Algorithms for Graphical Models (AGM)

Probability propagation in join forests

$Date: 2008/10/16 10:37:10 $

AGM-10

In this lecture

- Probability propagation in join forests
- Constructing decomposable hypergraphs from non-decomposable ones

What's wrong with variable elimination

- Typically, we want marginal distributions for *all* uninstantiated variables.

- We could run variable elimination from scratch for each of them.

- But, whatever the elimination ordering, we would end up repeating a lot of computational work.

- There has to be a better way . . .

Running variable elimination in a join forest

- Recall the cluster tree representation of the execution of the variable elimination algorithm. Cue `cluster_tree` from `gPy.Examples`

- A join forest is essentially a cluster tree with redundancy removed, so, unsurprisingly, we can run variable elimination 'in a join forest'.

- But what does that mean precisely?

Graham's algorithm

- Recall Graham's algorithm: it is 'structural' variable elimination and is used to build a join tree.

- Consider running variable elimination with an elimination ordering which can be used to run Graham's algorithm (ie is a zero fill-in on the associated graph).

- Assume further that the ordering is such that, when we eliminate an 'isolated' variable (ie it's in only one clique) we also eliminate all other isolated variables in that clique, if any.

- It's easy to see that this still gives us a zero fill-in.

Initialisation: situating factors

- Since our distribution is decomposable, there is a one-one mapping between the initial factors and the nodes (ie cliques) in the join forest.

- It is useful to think of each factor as being located at its corresponding clique.

- gPy specifics: all factored distributions have a dictionary `_factors` mapping hyperedges to factors. If the distribution is decomposable these hyperedges are also identified with the join forest nodes.

Absorbing messages

- Once all the isolated vertices in a clique have been summed out a new factor is created (unless the clique contained only isolated vertices).

- Call such factors *messages*.

- Rather than adding messages as a new factor in the factored distribution, we will immediately multiply them into an existing factor.

Absorbing messages (ctd)

- Decomposability guarantees that there is always a 'receiving' factor which includes all the variables of the message.

- This corresponds to the 'deleting a redundant hyperedge' step of Graham's algorithm.

- Since the join forest can be built from running Graham's algorithm, the 'sending' and 'receiving' cliques will be directly connected in the join forest.

Retaining the join forest

- Rather than actually delete a factor once it has produced a message, we can just treat it 'as if it were deleted'.

- So to run variable elimination in a join *tree* each clique except the last 'sends a message' to its neighbour.

- The last clique is the 'root' clique: once it has collected all its messages it contains the marginal distribution for the variables it contains.

- (The generalisation to join *forests* is not very interesting.)

Towards probability propagation

- Suppose we have run variable elimination in a join tree as just described.

- Consider a clique C_2 neighbouring the root clique C_1. C_2 has not received a message from the root C_1, but has received all the other messages it would have received had it (rather than C_1) been the root.

- If C_1 *were* to send it a message now it would be the wrong message since C_1 'includes' (by way of multiplication) the message sent to it from C_2, which it would not have received were C_2 the root.

Storing messages

- If we *store* the message $m_{C_2 \to C_1}$ that C_2 sends to C_1, then C_1 can send the right message back to C_2 even after receiving $m_{C_2 \to C_1}$

- It just *divides* the 'normal' message (produced by marginalisation) by $m_{C_2 \to C_1}$ thus cancelling out the effect of previously receiving it.

- Then C_2 will end up with exactly the same factor it would have had had it been the original root.

Probability propagation

- This argument can then be extended to apply to *all* cliques in the join forest

- Storing messages is the key to the *probability propagation* algorithm in a join forest.

- It's best to imagine the messages being stored 'in' the lines connecting neighbouring cliques in the join forest.

Introducing separators

- Having (hopefully) convinced you that join forests are the right structure for 'parallel' variable elimination, let's introduce a more elegant characterisation of what's going on.

- A *separator* between two neighbouring cliques in a join tree is just the intersection of the variables in the 2 cliques.

- The variables of a message constitute a separator.

- We can associate factors with separators, just as we associate factors with the cliques

A new factorisation

Let p be a decomposable distribution, so

$$p = \prod_{C \in \mathcal{C}} f_C$$

where the $\mathcal{C}$ are cliques, the vertices of a join forest.

Let $\mathcal{S}$ be the separators of a join forest, and set 1_S to be a table of ones for each $S \in \mathcal{S}$. We can then write:

$$p = \frac{\prod_{C \in \mathcal{C}} f_C}{\prod_{S \in \mathcal{S}} 1_S}$$

Sending messages

- Suppose C_1 sends a message $m_{C_1 \to C_2}$ to C_2 and their separator is S.

- To effect the probability propagation algorithm, the factor for C_2 gets multiplied by $m_{C_1 \to C_2}/f_S$ . . .

- . . . and so does the factor for S.

- So *sending a message does not alter the distribution*.

- We have a loop invariant.

It's really very simple

```
def send_message(self,frm,to):
  """Send a message from a clique to a neighbouring clique"""
  message = self._factors[frm].sumout(frm - to)
  edge = frozenset([frm,to])
  self._factors[to] *= (message/self._separators[edge])
  self._separators[edge] = message   # simple optimisation
```

`self._separators[edge]` is a table of ones initially.

Message scheduling

- Once each clique has received all its messages it will contain the marginal distribution for its variables.

- We just need to ensure that all messages in both directions are passed.

- One option is (1) to choose an arbitrary root, (2) send all messages towards this root (the *upward pass*) and then (3) send all messages in the direction away from the root (the *downward pass*).

Perfect sequences

- A *perfect sequence* is an ordering of the cliques in a join tree where the first clique is a root, and such that if C_i is nearer this root than C_j then $C_i < C_j$.

- Note: This is not the actual definition. Can be extended to join forests.

- We can use a perfect sequence to schedule messages.

Join forest calibration

```
def calibrate(self):
  """Alter a JFM so that the factors associated with both cliques an
  separators are the appropriate marginal distributions
  """
  perfect_sequence = self._hypergraph.perfect_sequence()
  self.send_messages(perfect_sequence[:])
  perfect_sequence.reverse()
  self.send_messages(perfect_sequence)
```

Why 'calibration'?

The hypergraph $\{\{E,L,T\},\{A,T\},\{E,X\},\{B,E,L\},\{B,L,S\},\{B,D,E\}\}$ is decomposable (honest!)

Before calibration:

$$P(A,B,D,E,L,S,T,X) = \frac{f_{ELT} \times f_{AT} \times f_{EX} \times f_{BEL} \times f_{BLS} \times f_{BDE}}{1_T \times 1_E \times 1_{EL} \times 1_{BL} \times 1_{BE}}$$

After calibration:

$$P(A,B,D,E,L,S,T,X) = \frac{P(E,L,T)P(A,T)P(E,X)P(B,E,L)P(B,L,S)P(B,D,E)}{P(T)P(E)P(E,L)P(B,L)P(B,E)}$$

Help! my distribution does not have a decomposable hypergraph

- Then: make it so!
- Find a (hopefully) small fill-in for the interaction graph.
- Or do something equivalent directly on the hypergraph.
- Map each original factor to a clique which includes its variables (alway possible).
- The factor for each clique in the new hypergraph is a product (possibly empty) of those factors mapped to it.