancer(p)) }

case class A(smokes: Seq[Double], cancer: Seq[Double])

case class V(ents: Map[Symbol, Seq[Double]])

def mf(w: World)(a: A)(v: V) = 
  sum(persons){p => I(w.smokes(p))*(a.smokes dot v.ents(p))}
  + sum(persons){p => I(w.cancer(p))*(a.cancer dot v.ents(p))}

def joint(w: World)(a: A)(v: V) = mln(w) + mf(w)(a)(v)

Easily combine with existing models (or relearn parameters for them)
Featurized Model

case class Chain(x: Seq[String], y: Seq[String])
val Y = List("PER", "LOC", "ORG", "O")
def chains = cross(Chain){seqs(strings) x seqs(Y)}

def feats(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)(c:Chain):Double = w dot feats(c)

def h(w:Vector)(x:Seq[String]) =
  argmax(chains where _.x==x){ m(w) }

def loss(data:Seq[Chain])(w:Vector) =
  sum(data){ c => m(w)(h(w)(c.x))-m(w)(c) }

val (train,test) = NLP.conll2000Data()
val w_opt = argmin(vectors) { loss(train) }
val predictions = map(test) { h(w_opt) }

C =\{...{(x_i,y_i)}...\}, C \in \mathcal{C}
\phi : \mathcal{C} \rightarrow \mathbb{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)

where, w : \mathbb{R}^d, m : \mathbb{R}^d \times \mathcal{C} \rightarrow \mathbb{R}

h_w(x) = \arg\max_{C \in \mathcal{C}, x^C = x} m_w(C)

L(C, w) = \sum_{C \in \mathcal{C}} m_w \left( h_w(x^C) \right) - m_w(C')

w^* = \arg\min_{w \in \mathbb{R}^d} L(C_T, w)

\hat{C} = \forall_{x \in X_t} h_w^*(x)
Practical Probabilistic Programming

Probabilistic Program

Inference Results

Inference and Learning
Wolfe: Engine

\[
def \text{crf}(w)(y) = \sum(y) \{ i \mapsto w \cdot y(i) \cdot y(i-1) \}
\]

\[
\text{argmax(\text{seqs}))(ys \mapsto \text{crf}(w)(y))
\]

Factored Space  Factored Model (in the same way)

\[
\text{argmax(\text{doubles}))(w \mapsto \text{cond_like}(w)(yt))
\]

Continuous Space  Automatically Differentiate (requires inference)
Customizations

• Specifying the algorithm: at model or at inference
  • @ArgmaxBy(MaxProductBP, tolerance=1e-10)
    val best = argmax(strings)(model)

• Different approach for different components
  • @ExpectBy(Gibbs, samples=1000)
    def m1(x) = ....

  • @ExpectBy(BP, tolerance=1e-5)
    def m2(x) = ....

  • expect(space)(m1 + m2)(_)
Practical Probabilistic Programming

Probabilistic Program

Inference Results

Inference and Learning
Probabilistic Programming IDE

Challenges and a Prototype
Desiderata from an IDE

• Interactive, Visual Environment for Debugging PPLs

• Three kinds of questions:
  1. What is this? Visualize Input/Output Data
  2. How is it happening? Details of the Model
  3. Why exactly did this happen? Details of Inference

• Inspired by browser-based editors for WebPPL (probmods.org) and ProbLog tutorial
Input/Output Data

- Structured Data: Graphs, Matrices, etc.
Input/Output Data

- Distributions over Structured Data

- Open: Visualizing arbitrary structured distributions
Model Visualization

• Visualizing structure and the “provenance”

• Open: scaling to user expertise gracefully

• Going beyond Andy’s “ML PhD” class of users
Inference Visualization

• Aggregate statistics or individual steps

• **Open**: scaling to user expertise gracefully

• **Open**: detect “failures”, & suggest alternatives
Future Work

• Wolfe
  • Support for more models/inference algorithms
  • Distributed Inference backend
  • Specify what is supported, and what is not

• IDE for PPL
  • Structured Distribution Visualizations
  • More provenance from inference and model
  • Easily extensible: Scala Object => HTML
Thanks

http://wolfe.ml/demos