Andy Gordon (MSR and University of Edinburgh)

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)
What next?
Bayesian Models

- We **train** on the observed inputs and outputs to **learn** the parameters, and to **predict** new outputs on unseen inputs.
- **Bayesian** models capture uncertainty about model components as probability distributions.

A Model

\[ y = Ax + B + e \]

where noise \( e \sim N(0, P) \)

\( x \) is an **input**, \( y \) is an **output**. \( A, B, P \) are the model **parameters**.
Five Distributions

- **Prior** distribution: \( p(w) \)
given by \( w = (A, B, P), A \sim N(0,1), B \sim N(0,1) \) and \( P \sim \Gamma(1,1) \)

- **Sampling** distribution: \( p(y|x, w) \)
given by \( y \sim N(Ax + B, P) \) for \( w = (A, B, P) \)

- **(Prior) Predictive** distribution:
\[
p(y|x) = \int p(y|x, w) \, p(w) \, dw
\]

- **Posterior** distribution, given training data \( d = (x, y) \):
\[
p(w|d) = \frac{p(y|x, w) \, p(w)}{p(y|x)}
\]

- **Posterior predictive** distribution, given \( d = (x, y) \):
\[
p(y'|x', d) = \int p(y'|x', w) \, p(w|d) \, dw
\]
Three Classes of Bayesian Inference

\[ p(w|d) = \frac{p(y|x, w)p(w)}{p(y|x)} \text{ where } d = (x, y) \]

- **Exact inference** for discrete distributions:
  Representation: enumerations of probabilities
  Example: \([HH, \frac{1}{10}; HT, \frac{2}{10}; TH, \frac{7}{10}; TT, 0]\)

- Approximate inference: sampling eg **Markov chain Monte Carlo**:
  Representation: finite ensemble of samples
  Example: \([A = 1.7, B = 1.6; A = 9.9, B = 9.8 ; ... ]\)

- Approximate inference: **belief propagation on factor graphs**:
  Representation: parameters for marginal of each variable
  Example: \([A = N(5.1,10), B = N(6.0,5)]\)
Bayesian Models are Widely Applicable

- Many machine learning tasks may be cast as Bayesian models.
- We infer functions from inputs to outputs, governed by uncertain parameters.
- Examples include:
  - A regression function inputs a tuple of independent variables, and produces one (or more) dependent variables (typically continuous).
  - A classifier inputs a vector of features and outputs a single value, the class (typically discrete).
  - A cluster analysis groups items so that items in each cluster are more like each other than to items in other clusters.
  - A recommender predicts the rating or preference that a user would give to an item (such as music, books, or movies) based on previous ratings by a set of users.
  - A rating system assesses a player's strength in games of skill (such as chess or Go) based on observed game outcomes.
Promise of Probabilistic Programming

- Custom inference code is hard to write, depends on mechanism
- Instead, user writes a probabilistic model for a Bayesian inference problem as a short piece of code, while the compiler turns this code into an efficient inference routine.
- Systems include BUGS, IBAL, BLOG, Church, STAN, Infer.NET, Fun, Factorie, Passage, HBC, HANSEI, and more.
- Still, no linguistic abstractions for Bayesian models.
- **Our contribution:** a new typed model abstraction to represent a function from $X$ to $Y$, governed by $W$:
  - may be composed to form richer models
  - via a sampler, may be run to draw from predictive distribution
  - via a learner, may be trained to make predictions
Distributions (1-3) as Probabilistic Code

- **Prior** distribution: \( p(w|h) \) for **hyperparameter** \( h \):

  ```
  let prior (h:TH) =
  { A = random (Gaussian(h.MeanA, h.PrecA))
  B = random (Gaussian(h.MeanB, h.PrecB))
  P = random (Gamma(h.Shape, h.Scale))} : TW
  ```

- **Sampling** distribution: \( p(y|x, w) \)

  ```
  let gen(w,x) =
  [ | for xi in x -> random(Gaussian(w.A * xi + w.B, w.P)) ]
  ```

- **(Prior) Predictive** distribution:

  \[
  p(y|x, h) = \int p(y|x, w) p(w|h) \, dw
  \]

  ```
  let predictive(h,x) = let w = prior h in gen (w,x)
  ```
Distributions (4-5) as Probabilistic Code

- **Posterior** distribution, \( p(w|d, h) \) where \( d = (x, y) \):
  \[
p(w|d, h) = \frac{p(y|x, w) p(w, h)}{p(y|x, h)}
  \]

  ```
  let posterior (h,x,y) =
  let w = prior h in
  observe (y = gen (w,x)); w
  ```

- **Posterior predictive** distribution:
  \[
p(y'|x', d, h) = \int p(y'|x', w) p(w|d, h) \, dw
  \]

  ```
  let posterior_predictive (h,x,y,x') =
  let w = posterior (h,x,y) in
  gen (w,x')
  ```
Inference on Probabilistic Code

- F# quotations represent probabilistic code:

```
let d = @{ fun m -> (random(Gaussian(m,1.0), random(Bernoulli(0.5))) @> : Expr<double -> double * bool>
```

- Infer.NET’s inference invoked by a dynamically typed function, returning a marginalized representation `marginal(‘U)`

```
infer d 42.0 : Gaussian * Bernoulli
```

- Hence, we train our linear regression example:

```
let wD:{A=Gaussian;B=Gaussian;P=Gamma} =
    infer <@ posterior @> (x,y)
let yD:Gaussian[]= =
    infer <@ posterior_predictive @> (x,y,x)
```
Abstraction 1: Model

- A model represents a probabilistic function from TX to TY, governed by an uncertain, learnable TW parameter, and a fixed TH hyperparameter.

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

```plaintext
{ HyperParameter = {MeanA=0.0; PrecA=1.0; ... } 
  Prior = @@ fun h ->
  { A = random(Gaussian(h.MeanA,h.PrecA))
    B = random(Gaussian(h.MeanB,h.PrecB))
    P = random(Gamma(h.ShapeN,h.ScaleN)) } @>
  Gen = @@ fun (w,x) -> [ | for xi in x ->
    random(Gaussian(w.A * xi + w.B, w.P))]|] @> }
```
Abstraction 2: Sampler

- A sampler is an object obtained from a model for sampling from the **prior** and **(prior) predictive** distributions, simply by running the code.

```typescript
type ISampler<'TW,'TX,'TY> =
  interface
    abstract Parameters: 'TW
    abstract Sample: x:'TX -> 'TY
end
```
Abstraction 3: Learner

- A learner is an object obtained from a model and an inference method, for computing the posterior and posterior predictive distributions, after training.

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

We have three efficient learners:
- Exact (ADD/CUDD): algebraic decision diagrams
- MCMC (Filzbach): ensembles of samples
- Factor graphs (Infer.NET): marginal parametric distributions

```csharp
type ReferenceLearner(m) =
    let mutable d = <@ (%m.Prior) (%m.HyperParameter) @>
interface I Learner<Expr<'TW>, 'TX, 'TY, Expr<'TW> with
    member l.Train(x, y) =
        d <- <@ let w = %d in observe(y = (%m.Gen)(w, x)); w @>
    member l.Posterior() = d
    member l.Predict(x) = <@ let w = %d in (%m.Gen)(w, x) @>
```
## Three Examples

<table>
<thead>
<tr>
<th>Linear Regression</th>
<th>BPM Classifier</th>
<th>TrueSkill</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>TH</strong></td>
<td>{MeanA: double; PrecA: double; … }</td>
<td>{Ncols:int}</td>
</tr>
<tr>
<td><strong>TW</strong></td>
<td>{A:double; B:double; Noise:double}</td>
<td>{Noise: double; Weights: Vector}</td>
</tr>
<tr>
<td><strong>TX</strong></td>
<td>double</td>
<td>Vector</td>
</tr>
<tr>
<td><strong>TY</strong></td>
<td>double</td>
<td>bool</td>
</tr>
<tr>
<td><strong>Posterior</strong></td>
<td>{A:Gaussian; B:Gaussian; Noise:Gamma}</td>
<td>{Noise:Gaussian, Weights:VectorGaussian}</td>
</tr>
<tr>
<td><strong>Predict</strong></td>
<td>double -&gt; Gaussian</td>
<td>Vector -&gt; Bernoulli</td>
</tr>
</tbody>
</table>
Generic Loopback Function

- Given these abstractions, we can write generic machine learning code, such as **loopback testing**

```haskell
let test (toLearner: Model<'TH,'TW,'TX,'TY> -> ILearner<'DistW,'TX,'TY,'DistY>)(m:Model<'TH,'TW,'TX,'TY>) (x:'TX): 'TW * 'DistW =
  let S = Sampler.FromModel(m)
  let y = S.Sample(x)
  let L = toLearner(m)
  do L.Train(x,y)
  (S.Parameters,L.Posterior())
```
Array Combinator

- Allows training and prediction on IID data

```fsharp
module IIDArray =
let M(m:Model<'TH,'TW,'TX,'TY>)
    : Model<'TH,'TW,'TX[],'TY[]> =
  { Prior = m.Prior
    Gen = <@
      fun (w,x) ->
        [| for xi in x -> (%m.Gen) (w,xi) |] @> }
```
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 = IIDArray.M(KwayMixture.M(VectorGaussian.M,k))

let sampler1 = Sampler.FromModel(M);
let xs = [| for i in 1..100 -> () |]
let ys = sampler1.Sample(xs);

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

```
let M(m1,m2) =
    {Prior = <@ fun (bias,h1,h2) ->
        (breakSymmetry(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

```fsharp
let mx k = NwayMixture.M(VectorGaussian.M,k)
let M2 = Evidence.M(mx 3, mx 6)
```
A Dozen Models

<table>
<thead>
<tr>
<th>Example / Learner</th>
<th>TH</th>
<th>TW</th>
<th>TDistW</th>
<th>TX</th>
<th>TY</th>
<th>TDistY</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sprinkler / A</td>
<td>SP.TH</td>
<td>SP.TW&lt;bool&gt;</td>
<td>ADD&lt;SP.TW&lt;bool&gt;&gt;</td>
<td>SP.TX</td>
<td>bool</td>
<td>ADD&lt;bool&gt;</td>
</tr>
<tr>
<td>TwoCoins / A</td>
<td>TC.TH</td>
<td>TC.TW&lt;bool&gt;</td>
<td>ADD&lt;TC.TW&lt;bool&gt;&gt;</td>
<td>TC.TX</td>
<td>bool</td>
<td>ADD&lt;bool&gt;</td>
</tr>
<tr>
<td>Two Coins / IN</td>
<td>TC.TH</td>
<td>TC.TW&lt;bool&gt;</td>
<td>TC.TW&lt;Bernoulli&gt;</td>
<td>TC.TX</td>
<td>bool</td>
<td>Bernoulli</td>
</tr>
<tr>
<td>Friends / A</td>
<td>bool[]</td>
<td>bool list list</td>
<td>ADD&lt;bool list list&gt;</td>
<td>int * int</td>
<td>bool</td>
<td>ADD&lt;bool&gt;</td>
</tr>
<tr>
<td>Students / A</td>
<td>int * int</td>
<td>bool list list</td>
<td>ADD&lt;bool list list&gt;</td>
<td>int * int</td>
<td>bool</td>
<td>ADD&lt;bool&gt;</td>
</tr>
<tr>
<td>Gaussian / IN</td>
<td>GM.TH</td>
<td>GM.TW&lt;β,γ&gt;</td>
<td>GM.TW&lt;β,γ&gt;</td>
<td>unit</td>
<td>real</td>
<td>N</td>
</tr>
<tr>
<td>Gaussian Mix / IN</td>
<td>MX1.TH</td>
<td>β<em>GaussW</em>GaussW</td>
<td>β*GM.TW&lt;β,γ&gt;</td>
<td>unit</td>
<td>real</td>
<td>N</td>
</tr>
<tr>
<td>Gaussian Mix / F</td>
<td>MX2.TH</td>
<td>GaussW*GaussW</td>
<td>(GaussW*GaussW)[]</td>
<td>unit</td>
<td>real</td>
<td>F[]</td>
</tr>
<tr>
<td>PlantGrowth / F</td>
<td>PG.TW</td>
<td>TS.TW&lt;γ&gt;</td>
<td>TS.TW&lt;γ&gt;</td>
<td>int</td>
<td>real</td>
<td>F[]</td>
</tr>
<tr>
<td>TrueSkill / IN</td>
<td>TS.TH</td>
<td>TS.TW&lt;γ&gt;</td>
<td>TS.TW&lt;γ&gt;</td>
<td>TrueSkill.TX</td>
<td>bool</td>
<td>Bernoulli</td>
</tr>
<tr>
<td>Lin. Reg. / IN</td>
<td>L.R.TH</td>
<td>L.R.TW&lt;β,γ&gt;</td>
<td>L.R.TW&lt;β,γ&gt;</td>
<td>real</td>
<td>real</td>
<td>N</td>
</tr>
<tr>
<td>MV Gaussian / IN</td>
<td>MVG.TH</td>
<td>MVG.TW&lt;β,γ&gt;</td>
<td>MVG.TW&lt;β,γ&gt;</td>
<td>unit</td>
<td>real</td>
<td>N</td>
</tr>
</tbody>
</table>

$β$ = real $N$ = Gaussian $β$ = Beta $ γ$ = Gamma $R$ = Vector $M$ = PositiveDefiniteMatrix $\mathcal{W}$ = Wishart /// generalizes $γ$ to multiple dimensions $\mathcal{N}$ = VectorGaussian /// multivariate Gaussian distribution

| GaussW = | Mean: β; Precision: γ |
| BetaW = | trueCount: β; falseCount: β |
| SP.TH = | (Rain: β; Sprinkler: β) |
| SP.TW<‘TB’> = | (Rain: ‘TB; Sprinkler: ‘TB) |
| SP.TX = | IsGrassWet /// a unit type |
| TC.TH = | (Bias1: β; Bias2: β) |
| TC.TW<‘TB’> = | (Heads1: ‘TB; Heads2: ‘TB) |
| TC.TX = | AreEitherHeads /// a unit type |
| GM.TW<‘TM,‘TP’> = | (Mean: ‘TM; Precision: ‘TP) |
| GM.TH = | Gaussian: GaussW, Gamma: GammaW |

MX1.TH = BetaW * GM.TH * GM.TH
MX2.TH = GM.TH * GM.TH
PG.TW = { alpha: β; topc: δ; trho: γ; imass: δ; sigma: β }
TS.TH = { Players: int; G: GaussW; P: GammaW }
TS.TX = { P1: int; P2: int }
LR.TH = { MeanA: β; PrecA: δ; MeanB: β; PrecB: δ; Shape: δ; Scale: β }
MVG.TH = { NCols: int; MeanVectorPrecisionCount: β; WishartShapeConstant: δ; WishartScaleConstant: δ }
MVG.TW<‘TM’, ‘TC’> = { Mean: ‘TM; Covariance: ‘TC }

Table 1. Rows show types for $L_{1}$Learner( TDistW, TX, TY, TDistY ) for $m_{1}$Model< TH, TW, TX, TY > (A=ADD, IN=Incr, NET, F=Filzbach)
Related and Future Work

- Roger Grosse’s compositional theory of Bayesian image processing UAI 2012, plus greedy model selection algorithm – fits model-learner pattern.

- Extend our learner API to support partially observed output, eg, for Naïve Bayes or Hidden Markov Models.

- Completeness? Which Bayesian models don’t fit?

- **Probabilistic metaprogramming** refers to automatic techniques for constructing probabilistic programs.

- Next, we aim to develop schema-directed probabilistic metaprogramming for inference on databases, an area in its infancy (cf Singh and Graepel’s InfernoDB).
The **model-learner** pattern brings structure and types, as well as PL syntax, to probabilistic graphical models.

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)

**type** `Model`

**type** `ISampler`

**type** `ILearner`

Train, predict, repeat

Synthetic data to test learner

http://research.microsoft.com/fun
The Paper

- The new conceptual insight is that code-based machine learning can be structured around typed Bayesian models, which are pairs of expressions representing prior and sampling distributions.
  - Definition of a type of Bayesian models, with combinators for compositionally constructing models, and operations to derive samplers and learners from an arbitrary model.
  - Many Bayesian examples expressed as models.
  - A formal semantics for models, learning, and prediction in Fun, and its semantics using measure transformers and probability monad.
  - Learners based on Algebraic Decision Diagrams, message-passing on factor graphs, and Markov chain Monte Carlo.
Questions?
Infer.NET (since 2006)

- Tom Minka, John Winn, John Guiver, and others
- 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
Some Probability Distributions in Fun