8. APPROXIMATE INFERENCE: STOCHASTIC SAMPLING AND MESSAGE PASSING

RENATO LUI GEH
NUSP: 8536030

Abstract. This document describes how approximate inference through stochastic sampling and sum-product message passing were implemented.

1. Project decisions

It was decided that the input bayesian network $\mathcal{N}$ must have its potentials ordered such that, for each potential ϕ in $\mathcal{N}$, the node variable must come last. That is, consider a potential $\phi(X_i, Pa(X_i))$, the input file must be ordered such that, in order of appearance, $Pa(X_i)$ must come first and X_i last. Consequently, the probability distribution must also obey such ordering.

This was decided as a consequence of factor implementation. Since the potential table is stored recursively $\phi[X_1]\ldots[X_n]$, finding the summed-out distribution $\sum_{\sim\{X_n\}}\phi$, where $\sum_{\sim\{X\}}\phi \equiv \sum_{\mathbf{X}\setminus\{X\}}\phi$, is easier when X_n is the recursive tail call result (i.e. the last table).

For more information, see `sampling.lua` and `parser.lua`.

In this document, each section is associated with a Lua module. When we refer to a function `func`, the reader must implicitly prepend the section module to its name (e.g. `Module.func`) unless the function already contains a prefix. Local functions are always hidden inside the section module, thus we do not need to prepend any modules to them.

2. Stochastic sampling

The associated module is `Sampling`.

Both logical and likelihood-weighting sampling were implemented and can be found in the `sampling.lua` source code file. Both sampling methods use functions `get_distribution` and `generate`. We only implement sampling for bayesian networks.

Function `get_distribution` takes a probability distribution `Qr` and evidence `E` and returns a probability distribution `Pr` such that `Pr` is the result of the reduction of `Qr` given `E`. That is, for every instantiation ν that is not consistent with evidence

`E`, delete ν. Return the remaining instantiations. Since every potential is a conditional distribution with the evidence being parents, we are actually selecting the instantiations that agree with the parents' instantiations. `E` must have values for every variable node's parents.

The function `generate` takes a probability distribution `Pr` and samples an instantiation according to $Val(X)$, where $\{X\} = Sc(\texttt{Pr})$ and $Val(\cdot)$ is the set of possible values of a variable. This is done with the Lua built-in pseudo-random number generator. It is guaranteed to generate a uniform distribution of double-precision numbers in Linux.

2.1. *Logical sampling*

Logical sampling was implemented in the form of the function `logical`. It takes as arguments sets V, P, E and an integer number m.

Let $\mathcal{N}$ be the bayesian network we wish to sample. Then V is the set of nodes in $\mathcal{N}$ ordered in a topological order such that all root nodes come first, and for every non-root node $V[j] = v_i \in V$, the indeces of $Pa(v_i)$ are all less than j. That is, every parent of v_i must be "behind" v_i. The set P is the ordered collection of potentials. For every variable X_i, the associated potential in $\mathcal{N}$ is P_i. The evidence set E is ordered such that every E_i can have two possible values. Either E_i has a value of variable X_i or $E_i =$`nil` if there is no evidence of that variable. The integer m is the number of samples we wish to generate.

Function `logical` first generates instantiations for the root nodes. Since root nodes have no dependencies, we simply generate possible instantiations with functions `generate` and `get_distribution`. When we encounter a non-root node, we must first build the evidence set according to previous instantiations. This evidence set is represented by table s. Once we have a set of possible instantiations, we call `generate` and `get_distribution`, this time with an evidence $s[j]$.

Once we have build m samples, we compute, through Monte Carlo, the probability in which the samples are consistent with the initial evidence E. This result is an approximation to the marginal probability $\Pr(E)$.

2.2. *Likelihood-weighting sampling*

The function `likelihood` implements likelihood-weighting sampling. It takes as arguments sets V, P, E and an integer number m. These arguments are ordered the same way as in logical sampling and represent the same sets as the previous function.

First we create a likelihood-weighting network $\mathcal{L}$ that is initially built just like the bayesian network $\mathcal{N}$. However, for every variable node V_i in which E_i is not `nil`, we replace the associated potential P_i with a potential where $\phi(\mathbf{x})$ is 1 if $\mathbf{x}$ agrees with E_i and 0 otherwise.

After we create $\mathcal{L}$, we generate samples similarly to `logical`, but with the likelihood-weighting network in place of $\mathcal{N}$ as the network model. Then, once we have a set of possible instantiations, we compute the probability of evidence $\Pr(E)$ and marginal $\Pr(X, E)$ by multiplying weights that are consistent with the evidence. Once we have the weights, we sum on each probability and later normalize the distribution.

The resulting values are the probability of evidence and the marginal probability of each variable.

3. Message passing

The associated module is `Message`.

We implement message passing through factor graphs according to *Kschischang et al* [KFL01].

Function `factor_graph` creates a Factor Graph $\mathcal{F} = (V, E)$. It takes sets P and S as arguments. P is the set of potentials and S is the scope of the network. Factor graph $\mathcal{F}$ is structured such that a function node f_i represents a potential P_i and a variable node v_j is a variable S_j. There exists an edge $f_i \leftrightarrow v_j$ iff $v_j \in Sc(f_i)$. In `factor_graph`, we first construct the nodes in the adjacency list `adj_list`. This list is structured such that each pair `(key, value)` in the adjacency list is an equivalent pair $(n_i, Ne(n_i))$, where n_i is an arbitrary node and Ne is the set of neighbours of a variable. Once we have created the graph's nodes, we then construct each edge following the factor graph edge rules. Each edge contains two elements: a message passing function λ_{ij} and the message ψ_{ij} to be passed from node i to j.

There are two possible message passing functions. Let f be a function node and x be a variable node. Then λ_{fx} is defined as:

```
1  λ_fx(x) = function (x)
2    return ∑_{∼{x}} ( f(x) ∏_{y∈Ne(x)\{x}} λ_yf(y) )
3  end
```

And λ_{xf} is defined as:

```
1  λ_xf(x) = function (x)
2    return ∏_{h∈Ne(x)\{f}} λ_hx(x)
3  end
```

In `factor_graph`, λ_{fx} is represented as `adj_list[`f`][`x`][1]`. The return value of λ_{xy}, ψ_{xy}, is stored in `adj_list[`x`][`y`][2]`.

Computing the marginal of each variable is done through function `sp_factor`, which takes a factor graph G and scope S as arguments. It then computes each marginal $\Pr(X)$ for each variable $X \in S$ by computing all messages that are passed on node X in graph G.

References

[KFL01] Frank R. Kschischang, Brendan J. Frey, and Hans-Andrea Loeliger. "Factor Graphs and the Sum-Product Algorithm". In: *IEEE Transactions on Information Theory, Vol. 47, No. 2* (2001).