@AndrewDGordon, Microsoft Research and University of Edinburgh

Based on joint work with Mihhail Aizatulin (OU), Johannes Borgström (Uppsala), Guillaume Claret (MSR), Thore Graepel (MSR), Aditya Nori (MSR), Sriram Rajamani (MSR), and Claudio Russo (MSR)
Machine Learning and Programming

- “Data widely available; what is scarce is the ability to extract wisdom from them”, Hal Varian, 2010
- “Machine learning!”, Mundie and Schmidt at Davos, 2012

Researchers use Bayesian statistics as unifying principle:
- Models are conditional probabilities; inference algorithms separate

For the programmer, what’s the problem?
- Cottage industry of inflexible libraries and algorithms
- Custom implementations are 1000s LOC

Probabilistic programming offers a solution
- Write your model as succinct, adaptable probabilistic program
- Run compiler to get efficient inference code
Murder Mystery in Fun

// Either Alice or Bob dunnit
// Alice dunnit 30%, Bob dunnit 70%
// Alice uses gun 3%, uses pipe 97%
// Bob uses gun 80%, uses pipe 20%
let mystery () =
    let aliceDunnit = random (Bernoulli 0.30)
    let withGun =
        if aliceDunnit
            then random (Bernoulli 0.03)
        else random (Bernoulli 0.80)
    aliceDunnit, withGun

// Pipe at scene - now Alice dunnit 69%
let PipeFoundAtScene () =
    let aliceDunnit, withGun = mystery ()
    observe(withGun = false)
    aliceDunnit, withGun
Probabilistic Programming

- BUGS (Spiegelhalter et al 1994, CU)
- IBAL (Pfeffer, 2002)
- BLOG (Milch et al 2005, UCB/MIT) – Gibbs sampling
- Alchemy (Domingos et al 2005, UW) – probabilistic logic programming
- CHURCH (Goodman et al 2008, MIT) – recursive probabilistic functional programming
- HANSEI (Kiselyov and Shan, 2009) – discrete distributions from Ocaml
- FACTORIE (McCallum et al 2008, UMASS)
- Infer.NET
Judea Pearl, Turing Award Winner 2011

For fundamental contributions to artificial intelligence through the development of a calculus for probabilistic and causal reasoning.

... He identified uncertainty as a core problem faced by intelligent systems and developed an algorithmic interpretation of probability theory as an effective foundation for the representation and acquisition of knowledge.
Probabilistic Graphical Models

- Pioneered by Bayes Networks (Pearl 1988)
  - Model of world, both observed and unobserved states
  - Probabilistic for uncertainty: missing data, noise, how data arises
  - Graphical notations capture dependence, for scalability

- Pearl “invented message-passing algorithms that exploit graphical structure to perform probabilistic reasoning effectively”

- Many application areas: “natural language processing, speech processing, computer vision, robotics, computational biology, and error-control coding”

- In last few years, large-scale deployments include:
  - TrueSkill – How do we rank Halo players?
  - AdPredictor – How likely is a user to click on this ad?
Infer.NET (since 2006)

- A .NET library for probabilistic inference
  - Multiple inference algorithms on graphs
  - Far fewer LOC than coding inference directly
  - Designed for large scale inference
  - User extensible
- Supports rapid prototyping and deployment of Bayesian learning algorithms
  - Graphs represented by object model for pseudo code, but not as runnable code
- **Realization**: language geeks can do machine learning, without comprehensive understanding of Bayesian stats, message-passing, etc
Infer.NET Fun – New Feature

- Bayesian inference by functional programming
  - Write your model in F#
  - Run forwards to synthesize data
  - Run backwards to infer parameters

- Benefits:
  - Models are simply code in F#’s simple succinct syntax
  - Higher-level features than C# OM: tuples, records, array comprehensions, functions
  - Custom graphical notations (“plates”, “gates”) just code
  - Testing inference by running forwards then backwards

http://research.microsoft.com/fun
Linear Regression

- Linear regression:
  Forwards, compute \( y_i = ax_i + b + \text{noise} \) from \( a \) and \( b \)
  Backwards, given \( y_i \) infer \( a \) and \( b \)

true a: -1.422354626
true b: 7.171306243
true prec: 0.1829893437
Linear Regression in Fun

```plaintext
let prior() =
    let a = random(Gaussian(0.0, 1.0))
    let b = random(Gaussian(5.0, 0.3))
    let noise = random(Gamma(1.0, 1.0))
    a, b, noise

let point x a b noise =
    x, random(Gaussian(a * x + b, noise))

let model data =
    let a, b, noise = prior()
    observe (data =
        [|| for x,_ in data -> point x a b noise ||])
    a, b, noise

let aD, bD, noiseD = inferFun3 @@ model @ data
```
Some Probability Distributions in Fun

type ::= // Fun value type
  unit
  bool
  int
  double
  (type1 * ... * typeN)
  { field1: type1; ...; fieldN: typeN}
  type[]

expr ::= // Fun expression
  var // variable
  literal // literal eg -1.0, true, 42
  { field1 = expr1; ...; fieldN = exprN } // record
  ( expr1, ..., exprN ) // tuple
  expr.field // record lookup
  fst(expr) // first projection
  snd(expr) // second projection
  not expr // negation
  expr1 R expr2 // relation (eg, =, >)
  expr1 f expr2 // function (eg, +, -)
  let var = expr1 in expr2 // let
  if expr1 then expr2 else expr3 // conditional
  expr : type // type annotation
  for var in expr1 do expr2 // iteration loop
  [| 0 .. expr |] // integer range
  [| for var in expr1 -> expr2 |] // comprehension
  Array.zip expr1 expr2 // zip two arrays
  random(dist) // draw from distribution
  observe expr // observation of boolean
TrueSkill in Fun

// prior distributions, the hypothesis
let skill() = sample (Gaussian(10.0, 20.0))
let Alice, Bob, Cyd = skill(), skill(), skill()
TrueSkill in Fun

// prior distributions, the hypothesis
let skill() = sample (Gaussian(10.0, 20.0))
let Alice, Bob, Cyd = skill(), skill(), skill()

// observe the evidence
let performance player = sample (Gaussian(player, 1.0))
observe (performance Alice > performance Bob) // Alice beats Bob
observe (performance Bob > performance Cyd) // Bob beats Cyd
observe (performance Alice > performance Cyd) // Alice beats Cyd

// return the skills
Alice, Bob, Cyd
The **model-learner** pattern brings structure and types, as well as PL syntax, to probabilistic graphical models.

```
module LinearRegression =
    type TH = {MeanA: double; PrecA: double; ... }
    let h = {MeanA=0.0; PrecA=1.0; ... }
    type TW<'a,'b,'c> = {A:'a; B:'b; Noise:'c}
    type TX = double
    type TY = double
    let M: Model<TH,TW<double,double,double>,TX,TY>
```

Write your model in F# or C#.

Or choose from library.

Or automatically generate.

Assemble multiple models.

Choose algorithm (eg, EP, VMP, Gibbs, ADD, Filzbach).

Train, predict, repeat.

http://research.microsoft.com/fun
Models, Samplers, and Learners

type Model<'TH,'TW,'TX,'TY> =
  { HyperParameter: 'TH
  Prior: Expr<'TH ->'TW>
  Gen: Expr<'TW *'TX ->'TY> }

type ISampler<'TW,'TX,'TY> =
  interface
    abstract Parameters: 'TW
    abstract Sample: x:'TX -> 'TY
  end

type ILearner<TDistW,'TX,'TY,'TDistY> =
  interface
    abstract Train: x:'TX * y:'TY -> unit
    abstract Posterior: unit -> 'TDistW
    abstract Predict: x:'TX -> 'TDistY
  end
TrueSkill

let perf(w,pid) =
    let m = w.Skills.[pid]
    Fun.random(Fun.GaussianFromMeanAndPrecision(m,1.0/beta2))

let M:Model<TH,TW<real>,TX,TY> =
    { HyperParameter = {Players = 4
        GM = {Mean=25.0;Precision=1.0/sigma2} } 
    Prior = <@ fun h ->
        {Skills =
            [ | for x in 0..h.Players-1 ->
                let m,p = h.GM.Mean,h.GM.Precision in
                Fun.random(Fun.GaussianFromMeanAndPrecision(m,p)) | ]
        } @>
    Gen = <@ fun (w,x) -> (perf(w,x.P1) > perf(w,x.P2)) @>}

Binary Mixture Combinator

- We code a variety of idioms as functions from models to models, eg, mixtures:

```plaintext
let Mixture(m1,m2) =
{Prior =
@@ fun h ->
{Bias=random(Uniform(0.0,1.0))
P1=(%m1.Prior) h
P2=(%m2.Prior) h}@>
Gen =
@@ fun (w,x) ->
if random(Bernoulli(w.Bias))
then (%m1.Gen) (w.P1,x)
else (%m2.Gen) (w.P2,x) @>}
```
let $k = 4$ // number of clusters in the model
let $M = \text{IIDArray.M}(\text{KwayMixture.M}(\text{VectorGaussian.M}, k))$

let $\text{sampler1} = \text{Sampler.FromModel}(M)$;
let $xs = [\mid \text{for} \ i \ \text{in} \ 1..100 \ \text{->} \ () \ \text{]}$
let $ys = \text{sampler1.Sample}(xs)$;

let $\text{learner1} = \text{InferNetLearner.LearnerFromModel}(M, mg0)$
do $\text{learner1.Train}(xs, ys)$
let $(\text{meansD2}, \text{precsD2}, \text{weightsD2}) = \text{learner1.Posterior}()$
Evidence Combinator

- A variation of mixtures, where the choice between models is made per-model, rather than per-output

```plaintext
let Evidence(m1,m2) =
    {Prior = @ fun (bias,h1,h2) ->
      (random(Bernoulli(bias))),
      (%m1.Prior) h1, (%m2.Prior) h2) @>
    Gen = @ fun ((switch,w1,w2),x) ->
      if switch then (%m1.Gen) (w1,x)
      else (%m2.Gen) (w2,x) @>}
```
Demo: Model Selection

let mx k = NwayMixture.M(VectorGaussian.M,k)
let M2 = Evidence.M(mx 3, mx 6)
Fitting Model to Climate Data (TACAS’13)

- We developed scientific models as Fun models
- One benefit is the automatic extraction of the likelihood function as the density of a probabilistic expression

```ocaml
module NPP = 
  let predict w x = 
    let prec_lim = w.max_NPP * (1.0 - exp (-w.p * x.MAP)) 
    let temp_lim = w.max_NPP / (1.0 + exp (w.t1 - w.t2 * x.MAT)) 
    let pred_NPP = min prec_lim temp_lim 
    pred_NPP 

  let model = 
    {Prior = 
      @@ fun () -> 
        {max_NPP = random(Gamma(1.0, 1.0)) 
          p = random(Gamma(1.0, 1.0)) 
          t1 = random(Gamma(1.0, 1.0)) 
          t2 = random(Gamma(1.0, 1.0)) 
          s_NPP = random(Gamma(1.0, 1.0))} @@} 

  Gen = 
    @@ fun (w,x) -> 
      {NPP = random(Gaussian(predict w x, 
                                w.s_NPP * w.s_NPP))} @@
```
Infer.NET Fun

- Bayesian inference by functional programming
  - Write your model in F#
  - Run forwards to synthesize data (normal F#)
  - Run backwards to infer parameters (via Infer.NET)

- Benefits:
  - Models are simply code in F#’s simple succinct syntax
  - Higher-level features than core Infer.NET: tuples, records, array comprehensions, and functions

- A wide range of efficient algorithms for regression, classification, and specialist learning tasks derive by probabilistic functional programming.

- Papers, download available: http://research.microsoft.com/fun
Challenges
Three Challenges

- Poor usability could be a show-stopper
- Fragmentation
- Potential beneficiaries may not have the time, inclination, or aptitude to learn to write and debug probabilistic programs.
Pain Points of Probabilistic Programming

- 15%: “Complicated object model in language/library syntax and type system.”
- 15%: “Gap between declarations and operational semantics.”
  - “You can write graphical models that make sense but can’t execute due to internal details of the engines.”
- 20%: “Tuning is time-consuming” (parameters/algorithm selection, no. of iterations).”
  - “I spent most of my time on robustness; setting hyperparameters and the priors.”
- 20%: “Performance” (cost of model in memory, perf impact of designs), scalability.”
  - “It would be nice if a simple annotation could inform the model of how to batch elements.”
- 30%: “Understanding inference results is hard.”
  - “Once you have a model running, there’s no explanation for the inference, hard to find whether issues come from modelling, features, parameters, or the data deficiencies.”

Is there better data? Should we gather more to create a baseline?
Probabilistic Metaprogramming

- Singh and Graepel’s InfernoDB

![Diagram of user and movie data structures with probabilistic metaprogramming model](image)
Questions?