Lifted Inference Tutorial Notes
UC Irvine
November 4-5, 2014

Rodrigo de Salvo Braz

Outline

- First-order representation
- Splitting Parfactors
- Multiplying Parfactors
- Counting Elimination
- Partial Inversion

Useful materials

- From a Variable Elimination perspective:
  thesis by Nima Taghipour (U Leuven)
  Very nicely presented

- From a And/Or search perspective:
  Probabilistic Theorem Proving,
  by Vibhav Gogate

Representation

- A First-Order Probabilistic Model (FOPM) is a set of parfactors
- A parfactor is a "universally quantified factor": for all Lvars : Constraint . phi(Atoms)
- An Atom is a relational random variable: happy(X), friends(X,Y), sunny(), etc.
- Lvars are "logical variables" (indices), discretely valued from a (large) set (the X and Y above)
- Constraint is a boolean formula on equalities on Lvars (for example: "X != bob and Y != X")

Representation Examples

- A parfactor is a "universally quantified factor":
  for all LVars : Constraint . phi(Atoms)

  Examples:

  for all X : X != bob . if happy(X) then 0.7 else 0.3

  for all X,Y : true . if friends(X,Y) and smoker(X)
  then if smoker(Y) then 0.8 else 0.2
  else if smoker(Y) then 0.4 else 0.6

Semantics

- Given a set of parfactors and domains for the logical variables, a FOPM defines a joint distribution identical to the one obtained by grounding, or instantiating, or yet propositionalizing, all parfactors into regular factors

Semantics Examples

- for all X:X != bob . if happy(X) then 0.7 else 0.3
  which X in {ann, bob, carl, dave, … }
  stands for

  if happy(ann) then 0.7 else 0.3
  (no instantiation for X = bob)
  if happy(carl) then 0.7 else 0.3
  if happy(dave) then 0.7 else 0.3

  and so on.

Joint probability

- A model defines a joint probability defined by the normalization of:
- prod_parfactor g

  prod_Lvars in g : Constraint_g

  phi_g(Atoms_g)

  Example:

  prod_{X:X != bob} phi1(happy(X))

  *

  prod_{X,Y: X!=Y}

  phi2(happy(X),married(X,Y),happy(X))

Marginalization

- In marginalization, we want to compute:

  sum_{RV \ query} parfactors product

  but the representation of random variables is often intensional, that is, by constraint:

  sum_{ {happy(X)}_{X: X!=ann} }
  sum_{ {married(X,Y)}_{X,Y : X != Y} }
  … product of parfactors

Note about notation

- Especially in early papers, there was confusion in notation, with
- sum_{ p(X) }

  being used when the meaning was

  sum_{ { p(X) }_{X : true} }

  mostly because the latter is not standard

Marginalization

- sum_{RV \ query} parfactors product
  is of the form:

  sum_{ Atom1 }_LVars1 : Constraint1
  sum_{ Atom2 }_LVars2 : Constraint2
  …
  prod_{Lvars_g1: Constraint_g1} phi_g1(Atoms_g1)
  prod_{Lvars_g2: Constraint_g2} phi_g2(Atoms_g2)
  prod_{Lvars_g3: Constraint_g3} phi_g3(Atoms_g3)
  …

Factoring out

- sum_{ Atom1 }_LVars1 : Constraint1
  …
  <factors *not sharing* RV with
  { Atom2 }_LVars2 : Constraint2>
  …
  sum_{ Atom2 }_LVars2 : Constraint2
  <factors *sharing* RV with Atom2: Constraint2>

- This splitting of *factors* is not a splitting of *parfactors*, because some instantiations of a parfactor may share RVs with Atom2:Constraint2 while others do not

Splitting a Parfactor

- We want to split a parfactor
  for all Lvars_g : Constraint_g . phi(Atom1,…,Atom_m)

  into two sets of factors (two parfactors), one parfactor sharing RVs with Atom:Constraint, and the other not sharing them.
  Example:
  for X,Y : X != Y . phi(happy(X),married(X,Y),happy(Y))
  sharing RVs with { happy(Z) }_{Z : Z != ann}

Splitting a Parfactor

- for X,Y : X != Y . phi(happy(X),married(X,Y),happy(Y)) sharing RVs with happy(Z) : Z != ann

  What are the instantiations such that happy(X) or happy(Y) is in { { happy(Z) }_{Z : Z != ann} ?
  These are the ones with X, Y such that
  ∃ Z : Z != ann and (happy(X) unifies with happy(Z) or
  happy(Y) unifies with happy(Z))
  which is
  ∃ Z : Z != ann and (X = Z or Y = Z)

  Let's call that IC (intersection constraint)

A fundamental tool: Equality Logic

- Often shoved aside as a detail in papers, but important and not very simple
- Model Counting:
  How many assignments to Lvars in equality boolean formula?

  #X,Y : X != bob and Y != X
  =
  (| Dom(X) | - 1) * (| Dom(Y) | - 1)

A fundamental tool: Equality Logic

- We also need *quantifier elimination:*

  $\exists$ X . Y = a and X != Y and X != a

  $\Leftrightarrow$

  Y = a

- This can be see as a projection (from relational algebra or databases) of tuples on X,Y on tuples on Y alone

Splitting a Parfactor

- for X,Y : X != Y . phi(happy(X), married(X,Y), happy(Y))
  sharing RVs with happy(Z) : Z != ann

  IC (intersection constraint) on X, Y is:
  ∃ Z : Z != ann and (happy(X) unifies with happy(Z) or
  happy(Y) unifies with happy(Z))
  =
  ∃ Z : Z != ann and (X = Z or Y = Z)
  =
  X != ann or Y != ann

  Then we obtain two parfactors:
  for X,Y : X != Y and IC . phi(…)

  for X,Y : X != Y and not IC . phi(…)

Splitting a Parfactor

- In general, for all Lvars_g : Constraint_g . phi(Atom1, …, Atom_m)

  sharing RVs with { Atom }_{ Lvars : Constraint }

  is the pair of parfactors:
  for all Lvars_g : (Constraint_g and IC) . phi(Atom1, …, Atom_m)
  for all LVars_g: (Constraint_g and not IC) . phi(Atom1, …, Atom_m)
  (the second one being called *residual*)

  where IC is
  ∃ Lvars .
  Constraint
  and (Atom unifies with Atom_1 or … or Atom unifies with Atom_m)

  (A unifies with B) iff A and B share predicate, and arguments are equal

Up to now

- So, up to now, we have learned how to represent the problem and how to factor parfactors out of a summation.
- Now we need to learn how to actually perform the summation, which is over an intensionally defined (that is, by constraint) set of random variables:

sum_{ Atom }_Lvars:Constraint
    prod_Lvars_g1:Constraint_g1 phi_g1(Atoms_g1)
    …
    prod_Lvars_gn:Constraint_gn phi_gn(Atoms_gn)

Algorithm Overview

1. Pick a set of random variables
2. Split parfactors involving those
3. Multiply them into one parfactor
4. Apply Partial Inversion and Counting variable conversion as much as possible
5. If you get summations without products inside, perform them, otherwise ground one of the variables in the product and go to 4.

Multiplying Parfactors

- Example:
  How can I replace the product of two parfactors potentials:
  (prod_X phi_1 (p(X))) . (prod_Y phi_2(q(Y)))
  by the potential of a single parfactor (a necessary condition for later operations)?

Multiplying Parfactors

(prod_X phi_1 (p(X))) . (prod_Y phi_2(q(Y)))
=
(prod_X phi_1^{1/|Y|} (p(X))) ^ {|Y|}
(prod_Y phi_2(q(Y)))
= (move each of |Y| instances of prod_X inside prod_Y, one per Y)
prod_Y (prod_X phi_1^{1/|Y|} (p(X)))
phi_2(q(Y))
= (now do the analogous operations to get phi_2(q(Y)) inside prod_X)
prod_Y prod_X phi_1^{1/|Y|} (p(X))
phi_2^{1/|X|} (q(Y))
=
prod_X,Y phi_3(p(X), q(Y))
which is the potential of a new parfactor with logical variables X, Y, potential function
phi_3(v1,v2) = phi_1^{1/|Y|}(v1) phi_2^{1/|X|}(v2)

Multiplying Parfactors

- In general:

sum_{ Atom }_Lvars:Constraint
(prod_Lvars1 : C1 phi_1(A_1))
…
(prod_Lvars_n : C_n phi_n(A_n))
these parfactors are multiplied into one:
sum_{ Atom }_Lvars:Constraint
prod_Lvars_1…Lvars_n : C1 and … and Cn
phi_1^c1(A1) … phi_n^{c_n}(A1)

where c_i = 1 / # Lvars_j : j != i . C1 and … and Cn
to compensate for the replication of phi_i due to its being inside a product of the Lvars that belong to the *other* parfactors

Alignment Multiplication

- Multiplication produces a parfactor with all logical variables from all multiplied factors.
- For complexity reasons, we want to keep the number of logical variables down.
- If two logical variables in distinct parfactors being multiplied have the same assignments, we can unify them. If X have the same domain:
- (prod_X phi1(p(X))) . (prod_Y phi2(q(Y)))
  =
  phi1(p(a))…phi1(p(z)) . phi2(q(a))…phi2(q(z))
  =
  phi1(p(a)) . phi2(q(a)) … phi1(p(z)) . phi2(q(z))
  =
  prod_X phi1(p(X)) . phi2(q(X))

Counting Variables (Milch et al)

- Example:
  prod_X,Y,Z : Z != b . phi(p(X), p(Y), q(Z)) =
  prod_Z : Z != b . prod_v phi(v1, v2, q(Z))^ {(#X : p(X) = v1)(#Y:p(Y)=v2)}
  prod_Z : Z != b . phi' (#X : p(X), #Y : p(Y), q(Z))
  where #X : p(X) and #Y:p(Y) are histogram-value random counting variable (two-bar histograms for boolean p(.), more for multi-value p(.))

- In general, if constraints if C_{X1} … C_{Xk} and C' do not share logical variables and Xi does not appear in Aj, for j != i, and not in atoms B:
  prod_X1, … Xk, Y: (C_X and C') . phi(A1(… X1 …), …, Ak(… Xk …), B(Y))
  =
  prod Y: C' . phi' (#X1 : C_{X1} . A1(… X1 …), #Xk : C_{Xk} . Ak(… Xk …), B(Y))
  where each
  #Xi : C_{Xi} . Ai(… Xi …)
  is a histogram-valued *counting variable* with dimension equal to the number of values in the range of Atom(… X …)

Why Counting Variables are Useful

- sum_{ {p(U)}$_U$ } sum_{ {q(V)}_{V :V!=b} }
  prod_X,Y,Z : Z != b . phi(p(X), p(Y), q(Z))
  has complexity dependent on domain size because of the summation on number of random variables dependent on domain size, and products also iterating over domain.
- But that's the same thing as
  sum_{ {p(U)}$_U$ } sum_{ {q(V)}_{V :V!=b} }
  prod_Z : Z != b . phi' (#X : p(X), #Y : p(Y), q(Z))
  =
  sum_{ {p(U)}$_U$ } sum_{ {q(V)}_{V :V!=b} } (#Y : p(Y) same as #X : p(X))
  prod_Z : Z != b . phi'' (#X : p(X), q(Z))
  =
  sum_{ #U : p(U) } sum_{ #V : V != b . q(V) }
  phi''' (#U : p(U), #V : V != b . q(V))
  which is now linear in domains of U and V

Partial Inversion

- sum_{ $\{p(U,W)\}_{U,W}$ } sum_{ $\{q(V)\}_{V:V!=b}$ }
  $\text{prod}_{R,X,Y,Z:Z!=b}$ phi(p(R,X), p(R,Y), q(Z))
- Cannot convert to counting because R is shared
- Note that R "slices" the set { $\{p(U,W)\}_{U,W}$ } in disjoint sets
- That would not be the case if we had
  p(R,X), p(Z,R) (mixing "columns and rows")
- sum { $\{q(V)\}_{V:V!=b}$ }
  prod_R sum_{ $\{p(R,W)\}_W$ } (X is now bound)
  $\text{prod}_{X,Y,Z:Z!=b}$ phi(p(R,X), p(R,Y), q(Z))
- sum { $\{q(V)\}_{V:V!=b}$ }
  prod_R sum_{ $\{p_R(W)\}_W$ } (p(R,.) can be seen as new predicate)
  $\text{prod}_{X,Y,Z:Z!=b}$ phi($p_R(X)$, $p_R(Y)$, q(Z))
- Simpler sub-problem, can be solved once for all R

Partial Inversion

- $\text{sum\_}\{ \{q(V)\}_{V:V!=b} \}$
  $\text{prod}_R \text{ sum\_}\{ \{p_X(W)\}_W \}$
  $\text{prod}_{X,Y,Z}\ \text{phi}(p_R(X),\ p_R(Y), q(Z))$

  =

  $\text{sum\_}\{ \{q(V)\}_{V:V!=b} \}$
  $\text{prod}_R \text{ sum\_}\{ \#W : p_X(W) \}$
  $\text{prod}_Z\ \text{phi'}((\#W : p_R(W)), q(Z))$

  =

  $\text{sum\_}\{ \#V : V!=b \,.\, q(V) \}$
  $\text{prod}_R \text{ sum\_}\{ \#W : p_R(W) \}$
  $\text{phi'}((\#W : p_R(W)), \#V: q(V))$

  = (defining phi''(v2) = sum_v1 phi'(v1, v2))

  $\text{sum\_}\{ \#V: V!=b \,.\, q(V) \}\ \text{prod}_R\ \text{phi''}(\#V: q(V))$

  = (defining phi'''(v) = phi''(v)^| Dom(R) |)

  $\text{sum\_}\{ \#V: V!=b \,.\, q(V) \}\ \text{phi'''}(\#V: q(V))$, linear in $|D(V)|$

Partial Inversion

- In general:
- We can invert on a set of logical variables appearing in the same argument position in a set of literals
- sum_{ A1(…) } … sum_{ A_m(…) }
  prod_X1,…Xk, Y1,…,Yn phi(…)
  and each Xj appear in Ai inside phi in same argument positions (this will cause them to slice the sets)
  =
  prod_X1,…,Xk
  sum_{ A1(…) with X1,…,Xk bound in their positions}
  …
  sum_{ A_m(…) with X1,…,Xk bound in their positions}
  prod_ Y1,…,Yn phi(…)

Generalizations and optimizations

- Nima Taghipour et al relaxed several conditions for both counting variables and inversion.
- Multiplying parfactors not always necessary, for example if your parfactors are clauses (Gogate)

Algorithm Overview

1. Pick a set of random variables
2. Split parfactors involving those
3. Multiply them into one parfactor
4. Apply Partial Inversion and Counting variable conversion as much as possible
5. If you get summations without products inside, perform them, otherwise ground one of the variables in the product and go to 4.