Probabilistic Programming

Miguel Lázaro-Gredilla
miguel@tsc.uc3m.es

January 2013
Machine Learning Group
http://www.tsc.uc3m.es/~miguel/MLG/

Contents

Classical machine learning

- Large number of tools with diverse backgrounds
  - k-means
  - Principal Component Analysis
  - Independent Component Analysis
  - Classical Neural Networks
  - Support Vector Machines
  - Density estimation via Parzen windows
  - Recursive Least Squares
  - Least Mean Squares
  - (just an arbitrary sample, we could go on and on...)
- Pragmatic, unsystematic, non-probabilistic

Third generation machine learning

- Data sets described as instances of a probabilistic model
  - Gaussian Process regression/classification
  - Latent Dirichlet Allocation
  - Bayes Point Machine
  - ...
- We can infer unknowns in a principled way
- Features
  - Each tool can be expressed as a Bayesian network
  - Systematic, modular, standardized approach
  - Proposals are models, not algorithms
  - Detach inference from model

Updating classical machine learning (I/V)

- Should we just dump classical ML and jump on the Bayesian bandwagon?
- Most classical ML tools can be written as some type of inference on some Bayesian network
- So just update classical ML using a Bayesian interpretation
  - Gain insight on the model behind the algorithm
  - See overlaps emerge
  - Use other types of inference
  - It may become obvious how to enhance them

Updating classical machine learning (II/V)

- Classical algorithm: k-means
- Bayesian model (assuming normalized data)

$$
\begin{aligned}
p(\mathbf{x}_n|z_n, \{\boldsymbol{\mu}_k\}) &= \mathcal{N}(\mathbf{x}_n|\boldsymbol{\mu}_{z_n}, v\mathbf{I}) \\
p(v) &= \text{InvGamma}(v|1,1) \\
p(\boldsymbol{\mu}_k) &= \mathcal{N}(\boldsymbol{\mu}_k|\mathbf{0}, 10\mathbf{I}) \\
p(z_n|\mathbf{w}) &= \text{Discrete}(z_n|\mathbf{w}) \\
p(\mathbf{w}) &= \text{Dirichlet}(\mathbf{w}|\mathbf{1}_{k\times 1})
\end{aligned}
$$

- Classical model corresponds to
  - Maximum likelihood for $p(\{\mathbf{x}_n\}|\{z_n\}, \{\boldsymbol{\mu}_k\})$, obtained using hard-EM optimization
  - Particular case of Gaussian mixture model

Updating classical machine learning (III/V)

- Classical algorithm: Extended Recursive Least Squares (adaptive)
- Bayesian model

$$
\begin{aligned}
p(x_n|\mathbf{w}_n) &= \mathcal{N}(\mathbf{x}_n|\mathbf{w}_n^\top \mathbf{u}_n, v) \\
p(\mathbf{w}_n|\mathbf{w}_{n-1}, \alpha, \beta) &= \mathcal{N}(\mathbf{w}_n|(1-\beta)\mathbf{w}_{n-1}, \beta\alpha\mathbf{I}) \\
p(v) &= \mathrm{InvGamma}(v|1,1) \\
p(\alpha) &= \mathrm{InvGamma}(\alpha|1,1) \\
p(\beta/(1-\beta)) &= \mathrm{InvGamma}(\beta/(1-\beta)|1,1)
\end{aligned}
$$

- Classical model corresponds to
  - Posterior mean for $\mathbf{w}_n$, for some magically selected α and v
  - Particular case of Kalman filter
  - Contrived assumptions needed to represent exponentially weighted RLS in this framework.
    - Hints that it might not make sense, see [KRLST].

Updating classical machine learning (IV/V)

- Classical algorithm: Principal Component Analysis
- Bayesian model

$$
\begin{aligned}
p(\mathbf{x}_n|\mathbf{y}_n, \mathbf{W}, v) &= \mathcal{N}(\mathbf{x}_n|\mathbf{W}^\top \mathbf{y}_n, v\mathbf{I}) \\
p(\mathbf{y}_n) &= \mathcal{N}(\mathbf{y}_n|\mathbf{0}, \mathbf{I}) \\
p([\mathbf{W}]_{d\text{-th col}}) &= \mathcal{N}(\mathbf{w}_d|\mathbf{0}, \mathrm{diag}([\alpha_1, \ldots, \alpha_D]) \\
p(v) &= \mathrm{InvGamma}(v|1, 1) \\
p(\alpha_d) &= \mathrm{InvGamma}(v|1, 1)
\end{aligned}
$$

- Classical model corresponds to
  - Maximum likelihood for $p(\mathbf{x}_n|\mathbf{W}, v)$ when $v \to 0$, with $\mathbf{W}$ product of orthogonal and ordered diagonal matrix
  - Restrictions on $\mathbf{W}$ make it unique, but don't change the model
  - New possibilities: What if we do max. lik. for $p(\mathbf{x}_n|\{y_n\}, v)$?

Updating classical machine learning (V/V)

- Classical algorithm: Support Vector Machines
- Bayesian model
  - See: Sollich, P. (2002). Bayesian Methods for Support Vector Machines: Evidence and Predictive Class Probabilities. *Machine Learning*, 46:21-52.
  - Ingenious approach, but not very natural (involves using three classes to solve a binary problem)

Contents

Some observations

According to the previous slides

- Most ML tools have a Bayesian model description
- Full description takes a few lines
- The type of inference (ML, MAP, point estimates, full Bayesian posterior) is independent of the model
  - Though the tractability of each does depend on the model

In the process of creating new ML tools

- What makes a new ML tool worth is:
  A novel model
- What we spend most time working on is:
  Making inference tractable on the new model

The idea

Probabilistic programming

- Define a language to describe Bayesian models ("programs")
- Create some "compiler" that understands those programs and generates inference engines for them

The new workflow (emphasis is on model design)

- Spend more time thinking about the model
- Program it (just a few lines!)
- Optional: Sample data from the model
- Feed data to the inference engine and assess the results
- Model wasn't that good, do it over

Programming paradigms (I/II)

A random assortment of them:

- Imperative (Matlab)
- Procedural (C)
- Object oriented (C++)
- Declarative (SQL)
- Functional (Ocaml, F#)
- Metaprogramming (LISP)
- Domain specific language (Spice)

Programming paradigms (II/II)

For probabilistic programming:

- A domain specific language may be simpler for the user, but
  - Doesn't integrate well with existing codebases
  - Doesn't interface well with DB access, plotting capabilities, etc.
- Functional languages with metaprogramming can be useful to write a "guest probabilistic program" within a "host programming language" (we'll see F# examples)
- This can be done with imperative style, but looks uglier (we'll see Python examples)

An incomplete list of probabilistic programming languages

- BUGS: Bayesian inference using Gibbs sampling
- HANSEI: Extends OCaml, discrete distributions only
- Hierarchical Bayesian Compiler (HBC): Large-scale models and non-parametric process priors
- PyMCMC: MCMC algorithms for Python classes
- Church: Extends Scheme to describe Bayesian models
- Infer.NET: Provides a probabilistic language within the .NET platform

See more on `http://probabilistic-programming.org`

Contents

The Java case

Three big operating systems

- Linux, OSX, Windows

...and one language to rule them all

- Sun Microsystems designed Java to run on the JVM
- ...and implemented the JVM to run on linux, mac, and windows
- platform independence, bliss for programmers

The .NET case

Microsoft reacted and created the JVM counterpart: The CLR
Which language was used to target the CLR?

- The .NET languages: VB.NET, C#, F#, IronPython...
- Different languages with a common set of libraries, it is easy to interface them

Which operating systems did the CLR run on?

- Microsoft released the specification and standardized it
- CLR-like implementations arose for linux/mac: Mono
- ...but the Windows version is always ahead, more complete, with additional tools, better tested, etc.

Microsoft tries to win in the cross-platform territory (sweet spot for developers) while still favoring its flagship product, Windows: Opposing objectives

Infer.NET's interoperability

- Infer.NET targets all .NET languages, with a focus on C#, F#, and IronPython
- It can be used on Mono (Mac/Linux) or the CLR (Windows) (better experience/less buggy on the latter)
- The .NET languages have a growing set of tools for scientific computing, but nowhere near Matlab yet
- IronPython cannot use Numpy/Scipy/Matplotlib natively
- A new tool called Sho provides an IronPython shell with Matlab-like capabilities (but Windows only)

Contents

Using Infer.NET

Infer.NET provides:

- A probabilistic modeling language embedded in other languages
  - F# allows a more natural embedding
- Compilation to three inference engines
  - EP: Approximation includes all non-zero probability points
  - VB: Approximation avoids zero probability points
  - MCMC: Gibbs sampling, slower

Let's browse Microsoft Research examples and create a new model
`http://research.microsoft.com/en-us/um/cambridge/projects/infernet/`

Two coins (F#)

```fsharp
////////////////////////////////////////////////
// Model
////////////////////////////////////////////////

open MicrosoftResearch.Infer.Fun.FSharp.Syntax

[<ReflectedDefinition>]
let coins () =
    let c1 = random (Bernoulli(0.5))
    let c2 = random (Bernoulli(0.5))
    let bothHeads = c1 && c2
    observe (bothHeads = false)
    c1, c2, bothHeads

////////////////////////////////////////////////
// Sampling
////////////////////////////////////////////////

// Sampling does not take observations into account.
printf "Sample: %0\n" (coins ())

////////////////////////////////////////////////
// Inference
////////////////////////////////////////////////

open MicrosoftResearch.Infer.Fun.FSharp.Inference

let (c1D,c2D,bothD) = inferFun3 <@ coins @> ()
printf "coinsD: \n%0\n%0\n%0\n" c1D c2D bothD
```

Two coins (IronPython)

```
#-----------------------------------------------------------------------
# Infer.NET IronPython example: Two Coins
#-----------------------------------------------------------------------

import InferNetWrapper
from InferNetWrapper import *

# two coins example
def two_coins() :
    print("\n\n----------------- Infer.NET Two Coins example -----------------
    # The model
    b = MicrosoftResearch.Infer.Distributions.Bernoulli(0.5)
    firstCoin = Variable.Bernoulli(0.5)
    secondCoin = Variable.Bernoulli(0.5)
    bothHeads = firstCoin & secondCoin

    # The inference
    ie = InferenceEngine()
    print "Probability both coins are heads:", ie.Infer(bothHeads)
    bothHeads.ObservedValue = False
    print "Probability distribution over firstCoin:", ie.Infer(firstCoin)
```

Learning a Gaussian (F#)

```
15  // The following are two equivalent models, which should return the same result.
16
17  [<ReflectedDefinition>]
18  let learningGaussianA data =
19      let mean = random (GaussianFromMeanAndPrecision(0.0, 100.0))
20      let prec = random (GammaFromShapeAndScale(1.0, 1.0))
21      for x in data do
22          let y = random (GaussianFromMeanAndPrecision(mean, prec))
23          observe (y = x)
24      mean, prec
25
26  [<ReflectedDefinition>]
27  let learningGaussianB data =
28      let mean = random (GaussianFromMeanAndPrecision(0.0, 100.0))
29      let prec = random (GammaFromShapeAndScale(1.0, 1.0))
30      observe (data = [| for x in data -> random (GaussianFromMeanAndPrecision(mean, prec)
31      mean, prec
32
33  ////////////////////////////////////////////////////
34  // Data
35  ////////////////////////////////////////////////////
36
37  let gaussianData = [|for x in 1..100 -> Rand.Normal(0.0, 1.0)|]
```

Learning a Gaussian (IronPython)

```
#----------------------------------------------------------------------------
# Infer.NET IronPython example: Learning a Gaussian
#----------------------------------------------------------------------------

import InferNetWrapper
from InferNetWrapper import *

def gaussian_ranges():
    print("\n\n------------------ Infer.NET Learning a Gaussian example ------------------\n");

    # The model
    len = Variable.New[int]()
    dataRange = Range(len)
    x = Variable.Array[float](dataRange)
    mean = Variable.GaussianFromMeanAndVariance(0, 100)
    precision = Variable.GammaFromShapeAndScale(1, 1)
    x[dataRange] = Variable.GaussianFromMeanAndPrecision(mean, precision).ForEach(dataRange)

    # The data
    data = System.Array.CreateInstance(float, 100)
    for i in range(0,100):
        data[i] = Rand.Normal(0, 1)

    # Binding the data
    len.ObservedValue = 100
    x.ObservedValue = data

    # The inference
    ie = InferenceEngine(VariationalMessagePassing())
    print "mean = ", ie.Infer(mean)
    print "prec = ", ie.Infer(precision)
```

Truncated Gaussian (F#)

```
16  // external data
17  let data = [0.0 .. 0.1 .. 1.0]
18
19  [<ReflectedDefinition>]
20  let truncatedGaussianA () =
21      let a = data
22      let result = [for x in a -> random (GaussianFromMeanAndPrecision(0.0, 1.0))
23      for x, y in List.zip a result do
24          observe (y > x)
25      result
26
27  // This produces a different result:
28  [<ReflectedDefinition>]
29  let truncatedGaussianB () =
30      let a = data
31      let y = random (GaussianFromMeanAndPrecision(0.0, 1.0))
32      let result = [for x in a -> y]
33      for x, y in List.zip a result do
34          observe (y > x)
35      result
```

Truncated Gaussian (IronPython)

```
#----------------------------------------------------------------------------
# Infer.NET IronPython example: Truncated Gaussian with different thresholds
#----------------------------------------------------------------------------

import InferNetWrapper
from InferNetWrapper import *

def truncated_gaussian():
    print("\n\n------------------ Infer.NET Truncated Gaussian example ------------------")
    # The model
    threshold = Variable.New[float]().Named("threshold")
    x = Variable.GaussianFromMeanAndVariance(0, 1).Named("x")
    Variable.ConstrainTrue(x > threshold)

    # The inference, looping over different thresholds
    ie = InferenceEngine()
    threshold.ObservedValue = -0.1
    for i in range (0, 11):
        threshold.ObservedValue = threshold.ObservedValue + 0.1
        print "Dist over x given thresh of ", threshold.ObservedValue, "=", ie.Infer(x)
```

Gaussian Mixture (F#)

```
17  [<ReflectedDefinition>]
18  let priors () =
19
20      let means = [|for i in 0 .. 1 -> random(VectorGaussianFromMeanAndPrecision(VectorFromArray [|0.0;
21      let precs = [|for i in 0 .. 1 -> random(WishartFromShapeAndScale(100.0, IdentityScaledBy(2,0.01))
22      let weights = random(Dirichlet([|1.0; 1.0|]))
23
24      (means, precs, weights)
25
26  [<ReflectedDefinition>]
27  let mix(means : Vector[], precs : PositiveDefiniteMatrix[], weights : Vector) =
28
29      let z = [|for i in 0 .. 300 -> random(Discrete(weights))|]
30      let data = [|for zi in z -> random(VectorGaussianFromMeanAndPrecision(means.[zi], precs.[zi]))|]
31      data, z
32
33  [<ReflectedDefinition>]
34  let mixtureModel (data : Vector[]) =
35      let (means, precs, weights) = priors ()
36      let mix, z = mix(means, precs, weights)
37      observe(data = mix)
38      (means, precs, weights, z)
```

Gaussian Mixture (IronPython)

```
34 def gaussian_mixture():
35     print("\n\n------------------ Infer.NET Mixture of Gaussians example ------------------\n")
36
37     # Define a range for the number of mixture components
38     k = Range(2)
39
40     # Mixture component means
41     means = Variable.Array[Vector](k).Named("means")
42     mm0 = Vector.FromArray(0.0,0.0)
43     mp0 = PositiveDefiniteMatrix.IdentityScaledBy(2,0.01)
44     means[k] = Variable.VectorGaussianFromMeanAndPrecision(mm0, mp0).ForEach(k)
45
46     # Mixture component precisions
47     precs = Variable.Array[PositiveDefiniteMatrix](k).Named("precs")
48     precs[k] = Variable.WishartFromShapeAndScale(100.0,
49         PositiveDefiniteMatrix.IdentityScaledBy(2,0.01)).ForEach(k)
50
51     # Mixture weights
52     weights = Variable.Dirichlet(k, System.Array[float]((1, 1))).Named("weights")
53
54     # Create a variable array which will hold the data
55     n = Range(300)
56     data = Variable.Array[Vector](n).Named("x")
57
58     # Initialise to break symmetry
59     length = n.SizeAsInt
60     # Create latent indicator variable for each data point
61     z = Variable.Array[int](n).Named("z")
62     # Call initialiser from wrapper
63     InitArray.init_var_arr(z, init_func, length)
64
65     #mixture of gaussians
66     with (Variable.ForEach(n)) :
67         z[n] = Variable.Discrete(weights)
68         with (Variable.Switch(z[n])) :
69           data[n] = Variable.VectorGaussianFromMeanAndPrecision(means[z[n]], precs[z[n]])
```

k-means (F#)

[Demo]

Conclusions

- Probabilistic programming is in its infancy
- We might produce alternative language definitions/implementations
- We can leverage it to test new models faster
- We can build custom models on the fly

References

- [BayesSVM] Sollich, P. (2002). Bayesian Methods for Support Vector Machines: Evidence and Predictive Class Probabilities. *Machine Learning*, 46:21-52.
- [ProbProg] The probabilistic programming wiki `http://probabilistic-programming.org/wiki/Home`
- [InferNET] T. Minka, J. Winn, J. Guiver, and D. Knowles Infer.NET 2.5, Microsoft Research Cambridge, 2012. `http://research.microsoft.com/infernet`
- [KRLST] M. Lázaro-Gredilla, S. Van Vaerenbergh, and I. Santamaría. "A Bayesian approach to tracking with kernel recursive least-squares", MLSP 2011.