Algorithms for Graphical Models (AGM)

Rejection and importance sampling

$Date: 2007/11/19 11:31:20 $

AGM-12

In this lecture

- Sampling
  - Forward
  - Rejection
  - Importance

What's wrong with computing probabilities exactly

- Recall that probability propagation in a join tree is exponential in the size of the biggest clique.
- Sometimes this is just too long to wait.
- We can use *sampling* algorithms to compute probabilities (and more generally *expectations*) approximately.

Approximating expectations by sampling

Approximate

$$\mathbf{E}_P(f) = \sum_{\mathbf{x}} P(\mathbf{x}) f(\mathbf{x})$$

by

$$\widehat{\mathbf{E}}_P(f) = \frac{1}{M} \sum_{m=1}^{M} f(\mathbf{x}_m)$$

where the $\mathbf{x}_m$ are joint instantiations *sampled* from P.

If f is the indicator function for an event, then $\mathbf{E}_P(f) = P(\text{event})$.

Sampling: what it is

- Consider a random variable $Lecture$ with 3 values $good, bad, soso$ with probabilities $0.7, 0.1$ and 0.2 respectively.

- A sampler for this distribution is a (randomised) algorithm which outputs $good$ with probability 0.7, bad with probability 0.1 and $soso$ with probability 0.2.

- In we draw a large number of samples from this sampler, then with high probability the relative frequency of $good$s will be close to 0.7

Sampling: how to do it

- Suppose we have a ‘random number’ generator which outputs numbers in $[0, 1)$ according to a uniform distribution over that interval . . .

- . . . then we can sample a value for $Lecture$ by seeing which of these intervals the random number falls in: $[0, 0.7), [0.7, 0.8)$ or $[0.8, 1)$

- In Python the `random` function in the module `random` does this.

Randomness and pseudo-randomness

```
>> import random
>>> random.seed(0.2)
>>> random.random()
0.39069412806005199
>>> random.random()
0.22251136096622171
>>> random.seed(0.2)
>>> random.random()
0.39069412806005199 #hmm, same 'random' number as earlier
```

If you don't supply a random seed, Python uses the time of day. Random seeds allow you get the same sequence of random(!) numbers many times over.

Sampling: why to do it

- Suppose we wished to work out the mean (average) number of throws needed to finish a "Snakes and Ladders" game (with a fair die)

- We can, in principle, compute this number—but this would be an unpleasant (and long) task

- Instead we can write a sampler to simulate playing the game (say 500 times); add up the total number of throws for all 500 games; divide by 500, and use this as an estimate for the mean.

Sampling: for approximate probabilistic inference

- If we can sample from a joint probability distribution then we can use the sample to derive estimates for probabilities which might be too costly to compute exactly.

- Suppose we have a distribution P over variables X_1, X_2, X_3 and we sample n joint instantiations.

- Let $n(X_1 = x_1, X_3 = x_3)$ be the number of times a joint instantiation with $X_1 = x_1, X_3 = x_3$ occurs.

- Then $\widehat{P}(X_1 = x_1, X_3 = x_3) = (X_1 = x_1, X_3 = x_3)/n$ is an estimate for $P(X_1 = x_1, X_3 = x_3)$

Sampling: why it (probably, approximately) works

- Consider tossing a coin a large number of times, where the probability of heads on any toss is p.

- Let S_n be the number of heads that come up after n tosses. Think of S_n as the number of successes.

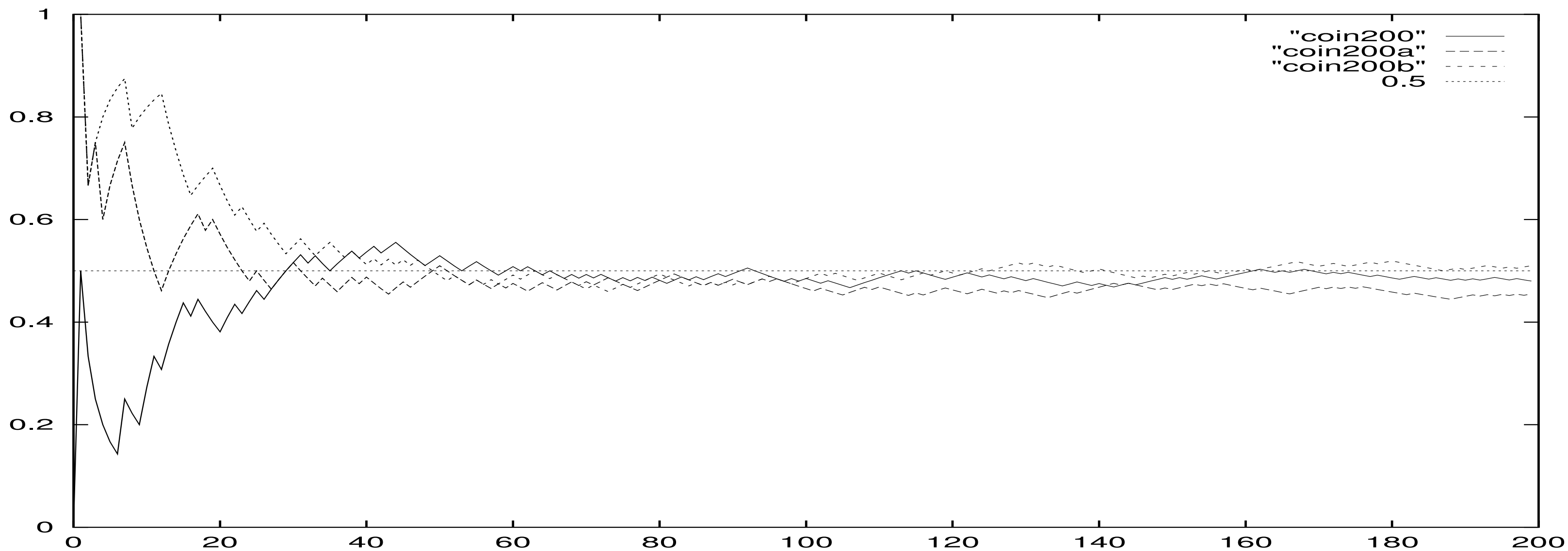
- The law of large numbers says that the probability that S_n/n differs much from p becomes smaller and smaller as n gets bigger.

- In brief, if $\epsilon > 0$, then as $n \to \infty$ $P\left(\left|\frac{S_n}{n} - p\right| < \epsilon\right) \to 1$

Simulating coin tosses

Here's a graph of S_n/n against n for three different sequences of simulated coin tosses, each of length 200.

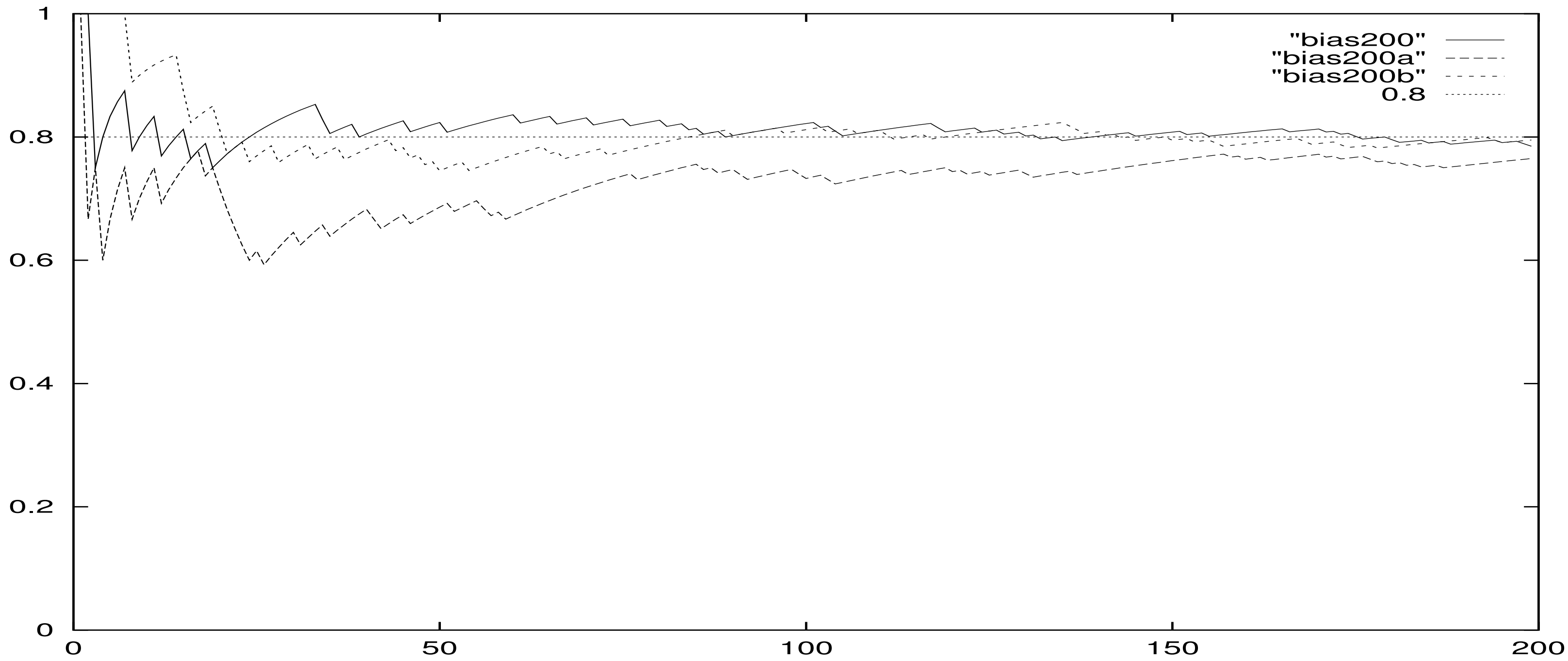
Remember that $p = 0.5$



AGM-12

A biassed coin

I changed p to 0.8 and repeated:



AGM-12

Sampling from a 2-variable distribution

- Suppose we have the BN $P(A, B) = P(A)P(B|A)$ where both variables are binary.

- To sample from $P(A, B)$ we first sample from $P(A)$. Suppose we get $A = 0$.

- In this case, we then sample from $P(B|A = 0)$.

- If we had sampled $A = 1$ then in the second step we sample from $P(B|A = 1)$.

Forward sampling in Bayesian nets

- In a BN (1) we can order parents before children (topological order) and (2) we have CPTs available.

- *If no variables are instantiated* this allows a simple algorithm: *forward sampling*.

- Just sample variables in some fixed topological order, using the previously sampled values of the parents to select the correct distribution to sample from.

Forward sampling in gPy

```
def sample(self):
 inst = self._instskel[:]   # make a list of Nones
 for i in self._indextuple: # variables in a top. order

  # yank out the parent instantiation ...
  parent_inst = tuple([inst[j] for j in self._indices[i]])

  # yank out the cond. distn. for variable i ...
  i_dist = self._dists[i][parent_inst]

  inst[i] = i_dist.sample() # sample a value for variable i
 return inst
```

Using samples

Cue `forward_sample_demo` from `gPy.Examples`

- If we keep the entire sample we don't need to decide ahead of time what we want to estimate.
- For example, we can compute conditional probabilities by throwing away instantiations which don't meet the condition.
- Alternatively, we can *reject* such instantiations as soon as we know they don't meet the condition.

Cue `rejection_sample_demo` from `gPy.Examples`

The problem with rejection sampling

- Rejection sampling is inefficient since many samples are thrown away.

- This is particularly so if the ‘evidence’ (the condition) is improbable.

- If the evidence is ‘near the bottom’ of the BN lots of work is done before a sample is rejected.

Alternatives to rejection sampling

1. Construct a new BN which represents the conditional distribution and just use forward sampling (particularly easy if instantiated variables have no parents).

2. Importance sampling

3. Gibbs sampling

(Unnormalised) Importance sampling

Basic idea: If a distribution P is difficult to sample from, sample from a *different* distribution Q and *weight* each sample $\mathbf{x}_m$ by $w(\mathbf{x}_m) = P(\mathbf{x}_m)/Q(\mathbf{x}_m)$ to compensate.

Estimate $\mathbf{E}_P(f)$ by $\frac{1}{M}\sum_{m=1}^{M} f(\mathbf{x}_m)w(\mathbf{x}_m)$ where the $\mathbf{x}_m$ are sampled from Q.

It works because $\mathbf{E}_P(f) = \mathbf{E}_Q\left(f\frac{P}{Q}\right)$

Normalised importance sampling

- Unnormalised importance sampling assumes that P, the distribution from which we wish to sample, is known.

- Often we just have $P'(\mathbf{x}) = \alpha P(\mathbf{x})$ where α is some unknown normalisation constant.

- So can only compute weights $w(\mathbf{x}_m) = P'(\mathbf{x}_m)/Q(\mathbf{x}_m)$

Estimate $\mathbf{E}_P(f)$ by

$$\frac{\sum_{m=1}^{M} f(\mathbf{x}_m) w(\mathbf{x}_m)}{\sum_{m=1}^{M} w(\mathbf{x}_m)}$$

where the $\mathbf{x}_m$ are sampled from Q.

Likelihood weighting

- Alter forward sampling so that instead of sampling values for instantiated variables (and just hoping the sampled value matches the given one) we *set* the value to the given value.

- The less likely the sampled value would be the set value, the more we have ‘cheated’.

- Suppose X is set to x, then reduce the *weight* of the sample by $P(X = x|Pa(X) = pax)$ where pax is the already sampled value of the parents of X.

Likelihood weighting in gPy

```
def importance_sample(self):
  inst = self._instskel[:]
  weight = 1
  for i, i_evidence in enumerate(self._cond_index):
    parent_inst = tuple([inst[j] for j in self._indices[i]])
    if i_evidence is None:
      inst[i] = self._dists[i][parent_inst].sample()
    else:
      inst[i] = i_evidence
      weight *= self._cpts[i][parent_inst][i_evidence]
  return inst, weight
```

Likelihood weighting is importance sampling

- The *proposal distribution* Q is defined by this BN:

  1. If $X = x$ in the evidence, make X an orphan and set $Q(X = x) = 1$ in its CPT.

  2. Leave non-evidence variables untouched.

- The sampling in the likelihood weighting sampler is then just forward sampling from this *mutilated* BN.