Notes on ProPPR

(DRAFT v0.02)

Vijay Saraswat
IBM T.J. Watson Research Center
1101 Kitchawan Road
Yorktown Heights, NY 10598
vijay@saraswat.org
(March 2017)

Abstract

Probabilistic logics offer a powerful framework for approximate representation and reasoning, key to working with domain-specific information, once one moves beyond surface-level extraction of meaning from natural language texts. We review a recent proposed framework, ProPPR [WMLC15] for probabilistic logic programming, intended to leverage ideas from approximate Personalized PageRank algorithms.

1 Introduction

[WMLC15] is an innovative attempt to combine machine learning ideas drawn from the literature on PageRank with the proof theoretic underpinnings of definite clause constraint logic programming (constrained SLD-resolution). This note is my attempt at understanding the ideas and laying out the underlying mathematical background clearly.

The basic idea is to view probabilistic SLD-resolution as a kind of graph traversal, and use ideas behind PageRank [PBMW99] to speed up traversal.

2 PageRank

First we cover some basics about PageRank, following [ACL06, ACL08]. We are interested in applying to constraint-SLD graphs, which we will develop in the next section. Crucially transitions in such graphs are associated with probabilities, hence we wish to consider the setting of directed, weighted graphs, unlike [ACL06].

Let $G = (V,E)$ be a directed graph with vertex set V (of size n) and edge set E (of size m). Let $\mathbf{1}_v$ be the n-vector which takes on value 1 at v and is 0 elsewhere. Let $d(v)$ be the out-degree of vertex $v \in V$. A *distribution* over V is a non-negative vector over V. By $\vec{k}$ we will mean the n-vector that takes on the value k everywhere, and by $\mathbf{1}_v$ we will mean the vector that is 0 everywhere except at v where it is 1. The *1-norm* of a distribution d is written $|d|_1$. The *support* of a distribution p, $Supp(p)$, is $\{v \mid p(v) \neq 0\}$. The *volume* of a subset $S \subseteq V$, $vol(S)$, is the sum of the degrees of the vertices in S.

A *Markov chain M* over G associates with each vertex a probability distribution over its outgoing edges. Specifically, M is an $n \times n$ matrix with positive elements, whose rows sum to 1 and whose non-zero elements M_{ij} (for $i, j \in 1 \ldots n$) are exactly (the transition probabilities of) the edges $(i, j) \in E$. Markov chains over G are our subject of interest.

Definition 2.1 *For a Markov chain M, the PageRank vector* $\mathrm{pr}_M(\alpha,s)$ *is the unique solution of the linear system*

$$\mathrm{pr}_M(\alpha,s) = \alpha s + (1-\alpha)\mathrm{pr}_M(\alpha,s)M \tag{1}$$

(Note that this definition is in line with [ACL08], but generalizes [ACL06] where it is given in an unweighted, undirected setting for the specific matrix $M = W = 1/2(I + D^{-1}A)$, where A is the adjacency matrix and D is the diagonal degree matrix.)

Proposition 2.2 (Linearity of *pr*$_M$) *For any* $\alpha \in [0,1)$ *there is a linear transformation* R_α *s.t.* $\mathrm{pr}_M(\alpha, s) = sR_\alpha$*, where*

$$R_\alpha = \alpha\Sigma_{t=0}^{\infty}(1-\alpha)^t M^t = \alpha I + (1-\alpha)MR_\alpha$$

Proposition 2.3

$$\mathrm{pr}_M(\alpha, s) = \alpha s + (1-\alpha)\mathrm{pr}_M(\alpha, sM) \qquad (2)$$

The proof is based on Proposition 2.2:

$$\begin{array}{rcll} pr_M(\alpha,s) & = & sR_\alpha & \\ & = & \alpha s + (1-\alpha)sMR_\alpha & \text{(Proposition 2.2)} \\ & = & \alpha s + (1-\alpha)pr_M(\alpha, sM) & \text{(Proposition 2.2)} \end{array}$$

The following ("PageRank Nibble") algorithm is taken from [ACL06] with a slight variation (working with a Markov chain rather than an adjacency matrix). The key insight is to work with a pair of distributions (p, r) satisfying:

$$p + pr_M(\alpha, r) = pr_M(\alpha, s) \qquad (3)$$

We shall think of p as an *approximate* PageRank vector, approximating $pr_M(\alpha, s)$ (from below) with *residual* vector r. Below we will use the notation $\mathrm{apr}_M(\alpha, s, r)$ to stand for a vector p in the relationship given by Equation 3. We are looking for an iterative algorithm that will let us approximate $pr_M(\alpha, s)$ as closely as we want. Specifically, we would like to get the probability mass in p, $|p|_1$, as close to 1 as we want.[1]

We now introduce the operation $push_u(p, r)$, generalizing [ACL06, Section 3] to the setting of Markov chains (and correcting some typos in [WMLC15, Table 2]):

1. Let $p' = p$ and $r' = r$ except for the following changes:
   (a) $p'(u) = p(u) + \alpha r(u)$
   (b) $r'(u) = (1-\alpha)r(u)M_{uu}$
   (c) For each v s.t. $(u, v) \in E$: $r'(v) = r(v) + (1-\alpha)r(u)M_{uv}$
2. Return (p', r').

It simulates one step of a random walk, at u, irrevocably moving some probability mass to u.[2]

The key lemma satisfied by this definition is:

Lemma 2.4 *Let* p', r' *be the result of the operation* $\mathrm{push}_u(p, r)$*. Then* $p' + \mathrm{pr}_M(\alpha, r') = p + \mathrm{pr}_M(\alpha, r)(= \mathrm{pr}_M(\alpha, s))$.

In proof, following [ACL06, Appendix], note that after a $push_u(p, r)$ operation the following is true:

$$\begin{array}{l} p' = p + \alpha r(u)\mathbf{1}_u \\ r' = r - r(u)\mathbf{1}_u + (1-\alpha)r(u)\mathbf{1}_u M \end{array}$$

[1] In this our interest is slightly different from [ACL06], which focuses on application to low conductance partitions.

[2] The definition given in [WMLC15, Table 2] differs in the update to r. However, we are not able to establish its correctness; indeed key lemmas below do not hold for that definition. We cannot relate the comment "α' is a lower-bound on $\Pr(v_0|u)$ for any node u to be added to the graph $\hat{G}$" to the code – given the term $(\Pr(v|u) - \alpha')\mathbf{r}[u]$ in the update to $\mathbf{r}[v]$ perhaps "α' is a lower-bound on $\Pr(v|u)$" was intended. But, in any case, we cannot see the need for this factor; in our code the corresponding factor is $(1-\alpha)r(u)M_{uv}$. We also cannot establish the subsequent assertion "it can be shown that afer each push $\mathbf{p} + \mathbf{r} = \mathbf{ppr}(v_0)$"; indeed we believe it is incorrect, the correct assertion is $\mathbf{p} + \mathbf{ppr}(\alpha, r) = \mathbf{ppr}(\alpha, v_0)$.

Now:

$$\begin{array}{rcll} p+pr_M(\alpha,r) & = & p+pr_M(\alpha,r-r(u)\mathbf{1}_u)+pr_M(\alpha,r(u)\mathbf{1}_u) & \text{(Linearity)} \\ & = & p+pr_M(\alpha,r-r(u)\mathbf{1}_u)+(\alpha r(u)\mathbf{1}_u+(1-\alpha)pr_M(\alpha,r(u)\mathbf{1}_uM) & (2) \\ & = & (p+(\alpha r(u)\mathbf{1}_u)+pr_M(\alpha,r-r(u)\mathbf{1}_u+(1-\alpha)r(u)\mathbf{1}_uM) & \text{(Linearity)} \\ & = & p'+pr_M(\alpha,r') & \end{array}$$

Some simple calculations establish:

Lemma 2.5 *Let p',r' be the result of the operation* push$_u(p,r)$. *Then $|p'|_1+|r'|_1=|p|_1+|r|_1$.*

Now define the ApproxPageRank(v,α,e) algorithm as follows:

1. Let $p=\vec{0}$ and $r=\mathbf{1}_v$.
2. While there exists a vertex $u\in V: r(u)\geq \varepsilon d(u)$, apply $push_u(p,r)$.
3. Return p.

The value returned is an apr$(\alpha,\mathbf{1}_v,r)$ s.t. for all $u\in V, r(u)<\varepsilon d(u)$. Hence we get an upper bound of εm on $|r|_1$ (by summing over all vertices) at the end of the program.

The main results are as follows, with proofs as in [ACL06, Appendix].

Lemma 2.6 *([ACL06, Lemma 2]) Let T be the total number of push operations performed by ApproxPageRank, and let d_i be the degree of the vertex u used in the i'th push. Then $\Sigma_{i=1}^{T}d_i\leq 1/\varepsilon\alpha$.*

Theorem 2.7 *ApproxPageRank(v,α,ε) runs in time $O(1/\varepsilon\alpha)$, and computes an approximate PageRank vector $p=apr_M(\alpha,\mathbf{1}_v,r)$ s.t.* $\max(1-\varepsilon m,\alpha\varepsilon\Sigma_{i=1}^{T}d_i)<|p|_1\leq 1$.

We note in passing that the termination condition for the ApproxPageRank(v,α,e) algorithm could be changed (e.g. to $r(u)\geq d(u)^c\varepsilon$, for some constant c) without affecting the correctness of the algorithm.

3 Constraint-SLD resolution

Though [WMLC15] is presented for just definite clause programs, we shall follow the tradition of logic programming research and consider constraint logic programming, after [JL87]. This gives us significant generality and lets us avoid speaking of syntactic notions such as most general unifiers. Hence we assume an underlying constraint system $\mathcal{C}$ [Sar92], defined over a logical vocabulary. Atomic formulas in this vocabulary are called *constraints*. $\mathcal{C}$ specifies the notions of *consistency* of constraints and *entailment* between constraints. We assume for simplicity the existence of a vacuous constraint `true`.

We assume a fixed program P, consisting of a (finite) collection of (implicitly universally quantified) clauses $h\leftarrow c,b_1,\ldots,b_k$ (where h,b_i are atomic formulas and c is a constraint). Below, for a formula ϕ we will use the notation $var(\phi)$ to refer to the set of variables in ϕ. Given a set of variables Z and a formula ϕ by $\delta Z\,\phi$ we will mean the formula $\exists V\,\phi$ where $V=var(\phi)\setminus Z$.

We assume given an initial *goal* g (an atomic formula), with $Z=var(g)$. A *configuration* (or *state*) s is a pair $\langle a_1,\ldots,a_n;c\rangle$, with $n\geq 0$, c a constraint, and goals a_i. The *variables* of the state are $var(a_1\wedge\ldots\wedge a_n\wedge c)$. s is said to be *successful* if $n=0$, *consistent* if c is consistent and *failed* (or *inconsistent*) if c is inconsistent.

Two states $\langle a_1,\ldots,a_n;c\rangle$ and $\langle b_1,\ldots,b_k;d\rangle$ are equivalent if $\vdash \delta Z(a_1\wedge\ldots\wedge a_n\wedge c)\Leftrightarrow\delta Z(b_1\wedge\ldots\wedge b_k\wedge d)$ (where the $\vdash$ represents the entailment relation of the underlying logic, including the constraint entailment relation). Note that any two inconsistent states are equivalent, per this definition.

We now consider the transitions between states. For simplicity, we shall confine ourselves to a fixed *selection rule*. Given the sequence of goals in a state, a selection rule chooses one of those goals for execution. Logically, any goal can be chosen (e.g. Prolog chooses the first goal).

A clause is said to be *renamed apart* from a state if it has no variables in common with the state. If $g=p(s_1,\ldots,s_k)$ and $h=p(t_1,\ldots,t_k)$ are two atomic formulas with the same predicate p and arity k, then $g = h$ stands for the collection of equalities $s_1 = t_1, \ldots, s_k = t_k$. We say that a state $s = \langle a_1, \ldots, a_n; c\rangle$ *can transition to* a state $\langle a_1, \ldots, a_{i-1}, b_1, \ldots, b_k, a_{i+1}, \ldots a_n; c, d, (a_i = h)\rangle$ provided that (a) s is consistent, (b) a_i is chosen by the selection rule, (c) there is a clause $C = h \leftarrow d, b_1, \ldots, b_k$ in P, renamed apart from s s.t. h and a_i are atomic formulas with the same predicate name and arity. We say that a_i is the *selected goal* (for the transition) and C the *selected clause*.

Note that the current state will have at most k states it can transition to, if the program has k clauses with the predicate and arity of the selected goal.[3] It will have fewer than k if resulting states are equivalent. Of course, not all resulting configurations may be consistent. Finally, note that a state may transit to itself.[4]

Constraint-SLD resolution starts with a state $\langle g; \texttt{true}\rangle$ and transitions to successive states, until a state is reached which is successful or failed.

We are interested in *stochastic logic programs*. For the purposes of this note, these are programs that supply with each transition a *probability* for the transition (a non-negative number bounded by 1) in such a way that the sum of the probabilities across all transitions from this state is 1. The probabilities may depend on the current state. In stochastic logic programs as described in [Mug96] the probability is a number directly associated with the clause (and independent of the state). [WMLC15] describes a more elaborate setting: a clause specifies "features" (dependent on the current state) which are combined with a (learnt) matrix of weights to produce the probability. For the purposes of this note we shall not be concerned with the specific mechanism.

Note that multiple transitions from a state s (each using a different clause) may lead to the same (equivalent) state t. In such cases we consider that there is only one transition $s \to t$, and its associated probability is the sum of the probabilitives across all clauses contributing to the transition.

In general, we will only be concerned with goals that have a finite derivation graph. This can be guaranteed by placing restrictions on the expressiveness of programs (e.g. by requiring that programs satisfy the Datalog condition), but we shall not make such further requirements.

3.1 Learning weights via training, using probabiistic constraint-SLD resolution

Top-down proof procedures for definite clauses, such as SLD, are readily adaptable to probabilistic calculations proofs and are not susceptible to the "grounding" problem that plagues Markov Logic Networks [DR07]. See e.g. [Sar16] for an implementation. One simply implements a meta-interpreter which carries the probability mass generated on the current branch, multiplying the current value with the probability of the clause used to extend the proof by one step, discarding failed derivations, and summing up the results for different derivations of the same result.

Training can be performed in a routine fashion. Given a proof tree for a particular query, the features used in each clause in the proof (and hence the weights used) are uniquely determined. A loss function similar to the one in [WMLC15, Sec 3.3] can be used to determine the gradient, and this can be propagated back to each weight used in the proof. Training for each query can be performed independently (the proof trees construted in parallel); though, as usual for SGD, weights must be updated using the gradients from all examples (e.g. in a mini-batch).

Note that the procedure described here naturally takes care of "impure" programs, that is, programs some of whose predicates have non-probabilitic clauses. This is the case in most real-world programs – certain parts of the program (e.g. those that deal with operations on data-structures) are usually deterministic. While finding a probabilistic SLD-refutation, steps that do not involve a probabilistic goal do not contribute to updating the probability associated with the branch.

In passing, we note that it is advisable to adopt an "Andorra"-style execution strategy which favors the selection of non-probabilistic predicates for execution, as long as there are any in the current resolvent. Among probabilistic goals, those with the least degree may be preferred. Most (constraint)

[3]Note that in theory a clause has an infinite number of variants that are renamed apart from a given state. It can be shown that only one of them needs to be considered for selection, the results for all other choices can be obtained by merely renaming the results for this choice.

[4]Consider for instance a program with a clause $p(X) \leftarrow p(X)$, and configuration $\langle p(U), \texttt{true}\rangle$, with $Z = \emptyset$. This configuration is equivalent to the one obtained after transition, $\langle p(X), \texttt{true}\rangle$.

logic programming implementations do not canonicalize and record generated states because of the book-keeping expense. Whether that is useful in the current setting should be determined empirically.

4 Applying PageRank to constraint-SLD graphs

We consider now the application of PageRank to constraint-SLD graphs, as described in [WMLC15] (but modified per Sections 2 and 3) and discuss its features.

[WMLC15, Sec 3.2] proposes to use the PageRank-Nibble algorithm (Section 2) to generate a constrained-SLD graph, per the following procedure ([WMLC15, Table 4] with some typos corrected):

1. Let $p = \text{ApproxPageRank}(\langle Q;\texttt{true}\rangle, \alpha, \epsilon)$.
2. Let $S = \{u : p(u) > \epsilon, u = \langle;c\rangle\}$.
3. Let $Z = \Sigma_{u \in S} p(u)$.
4. For every solution $u = \langle;c\rangle$ define $\Pr(u) = (1/Z)p(u)$.

Running ApproxPageRank will produce an SLD-graph with $O(1/\alpha\epsilon)$ nodes; this is independent of the size of the program (and its included database). The algorithm will also work with loops in the graph (as might be generated, for example, by recursive programs).

Unfortunately, the procedure is oblivious to the logical interpretation of the graph. In particular there is no guarantee that on termination the graph will contain *any* proof of the query described by the initial node v. Worse, even if the graph contains a successful terminal node (corresponding to a proof), if is does not contain nodes corresponding to *all* the proofs, the estimate of probability (computed in the last line of the algorithm above) could be arbitrarily off as we now analyze.

Consider a solution $u \in S$ computed by the PageRank-Nibble algorithm, with probability estimate $p(u)$. We consider now numerous factors that bear on the relationship between $p(u)$ and $\Pr(u)$, the true probability of u.

First, note that (unlike the claim in [WMLC15][5]) the error on $p(u)$ is *not* bound by $\epsilon d(u)$. While $r(u) < \epsilon d(u)$, the error $pr_M(\alpha, \mathbf{1}_v)(u) - p(u)$ is $pr_M(\alpha, r)(u)$, not $r(u)$. The only other conclusion that can be made is on the lower bound for $|p|_1$ (of which $p(u)$ is a summand).

Second, note that the probability of a final solution is the sum of all paths to that solution, divided by the sum of probabilities of all solutions. In particular the probability mass $|p|_1 - Z$ is allocated to either failed nodes or intermediate nodes. Ultimately, all of this must be divided up among successful nodes.

Let `false` stand for the node corresponding to the inconsistent state, with probability estimate $p(\texttt{false})$. The residual unallocated probability mass is then $1 - (Z + p(\texttt{false}))$. Some of it may go to failed nodes, rest must go to successful nodes. Consider several possibilities:

1. All the probability mass goes to `false`. In this case, $\Pr(u) = p(u)/Z$, as estimated above.
2. All the probability mass goes to other solutions. In this case, $\Pr(u) = p(u)/(1 - p(\texttt{false}))$.
3. All the probability mass goes to u. In this case, $\Pr(u) = (p(u) + (1 - (Z + p(\texttt{false}))))/(1 - p(\texttt{false})) = 1 - (Z - p(u))/(1 - p(\texttt{false}))$.

As we can see, the actual value may be a factor of Z off (second case, $p(\texttt{false}) = 0$, $\Pr(u) = p(u)$). Numerically, $\Pr(u)$ may be close to 1, even though $p(u)/Z$ is extremely small (e.g. $p(u) = 0.001, Z = 0.1, p(\texttt{false}) = 0$ gives an estimate of 0.01 for a true value of 0.901).

Thus $p(u)/Z$ cannot be used as a reliable predictor for $\Pr(u)$ unless Z is close to 1. But how much work has to be expended to get to a proof-graph for which Z is close to 1 ultimately depends on the logical structure of the program and may not be very strongly dependent on the particular control strategy used to develop the proof-graph.

[5][WMLC15, Sec 2.2] "Specifically, following their proof technique, it can be shown that after each push, $\mathbf{p} + \mathbf{r} = \mathbf{ppr}(v_0)$. It is also clear that when PageRank-Nibble terminates, then for any u, the error $\mathbf{ppr}(v_0)[u] - \mathbf{p}[u]$ is bounded by $\epsilon|N(u)| \ldots$"

Conclusions. We believe that a practical probabilistic logical system can be built utilizing the idea of per-clause features, and trainable weights to learn probability distributions. However, PageRank-based ideas may not be as useful as we thought.

Acknowledgements. Thanks to Radha Jagadeesan, Kyle Gao and Cristina Cornelio for discussions, and to William Cohen and Kathryn Mazaitis for responding to questions. The responsibility for conclusions drawn above remains mine...

References

[ACL06] Reid Andersen, Fan Chung, and Kevin Lang. Local graph partitioning using pagerank vectors. In *Proceedings of the 47th Annual IEEE Symposium on Foundations of Computer Science*, FOCS '06, pages 475–486, Washington, DC, USA, 2006. IEEE Computer Society.

[ACL08] Reid Andersen, Fan Chung, and Kevin Lang. Local partitioning for directed graphs using pagerank. *Internet Math.*, 5(1-2):3–22, 2008.

[DR07] P. Domingos and M. Richardson. *An Introduction to Statistical Relational Learning*, chapter Markov Logic Networks: A Unifying Framework for Statistical Relational Learning. MIT Press, 2007.

[JL87] J. Jaffar and J.-L. Lassez. Constraint logic programming. In *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '87, pages 111–119, New York, NY, USA, 1987. ACM.

[Mug96] Stephen Muggleton. Stochastic logic programs. In *New Generation Computing*. Academic Press, 1996.

[PBMW99] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking: Bringing order to the web. Technical Report 1999-66, Stanford InfoLab, November 1999. Previous number = SIDL-WP-1999-0120.

[Sar92] Vijay A. Saraswat. The Category of Constraint Systems is Cartesian-Closed. In *LICS*, 1992.

[Sar16] Vijay Saraswat. A Quick and Dirty Implementation of Probabilistic CCP (pcc) in Prolog, 2016.

[WMLC15] William Yang Wang, Kathryn Mazaitis, Ni Lao, and William W Cohen. Efficient inference and learning in a large knowledge base. *Machine Learning*, 100(1):101–126, 2015.