Projet MPRI 2.40 — Inférence semi-symbolique

Consignes : Le projet est à rendre pour le 15 Janvier 2024.

- Envoyer une archive `nom.[zip | tar.gz]` à `guillaume.baudart@inria.fr`.
- Inclure un rapport (format `.md` ou `.pdf`) dans l'archive.

Calculer exactement la distribution a posteriori décrite par un modèle est généralement un problème très difficile. Les méthodes d'inférences particulaires permettent d'approximer le résultat en rejouant le modèle de nombreuses fois de manière indépendante. Le calcul exact reste cependant possible pour les modèles les plus simples, ou pour certains sous-termes au sein d'un modèle complexe. Le but de ce projet est d'implémenter une méthode d'inférence semi-symbolique qui combine calcul symbolique exact et méthode particulaire approchée pour calculer la distribution a posteriori décrite par un modèle.

A. Préambule

Considérons les deux programmes suivants où `a`, `b`, `mu0`, `sigma0`, `sigma` sont des constantes.

```
let beta_bernoulli v =
  let p = sample (beta ~a ~b) in (* p ~ Beta(a, b) *)
  observe (bernoulli ~p) v;      (* v ~ Bernoulli(p) *)
  p

let gauss_gauss v =
  let mu = sample (gaussian ~mu:0. ~sigma:1.) in (* mu ~ N(0, 1) *)
  observe (gaussian ~mu:mu ~sigma:1.) v;         (* v ~ N(mu, 1) *)
  mu
```

Question 1. En utilisant la sémantique par noyaux vue en cours (cours 2), montrer que :

- $[\![\texttt{infer beta\_bernoulli}\ v]\!] = Beta(a+v,\ b+(1-v))$
- $[\![\texttt{infer gauss\_gauss}\ v]\!] = \mathcal{N}(m, s)$ avec $m = v/2$ et $s^2 = 1/2$

Plus généralement, si la distribution a posteriori est de la même famille que la distribution a priori après une observation, on dit que la distribution a priori et la distribution observée sont *conjugées*. On a ainsi :

- Si $\begin{cases} X \sim Beta(a,\ b) \\ Y \sim Bernoulli(X) \end{cases}$ alors $\begin{cases} Y \sim Bernoulli\left(\dfrac{a}{a+b}\right) & \textit{marginale} \\ (X \mid Y = v) \sim Beta(a+v,\ b+(1-v)) & \textit{conditionnement} \end{cases}$

- Si $\begin{cases} Y \sim \mathcal{N}(X, \sigma) \\ X \sim \mathcal{N}(\mu_0,\ \sigma_0) \end{cases}$ alors $\begin{cases} Y \sim \mathcal{N}(\mu_0, \sqrt{\sigma_0^2 + \sigma^2}) & \textit{marginale} \\ (X \mid Y = v) \sim \mathcal{N}(\mu_1,\ \sigma_1) & \textit{conditionnement} \end{cases}$

avec $\begin{cases} \mu_1 = \left(\dfrac{\mu_0}{\sigma_0^2} + \dfrac{v}{\sigma^2}\right) \Big/ \left(\dfrac{1}{\sigma_0^2} + \dfrac{1}{\sigma^2}\right) \\ \sigma_1^2 = 1 \Big/ \left(\dfrac{1}{\sigma_0^2} + \dfrac{1}{\sigma^2}\right) \end{cases}$

B. Calcul symbolique

On considère maintenant le type de distribution suivant.

```
type dist =
  | Beta of node * node
  | Bernoulli of node
  | Gaussian of node * node
  | V of float

and node = dist ref
```

Les paramètres de chaque distribution sont des références qui peuvent être mises à jour pendant l'inférence. Pour simplifier, on suppose que les valeurs concrètes sont toutes de type `float`.

Question 2. Implémenter les opérateurs probabilistes qui effectuent le calcul symbolique. `sample` associe simplement une variable à une distribution (sans tirer de valeur aléatoire), et `observe d v` met à jour les paramètres de la distribution `d` étant donné l'observation `v`. On lève une exception `Not_tractable` si les calculs symboliques échouent.

```
val sample : dist → node
(* return a new node with distribution [dist] *)

val observe : dist → float → unit
(* update the parameters of the distribution [dist] given an observation
   raise [Not_tractable] if this is not possible. *)

val infer : (α → node) → α → dist
(* return the posterior distribution of a model
   we assume that a model always returns a value of type [node]. *)
```

Nous avons maintenant un moteur d'inférence symbolique qui permet déjà de traiter les exemples de la section A. Malheureusement, cette implémentation ne fonctionne que pour très peu de modèles.

Question 3. Tester votre code sur les exemples de la section A.

Question 4. Donner un exemple (simple) de modèle pour lequel l'inférence échoue.

C. Inférence semi-symbolique

Pour traiter plus de cas, l'idée de l'inférence semi-symbolique est de combiner le calcul symbolique avec une méthode particulaire approchée. Chaque particule effectue les calculs symboliques de la Section B. Si les calculs symboliques échouent on tire aléatoirement des valeurs concrètes pour certaines variables aléatoires avant de continuer l'exécution. La distribution a posteriori est approximée par une *mixture* de distributions $\Sigma_{i=1}^{N} w_i \times d_i$ où N est le nombre de particules, w_i est le poids de la particule i, et d_i est la distribution calculée par la particule i.

Question 5. Implémenter l'opérateur `infer` avec l'algorithme *importance sampling* en utilisant un argument `prob` explicite (vu en cours).

```
type prob : {id: int; scores: float array}
val infer : (prob → α → β) → α → β Distribution.t
```

Notes :

- L'opérateur `infer` est polymorphe et peut être utilisé avec des modèles de type `prob` $\rightarrow \alpha \rightarrow$ `node` qui renvoient une distribution symbolique.
- L'implémentation de `infer` suppose que les opérateurs probabilistes `sample` et `observe` prennent un argument supplémentaire `prob` pour enregistrer le score des particules.

Question 6. Définir un opérateur `value` qui permet d'échantillonner un `node`, i.e., de fixer sa valeur à `V v` où `v` est une valeur tirée aléatoirement dans la distribution du `node` avant de renvoyer la valeur `v`.

```
val value : node → float
```

Question 7. Implémenter les opérateurs `sample` et `observe` pour finir l'implémentation de l'algorithme *importance sampling*. Pour cette question, chaque particule doit renvoyer une distribution `V v`.

```
val sample : prob → dist → node
val observe : prob → dist → float → unit
```

On veut maintenant ajouter le calcul symbolique de la section B à notre opérateur `observe`. Pour simplifier, on ne traite que les paires de variable aléatoires conjuguées X, Y où Y est la distribution observée dont l'un des paramètre est X (voir Section A).

Si les calculs symboliques ne sont pas possibles (X et Y ne sont pas conjuguées) on échantillonne X pour mettre à jour le score de la particule. Sinon (X et Y sont conjuguées) l'évaluation de `observe` Y `v` se décompose en 3 étapes:

- Si X est elle-même une distribution paramétrée, on commence par échantillonner les paramètres de X.
- On calcule la distribution *marginale* de Y qui permet de mettre à jour le score de la particule.
- On met à jour les paramètres de X en fonction de l'observation `v`.

Question 8. Implémenter l'opérateur `observe` avec ce nouveau comportement.

D. Évaluation

Question 9. Proposez quelques modèles simples pour illustrer les forces et les faiblesses de cet algorithme d'inférence.

Question 10. Comparer les performances des 3 algorithmes d'inférence sur ces exemples : symbolique, *importance sampling*, et semi-symbolique.

- Quelle est le surcoût du calcul symbolique pour un nombre de particules fixé ?
- Combien faut-il de particules pour atteindre une précision souhaitée ?

E. Extensions

Question 11. Quel est le problème si on remplace *importance sampling* par un *particle filter* pour l'implémentation de `infer` ? Proposer une solution.

Question 12. Quel est le problème si la valeur de retour d'un modèle n'est pas une variable aléatoire mais une expression, e.g., `let x = sample d in x + 1` ? Proposer une solution.

Question 13. Peut-on éviter d'échantillonner certaines variables pour améliorer les performances de l'algorithme ?

Plusieurs langages de programmation probabilistes exploitent les relations de conjugaisons pour améliorer la précision de l'inférence (Pyro, Augur, Birch, ProbZelus). Le moteur de calcul symbolique présenté ici est très simplifié (on ne traite que des paires de variables aléatoires conjuguées). Il est possible d'utiliser des moteurs symboliques plus puissants (e.g., *Delayed Sampling* ou *Belief Propagation*) au prix de structures de données plus élaborées.

▷

$$
\begin{aligned}
[\![\texttt{infer beta\_bernoulli } v]\!] &= \int \mathit{Beta}(a,\ b)(dp) \int_{()} \mathit{Bernoulli}_{\mathrm{pdf}}(p)(v) \\
&\propto \int p^{a-1}(1-p)^{b-1} * p^{v}(1-p)^{1-v} dp \\
&\propto \int p^{a+v-1}(1-p)^{b+(1-v)-1} dp \\
&\sim \mathit{Beta}(a+v,\ b+(1-v)) && \textit{def. of } \mathit{Beta}_{\mathrm{pdf}}
\end{aligned}
$$

$$
\begin{aligned}
[\![\texttt{infer gauss\_gauss } v]\!] &= \int \mathcal{N}(0,1)(d\mu) \int_{()} \mathcal{N}_{\mathrm{pdf}}(\mu,1)(v) \\
&\propto \int \exp(-\mu^2/2) * exp(-(\mu-v)^2/2) d\mu \\
&\propto \int \exp(-\mu^2 + \mu v - v^2/2) d\mu \\
&\propto \int \exp(-\mu^2 + \mu v) d\mu && v^2 \textit{ is a multiplicative constant} \\
&\propto \int \exp(-(\mu^2 - 2\mu m)/2s^2) d\mu && \textit{substitute } m = v/2 \textit{ and } s^2 = 1/2 \\
&\propto \int \exp(-(\mu^2 - 2\mu m)/2s^2 - m^2/2s^2) d\mu && \textit{add a constant term} \\
&\propto \int \exp(-(\mu-m)^2/2s^2) d\mu \\
&\sim \mathcal{N}(m,s) && \textit{def. of } \mathcal{N}_{\mathrm{pdf}}
\end{aligned}
$$

```
(* simple distribution type *)
type dist =
  | Beta of node * node
  | Bernoulli of node
  | Gaussian of node * node
  | V of float

(* nodes are ref that can be updated at runtime *)
and node = dist ref

(* build a constant node *)
let const (v : float) : node = ref (V v)

(* pretty print a [dist] *)
let rec pp_dist fmt dist =
  match dist with
  | Beta (alpha, beta) → Format.fprintf fmt "Beta(%a, %a)" pp_dist !alpha pp_dist !beta
  | Bernoulli p → Format.fprintf fmt "Bernoulli(%a)" pp_dist !p
  | Gaussian (mu, sigma) → Format.fprintf fmt "Gaussian(%a, %a)" pp_dist !mu pp_dist !sigma
  | V v → Format.fprintf fmt "%f" v

(* Symbolic computations only *)
module Symbolic = struct
  exception Not_tractable

  (* return the value of a constant node
     raise [Not_tractable] if the node is not constant. *)
  let value node = match !node with V v → v | _ → raise Not_tractable
```

```
  (* return a new node with distribution [dist] *)
  let sample (d : dist) : node = ref d

  (* update distribution [dist] given an observation
     raise [Not_tractable] if this is not possible. *)
  let observe (d : dist) (x : float) : unit =
    match d with
    | Bernoulli p → (
        match !p with
        | Beta (a, b) → (* case Beta/Bernoulli *)
            a := V (value a +. x);
            b := V (value b +. (1. -. x))
        | _ → raise Not_tractable)
    | Gaussian (m, s) → (
        match !m with
        | Gaussian (m0, s0) → (* case Gaussian/Gaussian *)
            let s2 = 1. /. ((1. /. (value s0 ** 2.)) +. (1. /. (value s ** 2.))) in
            m0 := V (s2 *. ((value m0 /. (value s0 ** 2.)) +. (x /. (value s ** 2.))));
            s0 := V (sqrt s2)
        | _ → raise Not_tractable)
    | _ → raise Not_tractable

  (* return the posterior distribution of a model
     we assume that a model always returns a value of type [node]. *)
  let infer (model : α → node) (obs : α) : dist = !(model obs)
end

let test_symbolic () =
  let open Symbolic in

  let coin () =
    let z = sample (Beta (const 1., const 1.)) in
    List.iter (observe (Bernoulli z)) [ 0.; 1.; 0.; 0.; 1.; 0.; 0.; 0. ];
    z
  in
  let dist = infer coin () in
  Format.printf "Dist coin : %a @." pp_dist dist;

  let gaussian_gaussian () =
    let mu = sample (Gaussian (const 0., const 1.)) in
    List.iter (observe (Gaussian (mu, const 1.0))) (List.init 100 (fun _ → 4.));
    mu
  in
  let dist = infer gaussian_gaussian () in
  Format.printf "Dist gaussian : %a @." pp_dist dist

(* here symbolic computation is not possible, raise [Not_tractable] *)
(* let coin_weird () =
     let z = sample (Gaussian (const 0.5, const 0.1)) in
     List.iter (observe (Bernoulli z)) [ 0.; 1.; 0.; 0.; 1.; 0.; 0.; 0. ];
     z
   in
   let dist = infer coin_weird () in
   Format.printf "Dist coin weird : %a @." pp_dist dist *)
```

```
module Semi_symbolic = struct
  (* import classic distributions with draw and logpdf *)
  open Byoppl
  open Distribution

  (* classic prob type for importance sampling *)
  type prob = { id : int; scores : float array }

  (* importance sampling, direct style, with explicit prob argument *)
  let infer ?(n = 10) model obs =
    let scores = Array.make n 0. in
    let values = Array.mapi (fun i _ → !(model { id = i; scores } obs)) scores in
    support ~values ~logits:scores

  (* draw a concrete value from the [dist] of a [node] and fix the node value *)
  let rec value node : float =
    let v = (* draw a concrete value *)
      match !node with
      | Beta (a, b) → draw (beta ~a:(value a) ~b:(value b))
      | Bernoulli p → draw (bernoulli_float ~p:(value p))
      | Gaussian (mu, sigma) → draw (gaussian ~mu:(value mu) ~sigma:(value sigma))
      | V v → v
    in
    node := V v; (* turn the node to a constant *)
    v (* return the value *)

  (* return a new node with distribution [dist] *)
  let sample (_prob : 'p) (d : dist) : node = ref d

  (* update distribution [dist] given an observation
     update the score of the current particle. *)
  let observe prob (d : dist) (x : float) : unit =
    let marginal = (* convert [dist] to a classic distribution *)
      match d with
      | Bernoulli p → (
          match !p with
          | Beta (a, b) → (* case Beta/Bernoulli *)
              let marg = bernoulli_float ~p:(value a /. (value a +. value b)) in
              a := V (value a +. x);
              b := V (value b +. (1. -. x));
              marg
          | _ → bernoulli_float ~p:(value p))
      | Gaussian (m, s) → (
          match !m with
          | Gaussian (m0, s0) → (* case Gaussian/Gaussian *)
              let m_0 = value m0 in (* save initial values that will be udpated *)
              let s_0 = value s0 in
              let s2 = 1. /. ((1. /. (value s0 ** 2.)) +. (1. /. (value s ** 2.))) in
              let marg = gaussian ~mu:m_0 ~sigma:(sqrt ((value s ** 2.) +. (s_0 ** 2.))) in
              m0 := V (s2 *. ((value m0 /. (value s0 ** 2.)) +. (x /. (value s ** 2.))));
              s0 := V (sqrt s2);
              marg
          | _ → gaussian ~mu:(value m) ~sigma:(value s))
      | Beta (a, b) → beta ~a:(value a) ~b:(value b)
      | V v → dirac ~v
    in (* update the score of particle prob.id *)
```

```
    prob.scores.(prob.id) ← prob.scores.(prob.id) +. logpdf marginal x

  (* pretty print mixture distributions computed by [infer] *)
  let pp_dist_support fmt (dist : dist Distribution.t) =
    let { values; probs; _ } = get_support ~shrink:true dist in
    Format.fprintf fmt "Support [ @[<hov>";
    Array.iter2
      (fun d p → Format.fprintf fmt "%f, %a; @," p pp_dist d)
      values probs;
    Format.fprintf fmt "]@]"
end

let test_semi_symb () =
  let open Semi_symbolic in

  let coin prob () =
    let z = sample prob (Beta (const 1., const 1.)) in
    List.iter (observe prob (Bernoulli z)) [ 0.; 1.; 0.; 0.; 1.; 0.; 0.; 0. ];
    z
  in
  let dist = infer coin () in
  Format.printf "Beta-Bernoulli: %a@." pp_dist_support dist;

  let gaussian_gaussian prob () =
    let mu = sample prob (Gaussian (const 0., const 1.)) in
    List.iter
      (observe prob (Gaussian (mu, const 1.0)))
      (List.init 100 (fun _ → 4.));
    mu
  in
  let dist = infer gaussian_gaussian () in
  Format.printf "Gauss-Gauss: %a@." pp_dist_support dist;

  (* Test with no conjugacy relation *)
  let coin_weird prob () =
    let z = sample prob (Gaussian (const 0.5, const 0.1)) in
    List.iter (observe prob (Bernoulli z)) [ 0.; 1.; 0.; 0.; 1.; 0.; 0.; 0. ];
    z
  in
  let dist = infer coin_weird () in
  Format.printf "Coin-Weird: %a@." pp_dist_support dist;

  (* [mu] should be symbolic, [z] should be sampled *)
  let weird prob () =
    let z = sample prob (Gaussian (const 0., const 10.)) in
    let mu = sample prob (Gaussian (z, const 0.1)) in
    observe prob (Gaussian (mu, const 0.5)) 1.;
    mu
  in
  let dist = infer weird () in
  Format.printf "Weird: %a@." pp_dist_support dist

let _ =
  Format.printf "@.XXX Symbolic XXX@.";
  test_symbolic ();
  Format.printf "@.XXX Semi-Symbolic XXX@.";
```

```
test_semi_symb ()
```