Introduction to probabilistic programming

Xiao Jia

Shanghai Jiao Tong University

Outline

1. Motivation
2. ProbLog
3. Dimple
4. Infer.NET
5. Summary

Read the code below and answer what does it do

```
from scipy.optimize.optimize import fmin_bfgs
from numpy import exp, dot, zeros, sum, array

def sigmoid(x):
    return 1.0 / (1.0 + exp(-x))

def negative_lik(betas):
    l = 0
    for i in range(y.shape[0]):
        l += log(sigmoid(y[i] * dot(betas, x[i,:])))
    for k in range(1, x.shape[1]):
        l -= (alpha / 2.0) * betas[k]**2
    return -l

def train():
    dB_k = lambda B, k : (k > 0) * alpha * B[k] - sum([      \
           y[i] * x[i, k] * sigmoid(-y[i] * dot(B, x[i,:])) \
           for i in range(y.shape[0])])
    dB = lambda B : array([dB_k(B, k) for k in range(x.shape[1])])
    betas = fmin_bfgs(negative_lik, zeros(x.shape[1]), fprime=dB)
```

Read the code below and answer what does it do

(written in an imaginery probabilistic Python dialect)

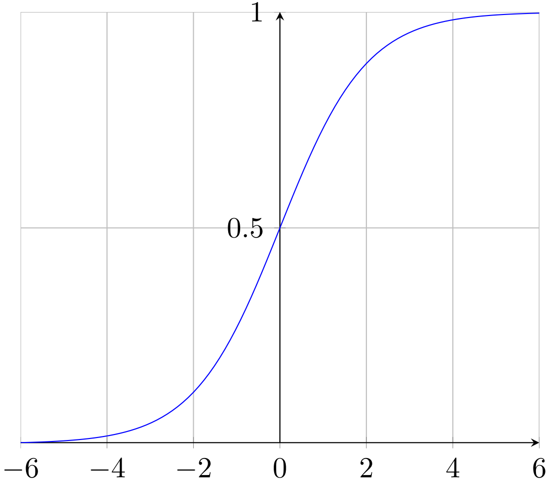
```
for k in range(M):
    w[k] ~ normal_distribution(mean=0, variance=0.0001)
for i in range(N):
    z = 0
    for k in range(M):
        z += w[k] * x[i, k]
    y[i] ~ bernoulli_distribution(p = 1 / (1 + exp(-z)))
```

- $v \sim D$ means random variable v is drawn from distribution D
- If X is a random variable with Bernoulli distribution, we have:

$$\Pr(X = 1) = 1 - \Pr(X = 0) = p.$$

Logistic regression

$$y_i = \lfloor \frac{1}{1 + e^{-\mathbf{w}^T \mathbf{x}_i}} + 0.5 \rfloor \in \{0, 1\} \text{ (see logistic curve below)}$$



Comparison

$$y_i = \lfloor \frac{1}{1 + e^{-\mathbf{w}^T \mathbf{x}_i}} + 0.5 \rfloor \in \{0, 1\}$$

Instead of fitting $\mathbf{w}$ from $\mathbf{x}_i$ and y_i, what if we have $\mathbf{x}_i$ and $\mathbf{w}$, and want to predict the values of y_i?

Comparison

$$y_i = \lfloor \frac{1}{1+e^{-\mathbf{w}^T \mathbf{x}_i}} + 0.5 \rfloor \in \{0, 1\}$$

Instead of fitting $\mathbf{w}$ from $\mathbf{x}_i$ and y_i, what if we have $\mathbf{x}_i$ and $\mathbf{w}$, and want to predict the values of y_i?

Code in Python/Java/...

- mix model and computation
- hardly reusable

Comparison

$$y_i = \lfloor \frac{1}{1 + e^{-\mathbf{w}^T \mathbf{x}_i}} + 0.5 \rfloor \in \{0, 1\}$$

Instead of fitting $\mathbf{w}$ from $\mathbf{x}_i$ and y_i, what if we have $\mathbf{x}_i$ and $\mathbf{w}$, and want to predict the values of y_i?

Code in Python/Java/...

- mix model and computation
- hardly reusable

Code in a probabilistic language

- define model only, let the interpreter compute
- model is reusable

Comparison

$$y_i = \lfloor \frac{1}{1 + e^{-\mathbf{w}^T \mathbf{x}_i}} + 0.5 \rfloor \in \{0, 1\}$$

Instead of fitting $\mathbf{w}$ from $\mathbf{x}_i$ and y_i, what if we have $\mathbf{x}_i$ and $\mathbf{w}$, and want to predict the values of y_i?

Code in Python/Java/...

- mix model and inference algorithm
- hardly reusable

Code in a probabilistic language

- define model only, let the inference engine compute
- model is reusable

What is probabilistic programming?

A probabilistic programming language is a high-level language that

- make it easy for a developer to define probability models
  - incorporate random events as primitives
  - abstract away the details of probabilistic inference
- "solve" these models automatically
  - enable a clean separation between modeling and inference
  - vastly reduce the time and effort associated with implementing new models and understanding data

Motivation of probabilistic programming

- separate specification and implementation
- design new models more easily
- improve inference algorithms without application domain details

Probabilistic languages can free developers from the complexities of high-performance probabilistic inference.

ProbLog: Tossing coins

```
%%% Probabilistic facts:
0.5::heads1.
0.6::heads2.

%%% Rules:
twoHeads :- heads1, heads2.

%%% Queries:
query(heads1).
query(heads2).
query(twoHeads).
```

Inference result:

```
{
  "heads1"   : 0.5,
  "heads2"   : 0.6,
  "twoHeads" : 0.3
}
```

Tossing coins

```
%%% Probabilistic facts:
0.6::heads(C) :- coin(C).

%%% Background information:
coin(c1).
coin(c2).
coin(c3).
coin(c4).

%%% Rules:
someHeads :- heads(_).

%%% Queries:
query(someHeads).
```

Inference result:

```
{
  "someHeads" : 0.9744
}
```

Bayesian networks

$p :: A \leftarrow B. \iff \Pr(A|B) = p$

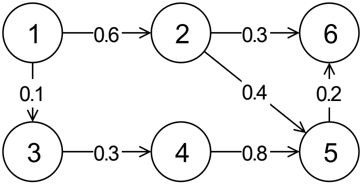
```
0.7::burglary.
0.2::earthquake.

0.9::alarm <- burglary, earthquake.
0.8::alarm <- burglary, \+earthquake.
0.1::alarm <- \+burglary, earthquake.

%%% Evidence:
evidence(alarm,true).

%%% Queries:
query(burglary).
query(earthquake).
```

Inference result:

```
{
  "burglary"   : 0.9896,
  "earthquake" : 0.2275
}
```

Probabilistic graphs

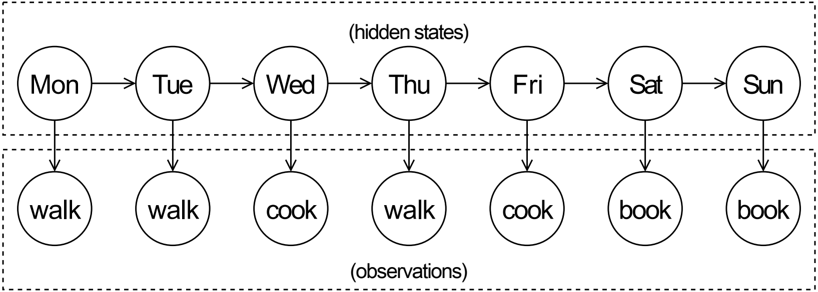
```
0.6::edge(1,2).
0.1::edge(1,3).
0.4::edge(2,5).
0.3::edge(2,6).
0.3::edge(3,4).
0.8::edge(4,5).
0.2::edge(5,6).

path(X,Y) :- edge(X,Y).
path(X,Y) :- edge(X,Z),
             Y \== Z,
             path(Z,Y).

query(path(1,5)).
query(path(1,6)).
```



Inference result:

```
{
  "path(1,5)" : 0.25824,
  "path(1,6)" : 0.2167296
}
```

Dimple: Hidden Markov model (HMM)



- Bob is only interested in three activities: (1) walking in the park, (2) reading a book, and (3) cooking. The choice of what to do is determined exclusively by the weather on a given day.
- Alice has no definite information about the weather where Bob lives, but she knows general trends. Based on what Bob tells her he did each day, Alice tries to guess the weather.

HMM parameters

```
states = ('sunny', 'rainy')

observations = ('walk', 'book', 'cook')

start_probability = {'sunny': 0.7, 'rainy': 0.3}

transition_probability = {
   'sunny' : {'sunny': 0.8, 'rainy': 0.2},
   'rainy' : {'sunny': 0.5, 'rainy': 0.5},
}

emission_probability = {
   'sunny' : {'walk': 0.7, 'book': 0.1, 'cook': 0.2},
   'rainy' : {'walk': 0.2, 'book': 0.4, 'cook': 0.4},
}
```

Prepare the factor graph

```
FactorGraph HMM = new FactorGraph();

DiscreteDomain domain
    = DiscreteDomain.create("sunny", "rainy");

Discrete mondayWeather    = new Discrete(domain);
Discrete tuesdayWeather   = new Discrete(domain);
Discrete wednesdayWeather = new Discrete(domain);
Discrete thursdayWeather  = new Discrete(domain);
Discrete fridayWeather    = new Discrete(domain);
Discrete saturdayWeather  = new Discrete(domain);
Discrete sundayWeather    = new Discrete(domain);
```

Adding transition factors

```
class TransitionFactor extends FactorFunction {
  double eval(String state1, String state2) {
    if (state1.equals("sunny"))
      return state2.equals("sunny") ? 0.8 : 0.2;
    else // rainy
      return 0.5;
  }}

TransitionFactor trans = new TransitionFactor();
HMM.addFactor(trans, mondayWeather,    tuesdayWeather);
HMM.addFactor(trans, tuesdayWeather,   wednesdayWeather);
HMM.addFactor(trans, wednesdayWeather, thursdayWeather);
HMM.addFactor(trans, thursdayWeather,  fridayWeather);
HMM.addFactor(trans, fridayWeather,    saturdayWeather);
HMM.addFactor(trans, saturdayWeather,  sundayWeather);
```

Adding observation factors

```
class ObservationFactor extends FactorFunction {
  double eval(String state, String observation) {
    if (state.equals("sunny")) {
      if (observation.equals("walk")) return 0.7;
      if (observation.equals("book")) return 0.1;
      if (observation.equals("cook")) return 0.2;
    } else { // rainy
      if (observation.equals("walk")) return 0.2;
      if (observation.equals("book")) return 0.4;
      if (observation.equals("cook")) return 0.4;
    }}}

ObservationFactor obs = new ObservationFactor();
HMM.addFactor(obs, mondayWeather,    "walk");
HMM.addFactor(obs, tuesdayWeather,   "walk");
HMM.addFactor(obs, wednesdayWeather, "cook");
HMM.addFactor(obs, thursdayWeather,  "walk");
```

Solving the factor graph

```
FactorGraph HMM = new FactorGraph();
Discrete mondayWeather = ...
...
TransitionFactor trans = ...
HMM.addFactor(trans, mondayWeather, tuesdayWeather);
...
ObservationFactor obs = ...
HMM.addFactor(obs, mondayWeather, "walk");
...
mondayWeather.setInput(0.7, 0.3); // initial state

HMM.getSolver().setNumIterations(20);
HMM.solve();

double[] belief = tuesdayWeather.getBelief();
println("sunny: " + belief[0]);
println("rainy: " + belief[1]);
```

Infer.NET: Two coins

```
Variable<bool> firstCoin = Variable.Bernoulli(0.5);
Variable<bool> secondCoin = Variable.Bernoulli(0.5);
Variable<bool> bothHeads = firstCoin & secondCoin;
```

Infer.NET: Two coins

```
Variable<bool> firstCoin = Variable.Bernoulli(0.5);
Variable<bool> secondCoin = Variable.Bernoulli(0.5);
Variable<bool> bothHeads = firstCoin & secondCoin;

InferenceEngine ie = new InferenceEngine();
Console.WriteLine("Probability both coins are heads: "
                  + ie.Infer(bothHeads));
```

```
Probability both coins are heads:  Bernoulli(0.25)
```

Infer.NET: Two coins

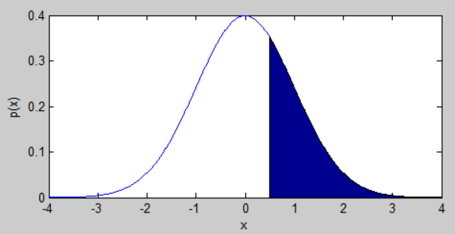
```
Variable<bool> firstCoin = Variable.Bernoulli(0.5);
Variable<bool> secondCoin = Variable.Bernoulli(0.5);
Variable<bool> bothHeads = firstCoin & secondCoin;

InferenceEngine ie = new InferenceEngine();
Console.WriteLine("Probability both coins are heads: "
                  + ie.Infer(bothHeads));
```

```
Probability both coins are heads:  Bernoulli(0.25)
```

```
bothHeads.ObservedValue = false;
Console.WriteLine("Probability distribution over firstCoin: "
                  + ie.Infer(firstCoin));
```

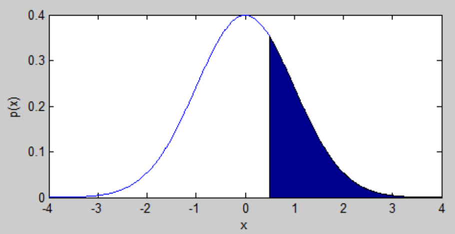
```
Probability distribution over firstCoin:  Bernoulli(0.3333)
```

Truncated Gaussian




```
Variable<double> x
    = Variable.GaussianFromMeanAndVariance(0, 1).Named("x");
Variable.ConstrainTrue(x > 0.5);
```

Truncated Gaussian




```
Variable<double> x
    = Variable.GaussianFromMeanAndVariance(0, 1).Named("x");
Variable.ConstrainTrue(x > 0.5);
```

If we use expectation propagation, the result will be the Gaussian distribution closest to the shaded area above

```
InferenceEngine engine = new InferenceEngine();
engine.Algorithm = new ExpectationPropagation();
Console.WriteLine("Dist over x=" + engine.Infer(x));
```

```
Dist over x=Gaussian(1.141, 0.2685)
```

There are many probabilistic programming languages

BLOG, BUGS, Church, Dimple, Factorie, Figaro, Hansei, PRISM, Infer.NET, ProbLob, PyMC, Stan, ...

- support different kinds of models
  - generative model
  - discriminative model
- include different inference engines
  - rejection sampling
  - Metropolis-Hastings algorithm, Gibbs sampling

Generative model versus discriminative model

Suppose we have the following training data in the form (x, y).

$$(1,0),(1,0),(2,0),(2,1)$$

Joint probability $\Pr(x, y)$ is

	$y=0$	$y=1$
$x=1$	$1/2$	0
$x=2$	$1/4$	$1/4$

Conditional probability $\Pr(y|x)$ is

	$y=0$	$y=1$
$x=1$	1	0
$x=2$	$1/2$	$1/2$

Generative model versus discriminative model

In *discriminative models*, to predict the label y from x, we must evaluate

$$f(x) = \arg\max_y \Pr(y|x)$$

Now using Bayes' rule, replace $\Pr(y|x)$ by $\dfrac{\Pr(x|y)\Pr(y)}{\Pr(x)}$.
$\Pr(x)$ is the same for every y, so we are left with

$$\begin{aligned} f(x) &= \arg\max_y \Pr(x|y)\Pr(y) \\ &= \arg\max_y \Pr(x, y) \end{aligned}$$

which is the equation we use in *generative models*.

Questions?

References:

- ProbLog: http://dtai.cs.kuleuven.be/problog/
- Dimple: http://dimple.probprog.org/
- Infer.NET: http://research.microsoft.com/infernet/