A Dynamic Programming Approach to Parameter Learning of Generative Models with Failure

Taisuke Sato SATO@MI.CS.TITECH.AC.JP
Yoshitaka Kameya KAMEYA@MI.CS.TITECH.AC.JP
Tokyo Institute of Technology / CREST

Abstract

We propose to apply dynamic programming to compute probabilities of failed goals for EM learning of generative models with failure described by symbolic-statistical modeling language PRISM. Programs for failed goals are synthesized deterministically by program transformation.

1. Introduction

The recent surge of interest in first-order statistical learning is motivated by the need of highly expressive language for modeling complex systems (De Raedt & Kersting, 2003). Logical variables and predicates in first-order logic together with probabilistic semantics enable us to define distributions over complex relational structures incorporating domain knowledge. PRISM[1] is a symbolic-statistical programming language designed for this purpose. It is a probabilistic extension of Prolog with a general EM learning routine based on formal semantics (Sato & Kameya, 1997; Sato & Kameya, 2001).

One of the most beneficial features of PRISM to the user is that parameter learning is for free. Distributions are defined by programs which consist of definite clauses and probabilistic built-in atoms. All we need for ML (maximum likelihood) estimation of parameters associated with probabilistic built-in atoms is just writing programs. The rest of the task is taken care of by the Prolog (tabled) search engine and the built-in general EM algorithm called *gEM* (graphical EM) algorithm (Kameya & Sato, 2000). Moreover thanks to dynamic programming nature of probability computation by the gEM algorithm, our EM learning is expected to be efficient. Actually adequately written PRISM programs for singly connected Bayesian networks, HMMs (hidden Markov models) and PCFGs (probabilistic context free grammars) are equivalent, time-complexity wise, to their specialized counterparts, i.e. EM learning by Pearl's belief propagation, the Baum-Welch algorithm and the Inside-Outside algorithm[2] respectively[3].

In PRISM semantics is well-defined for arbitrary programs. There is no restriction to Datalog programs or to range-restricted programs which are often imposed[4]. That being said however, we must add that EM learning is not without restrictions. PRISM excludes programs that do not satisfy '*no failure condition*' which states that computation must not eventually fail once probabilistic choices are made.

The objective of this paper is to remove this no failure condition thereby expanding the class of programmable models by PRISM to a wider class of models such as log-linear models. We allow programs to fail for example by conflicting constraints. Correspondingly when a program *DB* that may fail is given, we augment it with an auxiliary program that simulates failed computations caused by *DB* (sometime this is not possible though). The augmented program can be run on PRISM to infer parameters of *DB* by a new EM algorithm described in the **Appendix**.

Log-linear models cover a very wide class of probabilistic models and researchers have been seeking efficient EM algorithms (Abney, 1997; Riezler, 1998; Johnson

[1]URL = `http://sato-www.cs.titech.ac.jp/prism/`

[2]In the case of PCFGs, it is experimentally confirmed that the gEM algorithm runs faster than the Inside-Outside algorithm by orders of magnitude (Sato & Kameya, 2001).

[3]We also tested a variety of EM learning other than these popular models. The list includes Naive Bayes, linkage analysis, more sophisticated stochastic CFG grammars and stochastic graph grammars.

[4]Programs in which every variable in the head of a clause occurs in the body. Unit clauses must be ground.

et al., 1999) the latest one of which is the FAM algorithm, a specialized EM algorithm for SLPs (stochastic logic programs) proposed by Cussens (Cussens, 2001). He elegantly formulates the EM algorithm in the presence of failures by regarding observations as those with the failures truncated. FAM requires, however, to compute expected occurrences of clauses in the failed computations for which naive computation would obviously cause combinatorial explosion. Our approach opens a way to circumvent this problem by PRISM's dynamic programming.

Our contributions are as follows. We present a new EM algorithm which amalgamates the gEM algorithm and the FAM algorithm to perform ML estimation in a dynamic programming manner even in the presence of failed computations. An auxiliary PRISM program required by the new EM algorithm that simulates failed computations of the original program *DB* is synthesized from *DB* by deterministic program transformation. To our knowledge, this is the first attempt of program transformation in the logical-statistical setting. Also we present a novel class of constrained HMMs described by PRISM programs.

Our approach to probability computation is exact computation. We do not use sampling or other types of approximations. We take this approach because we believe that the dramatic increase of computation power nowadays makes exact computation more and more feasible and attractive as a means for fast probability computation. Also we would like to emphasize that our approach to modeling is generative. We describe a probabilistic model as a process that generates observable output like stochastic derivation of strings by PCFGs. Generative models are generally easy to understand and sampling is straightforward, but not necessarily outperform non-generative ones when we have poor knowledge of the generative mechanism of observations.

In the following, after providing preliminaries for logic programming and PRISM, we look at a motivating example in Section 3, and describe our compilation approach to failure in Section 4. We present the new EM algorithm in Section 5. Section 6 contains a modeling example of constrained HMM. Section 7 is the conclusion. The reader is assumed to be familiar with basics of logic programming (Sterling & Shapiro, 1986) and the EM algorithm (McLachlan & Krishnan, 1997).

2. Preliminaries

A logic program *DB* is a set of definite clause C of the form H `:-` $B_1, \ldots, B_n$ $(n \geq 0)$ where H (head) and B_i $(1 \leq i \leq n)$ (goal) are atoms. Every variable in C is universally quantified at the front of C. Clauses in *DB* are called program clauses. Hereafter we follow Prolog conventions and use strings beginning with upper case letters as variables[5]. So program clause `ancestor(X,Z) :- parent(X,Y), ancestor(Y,Z)` is read for example that for all `X`, `Y` and `Z` if `X` is a parent of `Y` and `Y` is an ancestor of `Z`, then `X` is an ancestor of `Z`. C is called a unit clause when $n = 0$. A clause or a term is said to be ground when no variable appears. The Herbrand universe (resp. the Herbrand base) of *DB* is the set of all ground terms (resp. ground atoms) whose function symbols (resp. function symbols and predicate symbols) occur in *DB*. A Herbrand interpretation is an assignment of truth values to each ground atom in the Herbrand base.

Computation by *DB* is nothing but search for a refutation of *DB* augmented with a query `:-` G such that $DB \vdash G\theta$ where θ is an answer substitution (variable bindings) for variables in G. Search is done by an SLD interpreter (SLD refutation procedure) which nondeterministically reduces a goal in a query to subgoals by a program clause. Define a set of ground atoms $I = \{A \mid DB \vdash A, A\,\text{ground atom}\}$. We identify I with a Herbrand interpretation M_{DB} such that $A \in I$ if-and-only if $M_{DB} \models A$. Then it is well-known that M_{DB} is a model of *DB*, i.e. satisfies every clause in *DB* and M_{DB} is the smallest as a set among Herbrand models of *DB*. M_{DB} is defined to be the denotation of *DB* as a program (*least model semantics*) (Doets, 1994).

PRISM generalizes this least model semantics probabilistically. A PRISM program DB' is the union of a set of definite clauses R and a set of ground atoms F representing probabilistic choices. DB' can be infinite. We give a probability measure P_F (*basic distribution*[6]) over the set of Herbrand interpretations of F and extends P_F to a probability measure $P_{DB'}$ over the set of Herbrand interpretations of DB' using R by way of least model semantics. We define $P_{DB'}$ as the denotation of DB' (*distribution semantics*) which regards ground atoms as binary random variables (Sato & Kameya, 2001). This extension is always possible for whatever P_F, but practical consideration restricts P_F to a (-n infinite) product of multinomial distributions.

Since distribution semantics is a generalization of least

[5]In Prolog, '`,`' (resp. '`;`') stands for conjunction (resp. disjunction) and '`=`' denotes unification.

[6]In this paper, we interchangeably use probability measure and probability distribution for the sake of familiarity.

model semantics, PRISM programs subsume logic programs. It also allows us to use arbitrary logic programs, arbitrary programming constructs like if-then, composition and recursion to define distributions and discrete stochastic processes. More importantly from a viewpoint of statistical learning, an EM algorithm is derived from this semantics for ML estimation of parameters specifying the multinomial distributions in P_F. The derived EM algorithm was general but too naive for real use. So we refined it to the gEM algorithm by incorporating the idea of dynamic programming (Kameya & Sato, 2000). The input of gEM is finite acyclic AND-OR graphs called *explanation (support) graph*s which encode statistical dependency among probabilistic atoms. An explanation graph for a goal G w.r.t. program DB' is obtained from tabled search for all refutations for `:-` G w.r.t. DB' by the SLD interpreter. Tabled search keeps the record of search in a table to prevent redundant search. PRISM adopts *linear tabling* as a tabling strategy (Zhou & Sato, 2003).

Here is an example of PRISM program. It defines a distribution over ground atoms of the form `bernoulli(`n`,`l`)` such that l is a list of outcomes of n coin tosses.

```
target(bernoulli,2).
values(coin,[heads,tails]).
:- set_sw(coin,0.6+0.4).

bernoulli(N,[R|Y]):-
    N>0,
    msw(coin,R),      % probabilistic choice
    N1 is N-1,
    bernoulli(N1,Y). % recursion
bernoulli(0,[]).
```

Figure 1. An example of PRISM program

Here `target(bernoulli,2)` is a declaration specifying the distribution of `bernoulli/2` atoms as a modeling target[7]. `values(coin,[heads,tails])` declares a discrete random variable named `coin` whose range is {`heads`,`tails`} which is implemented by atoms `msw(coin,`v`)` where v is either `heads` or `tails`[8].

`:- set_sw(coin,0.6+0.4)` is a directive on loading this program. It sets *parameter*s of `msw(coin,·)`, i.e. the probability of `msw(coin,heads)` to 0.6 and that of `msw(coin,tails)` to 0.4, respectively[9].

The next two clauses about `bernoulli/2` should be self-explanatory. They behave just like Prolog clauses except that `R` works as a random variable such that $P(\texttt{R} = \texttt{heads}) = 0.6$ and $P(\texttt{R} = \texttt{tails}) = 0.4$. The query `:- bernoulli(3,L)` will return for instance `L =[heads,heads,tails]`.

3. Loss of probability mass

PRISM programs fail just as Prolog programs do. Failure affects distributions and parameter learning, which can be seen by the following program.

```
target(test,1).
values(sw(0),[a,b,c]).
values(sw(1),[a,b,c]).
:- set_sw(sw0,0.5+0.3+0.2).
:- set_sw(sw1,0.5+0.3+0.2).

test(A):-
    msw(sw(0),A),
    msw(sw(1),B),
    A=B.          % A=B may fail
```

Figure 2. PRISM program causing failure

This program defines a distribution over `bernoulli/2`. It uses two probabilistic switches `sw(0)` and `sw(1)` to randomly choose one of {`a`,`b`,`c`}. They are i.i.d.s and represented by `msw` atoms whose parameters are set as $P(\texttt{msw(sw(0),a)}) = P(\texttt{msw(sw(1),a)}) = 0.5$, $P(\texttt{msw(sw(0),b)}) = P(\texttt{msw(sw(1),b)}) = 0.3$ and $P(\texttt{msw(sw(0),c)}) = P(\texttt{msw(sw(1),c)}) = 0.2$, respectively.

Suppose we call `:- test(X)` for sampling with `X` being variable. The clause head `test(A)` is unified and the leftmost goal `msw(sw(0),A)` is executed by randomly choosing `A`'s value from {`a`,`b`,`c`} according to probabilities set by `set_sw/2` directives. Suppose `a` is chosen. Next the second goal `msw(sw(1),B)` is executed similarly but independently. So it probabilistically happens that the sampled value of `B` is `b`. If this happens, the third goal `A=B`, the unification of `A` and `B`, fails and so does `:- test(X)`, which means we have no output, no observation despite sampling; we lost probability mass placed on `msw(sw(0),a)` and `msw(sw(0),b)`.

[7] `p/n` means that a predicate '`p`' has '`n`' arguments. We call an atom A `p` atom if the predicate symbol of A is `p`.

[8] `msw` atoms are most basic primitives in PRISM to make a probabilistic choice. They form constituents of the basic distribution and their probabilities are called *parameter*s.

[9] Parameters are inferred from observed goals consisting of `bernoulli/2` atoms by the gEM algorithm using `learn` command of PRISM.

Because failed computation yields no output, our observations should be interpreted as results of successful computations in our model. Consequently when we apply ML estimation to our observations, what must be maximized are conditional probabilities such as

$$P(\texttt{test(a)} \mid \exists \texttt{X}\, \texttt{test(X)}) = P(\texttt{test(a)})/P(\exists \texttt{X}\, \texttt{test(X)}).$$

$P(\exists \texttt{X}\, \texttt{test(X)})$ is the sum of probabilities of all successful computations for `:- test(X)` which is less than one[10]. In other words the current gEM algorithm is inapplicable as it merely maximizes unconditional probabilities of observations such as $P(\texttt{test(a)})$.

Instead of the gEM algorithm however, we can use the FAM algorithm (Cussens, 2001) to infer parameters. Unfortunately it requires to compute *unnormalized* expected occurrence $\mathbf{E}_{\texttt{fail}}[\eta(A)]$[11] of `msw` atoms A in all *failed* computations, which raises a question of how, generally, to capture all failed computations of a given program *DB*. We answer this question by synthesizing another program $DB^{\texttt{fail}}$ that can simulate failed computations of *DB*. Since failed computations of *DB* are exactly successful computations of $DB^{\texttt{fail}}$, $\mathbf{E}_{\texttt{fail}}[\eta(A)]$ is computed from $DB^{\texttt{fail}}$ efficiently by PRISM's dynamic programming. We next discuss how to obtain $DB^{\texttt{fail}}$.

4. Simulating failed computation

4.1. Negation elimination in non-probabilistic case by First-Order Compiler

We propose to synthesize $DB^{\texttt{fail}}$ by FOC (first-order compiler) (Sato, 1989). FOC is a deterministic program transformation algorithm that can compile negation, or more generally universally quantified implications[12] in a source program into an of executable logic program. Given a query `:- q(X)` and a logic program *DB* computing `q/1`, FOC eliminates negation automatically from *DB* ∪ { `failure :- not(exist([X],q(X)))` }. The resulting program $DB^{\texttt{fail}}$ faithfully simulates failed computations caused by query `:- q(X)` given to *DB*.

[10]
$$\begin{aligned} & P(\exists \texttt{X}\, \texttt{test(X)}) \\ &= P(\texttt{test(a)}) + P(\texttt{test(b)}) + P(\texttt{test(c)}) \\ &= P(\texttt{msw(sw(0),a)}) \times P(\texttt{msw(sw(1),a)}) \\ &\quad + P(\texttt{msw(sw(0),b)}) \times P(\texttt{msw(sw(1),b)}) \\ &\quad + P(\texttt{msw(sw(0),c)}) \times P(\texttt{msw(sw(1),c)}) \\ &= 0.38 < 1 \end{aligned}$$

[11] $\eta(A)$ is the number of occurrences of A. $\mathbf{E}_{\texttt{fail}}[\eta(A)]$ is not a conditional expectation : $\mathbf{E}_{\texttt{fail}}[\eta(A)] = \mathbf{E}[\eta(A) \mid \texttt{fail}]P(\texttt{fail})$ where `fail` denotes an occurrence of failure.

[12] Formulas of the form $\forall x(F \rightarrow G)$. Negation $\neg F$ is a special case because $\neg F$ is equivalent to $F \rightarrow \texttt{false}$.

Figure 3 is an example of negation elimination by FOC. As can be seen, '`not`' in the source program is

```
         *** source program ***

all_non_zero(L):- not(zero(L)).
                                % list L has no 0
zero(L):- mem(X,L),X=0.  % some X in L is 0
mem(X,[X|Y]).
mem(X,[H|Y]):-mem(X,Y).

        *** compiled program ***

all_non_zero(L):- closure_zero0(L,f0).
closure_zero0(L,C):-
   closure_mem0(L,f1(C)).
closure_mem0([],_).
closure_mem0([X|B],C):-
   cont(X,C),closure_mem0(B,C).
cont(X,f1(_)):- \+X=0.
```

Figure 3. Compilation example by FOC

compiled away and `all_non_zero(L)` in the compiled program computes `not(zero(L))` by definite clauses. The disunification `\+X=0` succeeds if-and-only-if the unification of `X` and `0` fails (negation-as-failure). The compiled program traces failed computations of the source program[13]. The (partial) correctness of FOC compilation is guaranteed by

Theorem 1 *(Sato, 1989) Suppose S, a source program is a set of clauses whose body include universally quantified implications, is compiled into S' by FOC. Then, iff(S)*[14] $\vdash A$ *(resp.* $\neg A$*) if* $S' \vdash A$ *(resp.* $\neg A$*) where A is a ground atom.*

This theorem roughly says that any computed result by the compiled program S' is a logical consequence of the source program S.

4.2. Probabilistic case

FOC was developed for non-probabilistic logic programs. Even when a program contains `msw` atoms and hence probabilistic however, compilation is possible.

[13] `closure_mem0([],_)` for example simulates the unification failure of `mem(X,L)` in case of `L=[]` with `mem(·,[·|·])`, the head of a `mem` clause.

[14] iff(S) is the union of S and some additional formulas reflecting Prolog's top-down proof procedure (Doets, 1994).

Suppose for example that we are asked to compile an implication $(\texttt{msw}(s,t) \rightarrow \phi)$ into an executable formula where s and t are terms and ϕ is some formula. Although msw is a probabilistic predicate, distribution semantics tells us to treat it as a normal predicate in compilation defined by some sampled atom, say $\texttt{msw}(s,v)$. We therefore compile the above formula into $(\texttt{msw}(s,\texttt{W}),(\texttt{W\==}t;\ (\texttt{W=}t,\phi)))$ where W is a new variable[15]. We can prove the correctness of this compilation under the condition that the compiled program terminates for any ground query regardless of sampling of msw atoms.

5. gEM for failure

5.1. Problem revisited

Suppose we write a PRISM program $DB_{\texttt{q}}$ to define a distribution of a target predicate q/1 which may fail[16]. Let $\texttt{q}(s_1),\ldots,\texttt{q}(s_T)$ be a random sample of length T. As pointed out in Section 3, if there is a loss of probability mass to failure, ML estimation should maximize $L(\boldsymbol{\theta}) = \prod_{t=1}^{T} P(\texttt{q}(s_t) \mid \exists \texttt{X}\, \texttt{q}(\texttt{X})) = \prod_{t=1}^{T} P(\texttt{q}(s_t))/P(\exists \texttt{X}\, \texttt{q}(\texttt{X}))$ where $\boldsymbol{\theta}$ collectively represents parameters to be estimated.

$L(\boldsymbol{\theta})$ can be maximized by the FAM algorithm (Cussens, 2001). It additionally computes in the E step, compared with non-failure case, $T * \mathbf{E}_{\texttt{fail}}[\eta(A)]/P(\exists \texttt{X}\, \texttt{q}(\texttt{X}))$ under the current $\boldsymbol{\theta}$ where $\mathbf{E}_{\texttt{fail}}[\eta(A)]$ is the unnormalized expected occurrence of msw atom A in all failed computations for `:- q(X)` w.r.t. $DB_{\texttt{q}}$. However the exact computation of $\mathbf{E}_{\texttt{fail}}[\eta(A)]$ as well as that of $P(\exists \texttt{X}\, \texttt{q}(\texttt{X}))$ is usually intractable if not impossible because of combinatorial explosion of computation paths. We solve this problem by using a combination of program transformation and dynamic programming

5.2. Augmentation with a compiled program for 'failure'

We extend PRISM's dynamic programming approach to the computation of $\mathbf{E}_{\texttt{fail}}[\eta(A)]$ by synthesizing a negation-free PRISM program using FOC that simulates the failed computations for `:- q(X)`.

Add a clause `failure :- not(exist([X],q(X)))` to $DB_{\texttt{q}}$ and let the augmented program be $\overline{DB}_{\texttt{q}}$. We remove negation from $\overline{DB}_{\texttt{q}}$ by FOC as described in Section 4 to obtain $\overline{DB}_{\texttt{q}}^{\texttt{fail}}$. $\overline{DB}_{\texttt{q}}^{\texttt{fail}}$ is a normal PRISM program without negation and dynamic programming is applicable to compute $P(\texttt{failure})$ and $\mathbf{E}_{\texttt{succ}}[\eta(A)]$, the unnormalized expected occurrence of msw atom A in the *successful* computations for `:- failure` w.r.t. $\overline{DB}_{\texttt{q}}^{\texttt{fail}}$.

We henceforth assume that $DB_{\texttt{q}}$ and $\overline{DB}_{\texttt{q}}^{\texttt{fail}}$ terminate with success or failure for query `:- q(X)` regardless of sampled values of msw atoms (*terminating condition*)[17].

We also assume that in a failed SLD derivation for `:- q(X)` w.r.t. $DB_{\texttt{q}}$ and in a successful derivation for `:- failure` w.r.t. $\overline{DB}_{\texttt{q}}^{\texttt{fail}}$, a goal has multiple callees only when it is an msw atom[18].

Then we can prove with one more assumption not mentioned here that $\mathbf{E}_{\texttt{fail}}[\eta(A)] = \mathbf{E}_{\texttt{succ}}[\eta(A)]$ (details omitted). Also we have $P(\exists \texttt{X}\, \texttt{q}(\texttt{X}) \mid \boldsymbol{\theta}) + P(\texttt{failure} \mid \boldsymbol{\theta}) = 1$ for any $\boldsymbol{\theta}$. So we can replace $\mathbf{E}_{\texttt{fail}}[\eta(A)]/P(\exists \texttt{X}\, \texttt{q}(\texttt{X}))$ in the E step by $\mathbf{E}_{\texttt{succ}}[\eta(A)]/(1 - P(\texttt{failure}))$, which is computable from $\overline{DB}_{\texttt{q}}^{\texttt{fail}}$ alone, just by treating 'failure' as a user-defined atom. We accordingly modify the gEM algorithm so that it additionally computes $\mathbf{E}_{\texttt{succ}}[\eta(A)]/(1 - P(\texttt{failure}))$ for every msw atom A in the E step.

5.3. New gEM algorithm

The modified new gEM algorithm is shown in the **Appendix**. Assuming appropriate conditions, it can perform EM learning in the presence of failure efficiently by dynamic programming. Modifications to the gEM algorithm are underlined. A brief explanation is in order (see (Sato & Kameya, 2001) for details of the gEM algorithm).

There $\overline{DB}^{\texttt{fail}}$ is, as in the previous subsection, the original program DB augmented with compiled clauses for 'failure' to simulate failed computations for the target predicate.

$\mathcal{G}' = G_0, G_1, \ldots, G_T$ is a list of observations $G_1,\ldots,G_T$ with special goal $G_0 = \texttt{failure}$. Each G_t $(0 \le t \le T)$ generates by tabled search an hierarchical graph called an explanation graph which is represented as an ordered list $\widetilde{\psi}_{DB}(\tau_k^t) = \{\tau_1^t,\ldots,\tau_{K_t}^t\}$. Each τ_k^t $(1 \le k \le K_t)$ is a disjunction of $\widetilde{S}_{k,j}^t$ $(1 \le j \le m_k)$

[15]This is a simplified compilation assuming that $\texttt{msw}(s,\cdot)$ atom is declared, i.e., the program includes a declaration of the form $\texttt{value}(s',\cdot)$ such that s is an instance of s'.

[16]We use a unary predicate for explanatory purpose.

[17]Or equivalently an SLD tree for `:-q(X)` w.r.t. $\overline{DB}_{\texttt{q}}^{\texttt{fail}}$ is finite regardless of sampled values of msw atoms.

[18]This assumption implies that computation proceeds deterministically as far as non-probabilistic predicates are concerned.

such that $\widetilde{S}^t_{k,j}$ is a conjunction of `msw` atoms and probabilistic atoms called table atoms.

The main routine **learn-gEM(** $\overline{DB}^{\texttt{fail}}$,$\mathcal{G}'$) performs EM learning calling two subroutines, **get-inside-probs(**$\overline{DB}^{\texttt{fail}}$, $\mathcal{G}'$) to compute inside probabilities and **get-expectations(** $\overline{DB}^{\texttt{fail}}$,$\mathcal{G}'$) to compute outside probabilities. Four arrays are used to store data, $\mathcal{P}[t,\tau]$ for the inside probability $P_{DB}(\tau \mid \boldsymbol{\theta})$, $\mathcal{Q}[t,\tau]$ for outside probability of τ w.r.t. G_t, $\mathcal{R}[t,\tau,\widetilde{S}]$ for $P_{DB}(\widetilde{S} \mid \boldsymbol{\theta})$ and finally $\eta[t,i,v]$ for the expected occurrence of `msw(`i`,`v`)` in a refutation for `:-` G_t w.r.t. DB.

Learning terminates when an increase of the log-likelihood of conditional probabilities is less than a given threshold ϵ. The time complexity in one iteration and the space complexity of the new gEM algorithm are linear in the size of the total size of the explanation graphs.

6. A learning example: constrained HMM

Here we give a learning example of generative model with failure. The example is small so that the reader can have an overview of our approach at a glance[19]. We introduce a class of HMM models with equality and disequality constraints (inequality constraints are treated similarly). For simplicity we describe an HMM model with a single constraint such that the first output alphabet and the last output alphabet must be identical. This model has five parameters. It seems difficult to express it succinctly by a PCFG with such a small number of parameters.

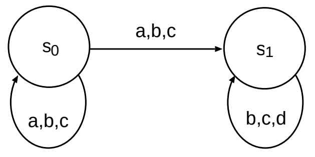

Figure 4. Two state HMM with constraint

Suppose here is a HMM which has two states $\{s_0, s_1\}$ and at s_0 one of $\{a,b,c\}$ is emitted and at s_1 one of $\{b,c,d\}$ is emitted, probabilistically. We place one constraint stated as above on this model.

A program $\overline{DB}_{\rm hmm}$ in Figure 5 specifies our model procedurally. It includes a `failure` clause saying that we

```
failure:-          % there is no output
  not(exist([X],hmm(X))).
string_length(5).  % output length is 5
hmm(Cs):-
  string_length(N),
  msw(obs(s0),C1), % s0 is an initial state
  N1 is N-1,       % C1 is a 1st alphabet
  Cs=[C1|Rest],    % keep C1 till end
  hmm(N1,s0,Rest,C1).
hmm(N,State,[C|Cs],C1):-
  N>1,
  msw(obs(State),C),
  ( State == s0, msw(tr(State),NextS)
  ; State == s1, NextS = s1 ),
  N1 is N-1,
  hmm(N1,NextS,Cs,C1).
hmm(N,State,[C1],C1):-
  N==1,  % the last alphabet must be C1
  msw(obs(State),C1).
```

Figure 5. A constrained HMM program $\overline{DB}_{\rm hmm}$

have no output string in spite of probabilistic choices made by `msw(obs(`$\cdot$`),`$\cdot$`)` (for alphabet emission) and `msw(tr(`$\cdot$`),`$\cdot$`)` (for state transition).

The subsequent clauses about `hmm/1` and `hmm/4` describe how state transition and alphabet emission at each state are made. For `:- hmm(Cs)`, we probabilistically choose an initial alphabet `C1` by executing `msw(obs(s0),C1)` and enter into recursion by `hmm/4` until the length of a generated string reaches the specified number, 5 in this case.

$\overline{DB}_{\rm hmm}$ is not runnable directly due to 'not' in the `failure` clause. We remove it by FOC and obtain a PRISM program $\overline{DB}^{\texttt{fail}}_{\rm hmm}$. We show part of it in Figure 6 concerning the computation of `failure` where '`==`' (resp. '`\==`') is a built-in predicate for strict equality (resp. strict disequality).

By inspection, we know that the search terminates at most in 5 steps. We also notice that the compiled program is tail recursive on `closure_hmm0/4`. So all refutation search for `:- failure` w.r.t $\overline{DB}^{\texttt{fail}}_{\rm hmm}$ by tabled search generates an explanation graph with trellis structure to which dynamic programming is effectively applicable. The time complexity of probability computations (that of one iteration in EM learning) by $\overline{DB}^{\texttt{fail}}_{\rm hmm}$, is $O(m^2 * n)$ where m is the number of states and n the length of input string, contrary to the exponential order caused by the naive non-dynamic programming approach. Also $O(m^2 * n)$ is equal to the

[19]We have applied our framework to PCFGs and found that EM learning is possible in polynomial time (and space) complexity in some cases. We also have conducted a learning experiment with a PCFG using a real corpus of moderate size, which will be reported elsewhere.

```
target(failure,0).
values(tr(s0),[s0,s1]).
values(obs(s0),[a,b,c]).
values(obs(s1),[b,c,d]).
:- set_sw(obs(s0),0.2+0.4+0.4).
:- set_sw(obs(s1),0.4+0.4+0.2).
:- set_sw(tr(s0),0.7+0.3).

failure:- closure_hmm0(f0).
closure_hmm0(A):-
   closure_string_length0(f2(A)).
closure_string_length0(A):- cont(5,A).
cont(A,f2(B)):-
   msw(obs(s0),C),
   D is A-1,
   closure_hmm0(D,s0,C,B).
closure_hmm0(A,B,C,D):-
  ( A>1, msw(obs(B),_),
     ( B\==s0
     ; B==s0, msw(tr(B),E), F is A-1,
         closure_hmm0(F,E,C,D) ),
     ( B\==s1
     ; B ==s1, G is A-1,
         closure_hmm0(G,s1,C,D) )
  ; A=<1 ),
  ( A\==1 ; A ==1, msw(obs(B),H), \+H=C ).
```

Figure 6. The compiled program $\overline{DB}^{\mathtt{fail}}_{\mathrm{hmm}}$ (part)

space complexity as it is the size of the explanation graphs for the failed and successful computations.

We computed P(`failure`) with parameters set by `set_sw` in $\overline{DB}^{\mathtt{fail}}_{\mathrm{hmm}}$ and obtained the following value[20].

```
?- prob(failure).
...
The probability of failure is: 0.66628
```

We also conducted a learning experiment with $\overline{DB}^{\mathtt{fail}}_{\mathrm{hmm}}$ using the new gEM algorithm in the **Appendix**. In each trial we randomly sampled 10,000 data by running `hmm/1` with the original parameters indicated in Figure 6 and then let $\overline{DB}^{\mathtt{fail}}_{\mathrm{hmm}}$ learn parameters from the generated data by using the new gEM algorithm with randomized initial values. We show in Figure 7 averages (with standard deviations) of parameters learned from 10 trials (threshold is 10^{-4}). For comparison, we add the averages of learned parameters by the gEM algorithm under the same condition.

[20]`prob(G)` is a PRISM built-in to compute the probability of a goal `G`.

msw name	learned parameter (ave. of 10 trials)		
`obs(s0)`	`a`	`b`	`c`
original	0.2	0.4	0.4
new gEM	$0.196(9.0 \times 10^{-4})$	$0.409(1.2 \times 10^{-4})$	$0.394(6.8 \times 10^{-4})$
gEM	$0.142(1.9 \times 10^{-4})$	$0.437(4.7 \times 10^{-4})$	$0.421(5.5 \times 10^{-4})$
`obs(s1)`	`b`	`c`	`d`
original	0.4	0.4	0.2
new gEM	$0.390(1.6 \times 10^{-3})$	$0.396(2.7 \times 10^{-3})$	$0.214(1.7 \times 10^{-3})$
gEM	$0.444(7.8 \times 10^{-4})$	$0.435(5.8 \times 10^{-4})$	$0.120(6.0 \times 10^{-4})$
`tr(s0)`	`s0`	`s1`	
original	0.7	0.3	
new gEM	$0.688(2.1 \times 10^{-3})$	$0.312(2.1 \times 10^{-3})$	
gEM	$0.713(1.2 \times 10^{-3})$	$0.286(1.2 \times 10^{-3})$	

Figure 7. Learned parameters

This table reads for instance P(`msw(obs(s0),a)`) $=$ 0.196 with standard deviation 9.0×10^{-4} is obtained by the new gEM algorithm. We see that the new gEM algorithm infers better parameters than the gEM algorithm.

7. Conclusion

We have proposed a new EM algorithm applicable to generative models with failure described by PRISM programs. It is an amalgamation of two EM algorithms, one the gEM algorithm which is based on dynamic programming (Kameya & Sato, 2000; Sato & Kameya, 2001) and the other the FAM algorithm which considers failure in refutation search for SLPs (Cussens, 2001).

To realize this amalgamation, we also proposed to apply FOC (first-order compiler) (Sato, 1989) to PRISM programs containing '`not`,' logical negation, to synthesize a program which is able to simulate failed computations by the original program.

We also explained how universally quantified logical formula with probabilistic built-ins are deterministically transformed to executable codes in PRISM while preserving distribution semantics.

References

Abney, S. (1997). Stochastic attribute-value grammars. *Computational Linguistics*, *23*, 597–618.

Cussens, J. (2001). Parameter estimation in stochastic logic programs. *Machine Learning*, *44*, 245–271.

De Raedt, L., & Kersting, K. (2003). Probabilistic logic learning. *ACM-SIGKDD Explorations, special issue on Multi-Relational Data Mining*, *5*, 31–48.

Doets, K. (1994). *From logic to logic programming.* The MIT Press.

Johnson, M., Geman, S., Canon, S., Chi, Z., & Riezler, S. (1999). Estimators for stochastic unification-based grammars. *Proceedings of the 37th Annual Meeting of the Association for Computational Linguistics (ACL'99)* (pp. 535–541).

Kameya, Y., & Sato, T. (2000). Efficient EM learning for parameterized logic programs. *Proceedings of the 1st Conference on Computational Logic (CL2000)* (pp. 269–294). Springer.

McLachlan, G. J., & Krishnan, T. (1997). *The EM algorithm and extensions.* Wiley Interscience.

Riezler, S. (1998). *Probabilistic constraint logic programming.* Doctoral dissertation, Universität Tübingen.

Sato, T. (1989). First order compiler: A deterministic logic program synthesis algorithm. *Journal of Symbolic Computation*, *8*, 605–627.

Sato, T., & Kameya, Y. (1997). PRISM: a language for symbolic-statistical modeling. *Proceedings of the 15th International Joint Conference on Artificial Intelligence (IJCAI'97)* (pp. 1330–1335).

Sato, T., & Kameya, Y. (2001). Parameter learning of logic programs for symbolic-statistical modeling. *Journal of Artificial Intelligence Research*, *15*, 391–454.

Sterling, L., & Shapiro, E. (1986). *The art of prolog.* The MIT Press.

Zhou, N.-F., & Sato, T. (2003). Efficient Fixpoint Computation in Linear Tabling. *Proceedings of the Fifth ACM-SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP2003)* (pp. 275–283).

Appendix: gEM for failure

The followings are the procedures of the new graphical EM algorithm for failure.

```
1:  procedure learn-gEM( $\overline{DB}^{\mathtt{fail}}$, $\mathcal{G}'$)
2:  begin
3:    Select some $\boldsymbol{\theta}$ as initial parameters;
4:    get-inside-probs($\overline{DB}^{\mathtt{fail}}, \mathcal{G}'$);
5:    $\lambda^{(0)} := \underline{\sum_{t=1}^{T} \ln\left(\mathcal{P}[t,G_t]/(1-\mathcal{P}[0,G_0]\right))}$;
6:    repeat
7:      get-expectations($\overline{DB}^{\mathtt{fail}}, \mathcal{G}'$);
8:      foreach $i \in I, v \in V_i$ do
9:        $\eta[i,v] := \underline{T * \eta[0,i,v]/(1-\mathcal{P}[0,G_0])}$
                 $+ \sum_{t=1}^{T} \eta[t,i,v]/\mathcal{P}[t,G_t]$;
10:     foreach $i \in I, v \in V_i$ do
11:       $\theta_{i,v} := \eta[i,v]/\sum_{v' \in V_i} \eta[i,v']$;
12:     get-inside-probs($\overline{DB}^{\mathtt{fail}}, \mathcal{G}'$);
13:     $m := m+1$;
14:     $\lambda^{(m)} := \underline{\sum_{t=1}^{T} \ln\left(\mathcal{P}[t,G_t]/(1-\mathcal{P}[0,G_0]\right))}$;
15:   until $\lambda^{(m)} - \lambda^{(m-1)} < \varepsilon$
16: end.
```

```
1:  procedure get-inside-probs($\overline{DB}^{\mathtt{fail}}, \mathcal{G}'$)
2:  begin
3:    for $t := 0$ to $T$ do begin
4:      Let $\tau_0^t = G_t$;
5:      for $k := K_t$ downto $0$ do begin
6:        $\mathcal{P}[t,\tau_k^t] := 0$;
7:        foreach $\widetilde{S} \in \widetilde{\psi}_{DB}(\tau_k^t)$ do begin
8:          Let $\widetilde{S} = \{A_1, A_2, \ldots, A_{|\widetilde{S}|}\}$;
9:          $\mathcal{R}[t,\tau_k^t,\widetilde{S}] := 1$;
10:         for $l := 1$ to $|\widetilde{S}|$ do
11:           if $A_l = \mathtt{msw}(i,\cdot,v)$ then
12:             $\mathcal{R}[t,\tau_k^t,\widetilde{S}] *= \theta_{i,v}$
13:           else $\mathcal{R}[t,\tau_k^t,\widetilde{S}] *= \mathcal{P}[t,A_l]$;
14:         $\mathcal{P}[t,\tau_k^t] += \mathcal{R}[t,\tau_k^t,\widetilde{S}]$
15:       end /* foreach $\widetilde{S}$ */
16:     end /* for $k$ */
17:   end /* for $t$ */
18: end.
```

```
1:  procedure get-expectations($\overline{DB}^{\mathtt{fail}}, \mathcal{G}'$)
2:  begin
3:    for $t := 0$ to $T$ do  begin
4:      foreach $i \in I, v \in V_i$ do $\eta[t,i,v] := 0$;
5:      Let $\tau_0^t = G_t$; $\mathcal{Q}[t,\tau_0^t] := 1.0$;
6:      for $k := 1$ to $K_t$ do $\mathcal{Q}[t,\tau_k^t] := 0$;
7:      for $k := 0$ to $K_t$ do
8:        foreach $\widetilde{S} \in \widetilde{\psi}_{DB}(\tau_k^t)$  do begin
9:          Let $\widetilde{S} = \{A_1, A_2, \ldots, A_{|\widetilde{S}|}\}$;
10:         for $l := 1$ to $|\widetilde{S}|$ do
11:           if $A_l = \mathtt{msw}(i,\cdot,v)$ then
12:             $\eta[t,i,v] += \mathcal{Q}[t,\tau_k^t] \cdot \mathcal{R}[t,\tau_k^t,\widetilde{S}]$
13:           else
14:             $\mathcal{Q}[t,A_l] += \mathcal{Q}[t,\tau_k^t] \cdot \mathcal{R}[t,\tau_k^t,\widetilde{S}]/\mathcal{P}[t,A_l]$
15:       end /*  foreach $\widetilde{S}$ */
16:   end /*  for $t$ */
17: end.
```