An Analysis of Gibbs Sampling for Probabilistic Logic Programs

Damiano Azzolini[1⋆], Fabrizio Riguzzi[2], and Evelina Lamma[1]

[1] Dipartimento di Ingegneria – University of Ferrara
Via Saragat 1, I-44122, Ferrara, Italy
[2] Dipartimento di Matematica e Informatica – University of Ferrara
Via Saragat 1, I-44122, Ferrara, Italy
[damiano.azzolini,fabrizio.riguzzi,evelina.lamma]@unife.it

Abstract. Markov Chain Monte Carlo (MCMC) is one of the most used families of algorithms based on sampling. They allow to sample from the posterior distribution when direct sampling from it is infeasible, due to the complexity of the distribution itself. Gibbs sampling is one of these algorithm that has been applied in many situations. In this paper we compare an implementation of Gibbs sampling for Probabilistic Logic Programs on several datasets, in order to better understand its performance. For all the experiments we compute the convergence time, execution time and population standard deviation of the samples.

Keywords: Gibbs Sampling, Markov Chain Monte Carlo, Probabilistic Logic Programming.

CEUR-WS.org/Vol-2678/paper12.pdf

1 Introduction

Probabilistic Logic Programming has been proved effective in modelling several real world situations [3,10], thanks to the possibility of representing probability models with an expressive language such as logic programming.

Inference in probabilistic logic programs is a computationally hard task. There are two main types of inference. The first type is exact inference, which is usually based on the representation of the program in a compact form such as decision diagrams, and it is aimed to compute exact answers. However, this type of inference is not always easy to apply due to some limitations, among them, the time required to get an answer. Approximate inference based on sampling may overcome the exact inference limitations but with a cost: the accuracy of the results depends on the number of samples. Moreover, different algorithms may have different performance on the same dataset.

Among the approximate inference algorithms, Markov Chain Monte Carlo (MCMC) allows to efficiently sample from the posterior distribution. In this

⋆ Copyright © 2020 for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

paper, we focus on one of them, Gibbs sampling, and we test it on several datasets in order to get a quantitative analysis of its performance.

The paper is structured as follows: in Section 2 we introduce the basic concepts of Probabilistic Logic Programming and in Section 3 we analyze approximate inference and Gibbs sampling. Section 4 illustrates the results of the experiments and Section 5 concludes the paper.

2 Probabilistic Logic Programming

Here we consider Probabilistic Logic Programs (PLPs) based on the Distribution Semantics [15] and in particular Logic Programs with Annotated Disjunctions (LPADs) proposed in [17] All the languages based on this semantics have the same expressive power but they differ in the representation of the choices for different clauses.

An LPAD clause has the form $h_1 : \pi_1; \ldots ; h_n : \pi_n \text{ :- } b_1, \ldots , b_m$ where the head is a set of logical atoms and the body is a set of logical literals. π_is are real values in the interval $[0, 1]$ that represent the probability that the i-th head is chosen. Moreover, $\sum_i \pi_i = 1$. If this is not true, i.e. $0 < \sum_i \pi_i < 1$ an extra *null* atom is added to the head with probability $1 - \sum_i \pi_i$.

A PLP program defines a probability distribution over *worlds*. Recall that a *substitution* is a function mapping variables to terms represented with $\theta = \{X_1/t_1, \ldots , X_n/t_n\}$, where X_k/t_k means that all the occurrences of the variable X_k are replaced by the term t_k. A world can be obtained by choosing one atom from the head of a grounding of a LPAD clause. The probability of a query is then computed by considering a joint probability distribution between the query and the worlds and by summing out the worlds.

The previous definition can only be applied for programs without functions symbols, since their grounding is finite. In case of function symbols, the definition must be extended. See [12,13] for a complete treatment of the field.

Let us introduce an example of LPAD:

$$
\begin{array}{l}
mistake(X) : 0.1 \text{ :- } good_player(X). \\
mistake(X) : 0.05 \text{ :- } focused(X). \\
good_player(kasparov). \\
focused(kasparov).
\end{array}
$$

The previous program represents a situation where X makes a mistake with probability 0.1 if X is a *good_player* and nothing happens with probability 0.9. Similarly, if X is *focused* during the match, he will make a mistake with probability 0.05 and nothing happens with probability 0.95. Moreover, we know for sure that *kasparov* is a *good_player* and he is *focused* during a match. We may be interested, for instance, in the probability that kasparov makes a mistake, denoted with $P(mistake(kasparov))$. This can be computed as: $0.1 \cdot 0.05 + 0.1 \cdot (1 - 0.05) + (1 - 0.1) \cdot 0.05 = 0.145$.

The probability of the previous example can be computed exactly since it has a finite grounding. Consider now the following example, representing a player

that repeatedly toss a coin.

$$
\begin{aligned}
&heads(N) : 0.5; tails(N) : 0.5.\\
&on(0, h) :- \ heads(0).\\
&on(0, t) :- \ tails(0).\\
&on(s(N), h) :- \ \sim on(N, t), heads(s(N)).\\
&on(s(N), t) :- \ \sim on(N, t), tails(s(N)).\\
&at_least_once_head :- \ on(_, h).
\end{aligned}
$$

If we want to know the probability that the player will get at least once heads ($P(at_least_once_head)$), exact inference cannot be used since the grounding of the program is infinite. However, using approximate methods based on sampling, we can sample the query a certain number of times and return the number of successes over the number of samples. As the number of samples goes to infinity, the approximated probability reaches value 0.5 (since we need to compute the sum of a geometric series).

In this paper, we utilize the *cplint* framework [1,14] that also allows the definition of densities using the syntax `A:Density :- Body`. For example,

$$var(X) : gaussian(X, \mu, \sigma^2)$$

states that the variable X follows a Gaussian distribution with mean μ and variance σ^2.

The computation of the probability of a query, a task called *inference*, can be performed both with exact and approximate methods. Exact inference from discrete programs (programs without continuous random variables) is a #P-complete task [8] so, as the size of the problem increases, the execution time becomes intractable. Currently there are no solutions that perform exact inference in hybrid domains (i.e., domains characterized by both discrete and continuous random variables) with complex relationship among the variables in acceptable time. For all the previous reasons, approximate inference methods may be more suitable in several situations.

3 Approximate Inference

Approximate inference methods may be implemented as a variation of exact inference algorithms or can be based on sampling. Here we focus on the latter. One of the most used approximate approach is given by Monte Carlo sampling: in a nutshell, the algorithm samples a world by sampling every ground probabilistic fact, checks if the query is true in the world and computes the probability of the query as the fraction of samples where the query is true. This process is repeated for a fixed number of steps or until convergence (i.e., the difference between two consecutive samples is less than a certain threshold).

The cplint framework [1,14] allows both exact and approximate inference. Monte Carlo methods are implemented in the MCINTYRE [11] module, that performs inference by querying a transformed program where multiple heads

generates multiple clauses with the same body. Here we focus on conditional approximate inference, i.e., computing the probability of a query when an event (o a set of events) has been observed.

The simplest algorithm used to perform this task is rejection sampling: it queries the evidence in a sample program and, if the query is successful, queries the goal. The probability is then computed and the fraction of successes over the number of samples. Despite its simplicity, rejection sampling may suffer from the fact that, if the evidence has low probability, a lot of samples are discarded making the algorithm very slow to converge.

An alternative is represented by the Markov Chain Monte Carlo (MCMC) methods. The main idea behind MCMC is to construct a Markov Chain that has as equilibrium distribution the desired distribution. Then, samples can be drawn from this Markov Chain in order to get the probability of a query. As the number of steps increase, the approximation gets closer to the real distribution. In the limit, MCMC can represent the real posterior distribution. A common operation is to discard the first samples since they do not represent the real distribution, with an operation called *burn-in*. There are several algorithms in this family but here we focus on Gibbs sampling.

Gibbs sampling [7] considers each variable (or group of variables, in this case the algorithm goes under the name of *Blocked* Gibbs sampling) and then samples from its conditional distribution (or joint conditional distribution) keeping all the other variables fixed. Several approaches to perform MCMC inference over probabilistic logic programs have been presented in the literature, such as [2,4]. Here we consider the implementation proposed in [4] which is based on a dynamic list of samples that is iteratively modified using prolog assert and retract. For clarity, the main part of the algorithm is repeated here in Alg. 1. Briefly, a list of random choices is maintained in memory using Prolog asserts. Function *SampleCycle* keeps querying the evidence until the value true is obtained. Then, a number (block) of random choices are removed from the list using function *RemoveSamples*. Finally, the query is called and, if it is successful, the counter of the successes is incremented by one. The probability of the query is computed as the number of successes over the number of samples.

Algorithm 1 Function Gibbs: Gibbs MCMC algorithm

```
1: function GibbsCycle(query, evidence, Samples, block)
2:     Succ ← 0
3:     for n ← 1 → Samples do
4:         Save a copy of samples C
5:         SampleCycle(evidence)                      ▷ samples the evidence
6:         Delete the copy of samples C
7:         RemoveSamples(block)          ▷ retracts samples from the top of the list
8:         Call query             ▷ new samples are asserted at the bottom of the list
9:         if query succeeds then
10:            Succ ← Succ + 1
11:        end if
12:        Sample a value for deleted samples
13:    end for
14:    return Succ/Samples
15: end function
```

4 Experiments

To deeply analyze the performance of Gibbs sampling for PLPs, we tested the algorithm on eight different datasets. For each experiment, we plotted three graphs: one to track the performance in terms of number of samples required to convergence, one to track the execution time and one to track the population standard deviation of the samples. For all the experiments the code is available online at the indicated URLs. All the experiments were conducted on a cluster[3] with Intel® Xeon® E5-2630v3 running at 2.40 GHz. The results are averages of 10 runs. Execution times are computed using the SWI-Prolog built-in predicate `statistics/2` with the keyword `walltime`. The probability of the evidence is computed using the cplint predicate `mc_sample/3` with 10^6 samples. The predicate samples the query a certain number of times and returns the probability of success. The number of discarded samples can be set with the option `mix/1` while the block size with `block/1`. Here we set mix to 100 and block variable between 1 and 5. The experiments are described below:

- Arithm[4]: the program represents a random arithmetic function. The goal is to compute the conditional probability of the value of the function given that a couple of input output was observed. A characteristic of this program is that it has an infinite number of explanations. The probability of evidence is 0.05. Fig. 1 shows that with a relatively small number of samples, the probability computed with Gibbs sampling with block set to 1 oscillates between 0.1 and 0.25, as also confirmed by the large variation in the standard deviation (Fig. 9 Left). For the other values, the oscillation is smaller but still present.
- Diabetes[5]: the program models the probability of insurgence of diabetes given that some genetic factors are observed. This is an example of probabilistic constraint logic program [9], i.e., a logic program that also contains constraints. In detail, the diabetes predisposition of a person influences the probability of diabetes mellitus type 2. The level of the glucose is modelled with two normal distribution with different mean and variance, related to the fact that a person has diabetes or not. The level of hemoglobin linked to sugar depends linearly from the level of glucose plus some noise. We observe that the hemoglobin is greater than a certain threshold (evidence probability 0.1417) and we want to know the probability of diabetes type 2 a priori and given the evidence. Fig. 2 shows that smaller sizes of blocks drive to lower accuracy in the probability computation and greater standard deviation (Fig. 9 Right). Block set to 5 makes the variation of the standard deviation almost negligible but at the cost of larger execution time. It is interesting to observe that block set to 3 and 5 seems to underestimate the probability.

[3] http://www.fe.infn.it/coka/doku.php?id=start
[4] http://cplint.eu/e/arithm.pl
[5] http://cplint.eu/e/diabetes.swinb

- Graph[6]: in the following experiment we want to test the accuracy of Gibbs sampling on a Barabási Albert preferential attachment model. Given a graph with initially connected nodes, new nodes are added to the graph. The probability that these new nodes are connected to other nodes is proportional to the number of edges that the already existent nodes have. We generated the graph with the python library `networkx`[7] using the function `barabasi_albert_graph(40,10)`. For all the edges we set a probability of 0.1. We compute the probability that two nodes are connected given that a portion of the path has already been observed (probability of the evidence 0.42). The probability of this query can also be computed using exact inference algorithms. However, when the size of the graph increases, exact inference may be too expensive in terms of execution time. Fig. 3 shows that, as for the previous experiments, smaller values of blocks leads to very different probability values. In particular, with block set to 1, there is a gap of 0.2 between some values. Execution times for all the five block settings are equivalent. Standard deviation of the samples (Fig. 10 Left) decreases as the block number increases. With block set to 3, 4 and 5, the computed values are almost the same.
- Hidden Markov Model[8] (HMM). The program represents a model of DNA sequences using an HMM [5]. In detail, the model has three states (`q1`, `q2` and `end`) and four output symbols, (`a`, `c`, `g`, and `t`), that represents the four nucleotides. We want to compute the probability of the sequence `[a,c]` given that the letter `a` has been emitted in state `q1`. The evidence has probability 0.25. Fig. 4 shows that, even with a relatively small number of samples, all the five block values performs well and the probability presents small fluctuations. However, when the value of block is set to one, the algorithm seems to take more time to stabilize, as described also by the standard deviation plot (Fig. 10 Right). It is interesting to note that the execution time for blocked Gibbs with value of block set to 4 and 5 are very similar.
- Latent Dirichlet Allocation (LDA)[9] for natural language processing. LDA is commonly used in text analysis with the goal to identify the topic of a text by analyzing the words in it. In the following example, we consider the first 10 words of the document and we set the number of topics to 2. We compute the probability that the document associates the first topic to the first word, observing the type of the first word (the probability of the evidence is 0.10). This is a hybrid program, since it contains both discrete and continuous variables. Fig. 5 shows that the values of probability for all five settings of block do not stabilize even after 10^4 samples, as also illustrated by the standard deviation plot (Fig. 11 Left).
- NBalls[10]. This program models an urn that contains n balls, where n is a random variable. Each ball is characterized by color, material and size, with

[6] http://cplint.eu/e/barabasiGraph.pl
[7] https://networkx.github.io/documentation/networkx-1.9.1/index.html
[8] http://cplint.eu/e/hmm.pl
[9] http://cplint.eu/e/lda.swinb
[10] http://cplint.eu/e/nballs.pl

known distributions. We want to know the probability that the first drawn ball is made of wood given that its color is black. The probability of the evidence is 0.38. Fig. 6 shows that the probability completely stabilizes after 10^4 samples only for large values of block. Block set to 1 and 3 seems to be the values with more instability. However, the standard deviation of the samples decreases for all the block values, as the number of samples increase (Fig.11 Right). Execution times are comparable.

- Prefix parser[11]. This program models a prefix parser for probabilistic context free grammars [16]. The program computes the probability that a certain string is a prefix of a string generated by the grammar. In the code we consider a grammar composed by two words `a` and `b` and we observe that the first emitted letter is `a` (probability 0.5). We want to compute the probability that the emitted string is `[a,b,a]`. The conditional probability has a small value (Fig. 7, Fig.12 Left) but all the five values of block seems to perform well. However, the execution time is the greatest among all the experiments.
- Stochastic Logic Program[12]: the program defines a probability distribution over sentences. A feature of this program is that there is no stochastic memoization, i.e., repeated choices are independent. Moreover, rules with the same head have probabilities that sum up to one, and are mutually exclusive. In the experiment, we want to know the probability that three particular word are sampled given that the first one has been observed (probability 0.006). Fig. 8 shows that after a few thousands samples the probability starts to stabilize for all the five block values. As expected, the standard deviation reduces as the number of samples increases (Fig. 12 Right). The execution time for block set to 3 is, by far, the greatest of all five.

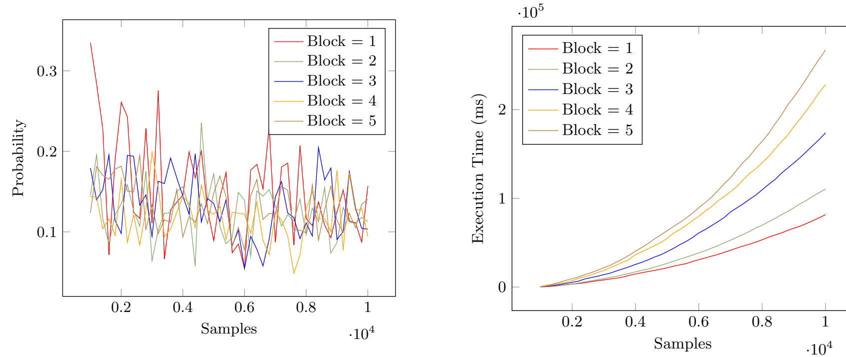

Fig. 1. Results for the arithm experiment.

[11] http://cplint.eu/e/prefix.pl
[12] http://cplint.eu/e/slp_pdcg.pl

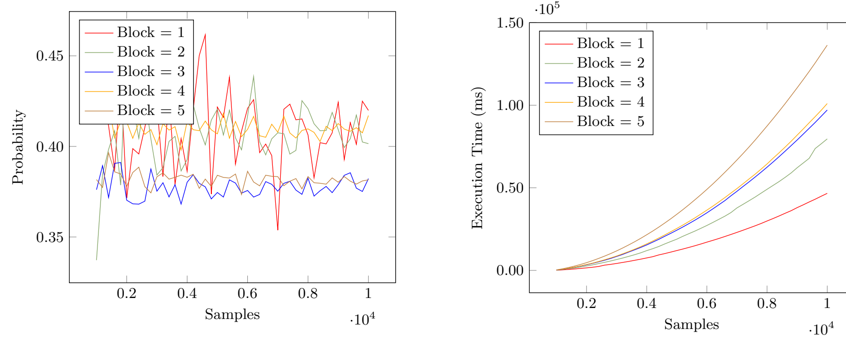

Fig. 2. Results for the diabetes experiment.

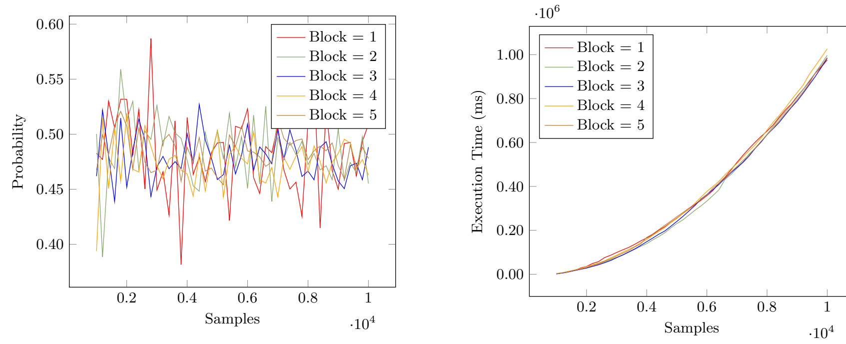

Fig. 3. Results for the graph experiment.

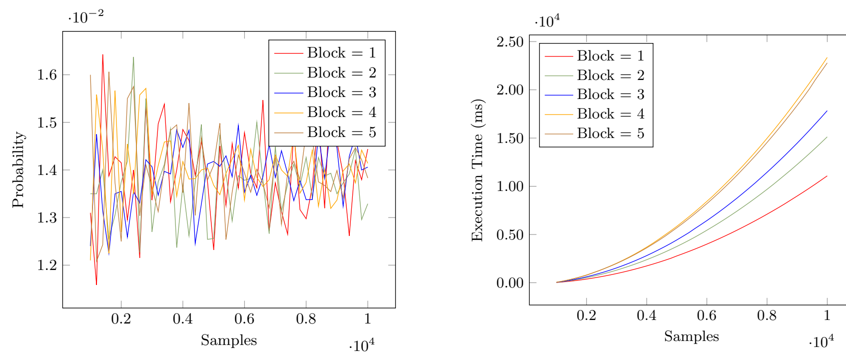

Fig. 4. Results for the HMM experiment.

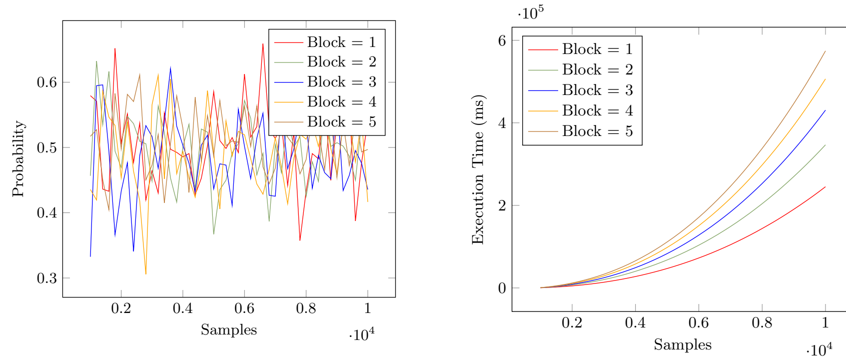

Fig. 5. Results fot the LDA experiment.

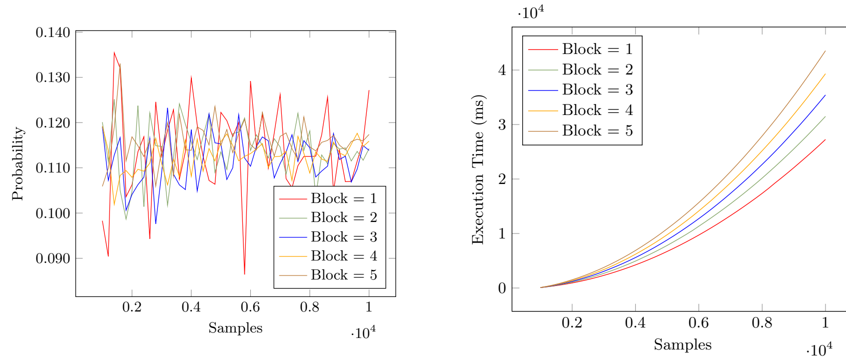

Fig. 6. Results for the nballs experiment.

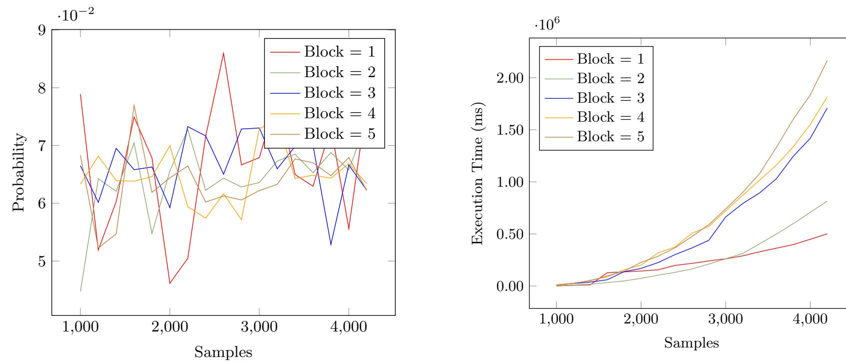

Fig. 7. Results for prefix parser experiment.

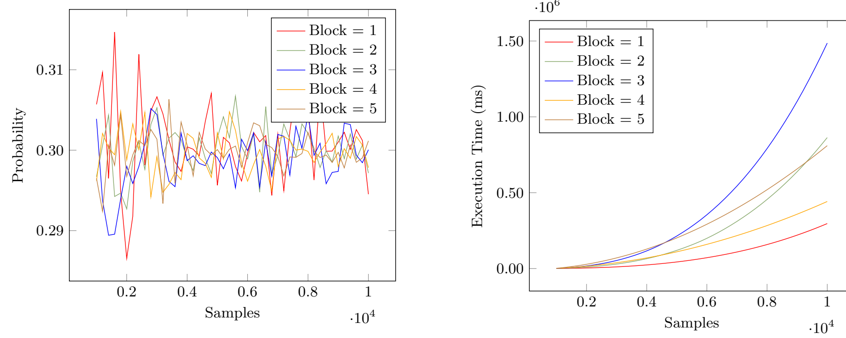

Fig. 8. Results for the stochastic logic program experiment.

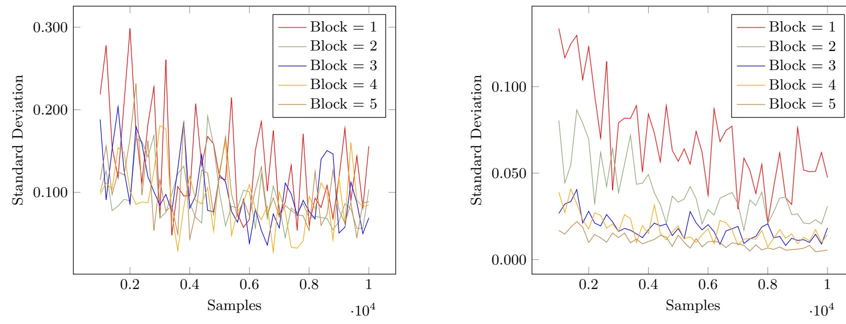

Fig. 9. Standard Deviation for arithm (left) and diabetes (right) experiments.

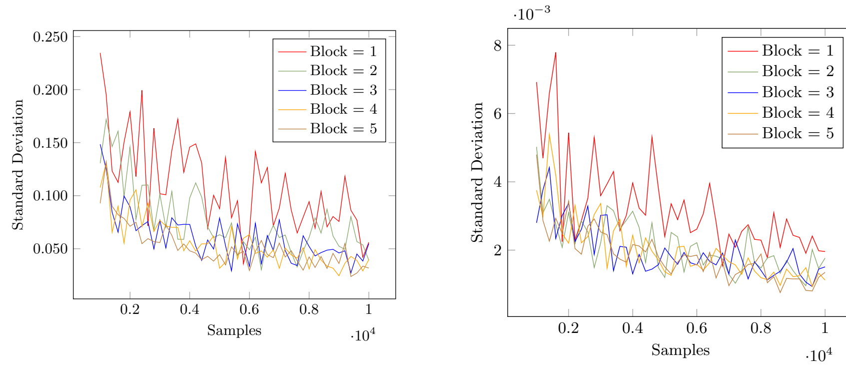

Fig. 10. Standard Deviation for graph (left) and HMM (right) experiments.

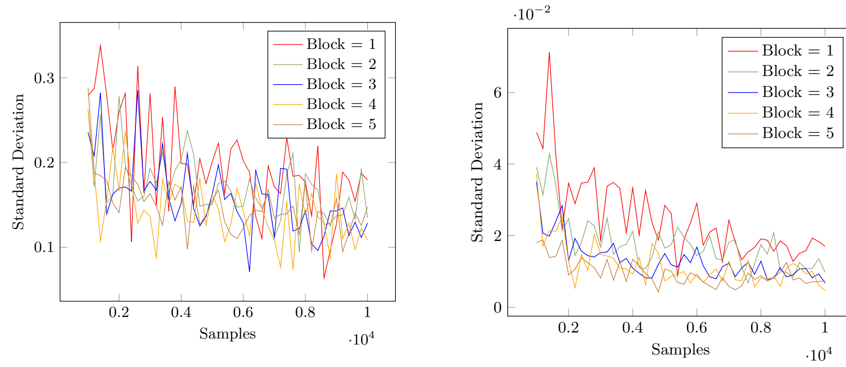

Fig. 11. Standard Deviation for LDA (left) and nballs (right) experiments.

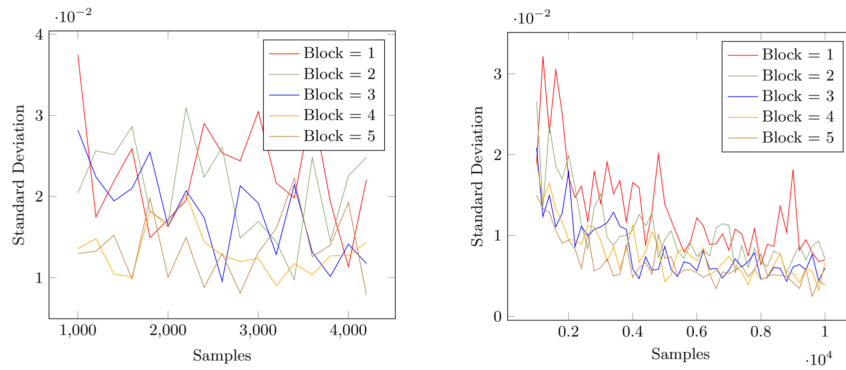

Fig. 12. Standard Deviation for prefix parser (left) and stochastic logic program (right) experiments.

5 Conclusions

In this paper we tested Gibbs sampling for probabilistic logic programs on eight different datasets. For each dataset we tracked execution time, computed probability and standard deviation of the samples. We used the *cplint* framework and the code for all the experiments can be analyzed trough a web interface accessible at cplint.eu. Empirical results shows that, when the value of block increases, the probability and the standard deviation seems to stabilize with a smaller number of samples. However, the execution time increases as well with an increment of the block value. As a future work, we plan to apply approximate reasoning also to abduction [6].

References

1. Alberti, M., Cota, G., Riguzzi, F., Zese, R.: Probabilistic logical inference on the web. In: Adorni, G., Cagnoni, S., Gori, M., Maratea, M. (eds.) AI*IA 2016. LNCS, vol. 10037, pp. 351–363. Springer International Publishing (2016)
2. Angelopoulos, N., Cussens, J.: Distributional logic programming for bayesian knowledge representation. Int. J. Approx. Reasoning 80(C), 52–66 (Jan 2017), https://doi.org/10.1016/j.ijar.2016.08.004
3. Azzolini, D., Riguzzi, F., Lamma, E.: Studying transaction fees in the bitcoin blockchain with probabilistic logic programming. Information 10(11), 335 (2019)
4. Azzolini, D., Riguzzi, F., Lamma, E., Masotti, F.: A comparison of MCMC sampling for probabilistic logic programming. In: Alviano, M., Greco, G., Scarcello, F. (eds.) Proceedings of the 18th Conference of the Italian Association for Artificial Intelligence (AI*IA2019), Rende, Italy 19-22 November 2019. Lecture Notes in Computer Science, Springer, Heidelberg, Germany (2019)
5. Christiansen, H., Gallagher, J.P.: Non-discriminating arguments and their uses. In: Logic Programming, 25th International Conference, ICLP 2009, Pasadena, CA, USA, July 14-17, 2009. Proceedings. Lecture Notes in Computer Science, vol. 5649, pp. 55–69. Springer (2009)
6. Gavanelli, M., Lamma, E., Riguzzi, F., Bellodi, E., Zese, R., Cota, G.: An abductive framework for datalog± ontologies. In: Vos, M.D., Eiter, T., Lierler, Y., Toni, F. (eds.) Technical Communications of the 31st International Conference on Logic Programming (ICLP 2015). CEUR-WS, vol. 1433. CEUR-WS.org (2015)
7. Geman, S., Geman, D.: Stochastic relaxation, gibbs distributions, and the bayesian restoration of images. In: Readings in computer vision, pp. 564–584. Elsevier (1987)
8. Koller, D., Friedman, N.: Probabilistic Graphical Models: Principles and Techniques. Adaptive computation and machine learning, MIT Press, Cambridge, MA (2009)
9. Michels, S., Hommersom, A., Lucas, P.J.F., Velikova, M.: A new probabilistic constraint logic programming language based on a generalised distribution semantics. Artif. Intell. 228, 1–44 (2015)
10. Nguembang Fadja, A., Riguzzi, F.: Probabilistic logic programming in action. In: Holzinger, A., Goebel, R., Ferri, M., Palade, V. (eds.) Towards Integrative Machine Learning and Knowledge Extraction, LNCS, vol. 10344. Springer (2017)
11. Riguzzi, F.: MCINTYRE: A Monte Carlo system for probabilistic logic programming. Fund. Inform. 124(4), 521–541 (2013)

12. Riguzzi, F.: The distribution semantics for normal programs with function symbols. Int. J. Approx. Reason. 77, 1–19 (2016)
13. Riguzzi, F.: Foundations of Probabilistic Logic Programming. River Publishers, Gistrup,Denmark (2018), http://www.riverpublishers.com/book_details.php?book_id=660
14. Riguzzi, F., Bellodi, E., Lamma, E., Zese, R., Cota, G.: Probabilistic logic programming on the web. Softw.-Pract. Exper. 46(10), 1381–1396 (10 2016)
15. Sato, T.: A statistical learning method for logic programs with distribution semantics. In: Sterling, L. (ed.) ICLP 1995. pp. 715–729. MIT Press (1995)
16. Sato, T., Meyer, P.: Tabling for infinite probability computation. In: Dovier, A., Costa, V.S. (eds.) ICLP TC 2012. LIPIcs, vol. 17, pp. 348–358. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2012)
17. Vennekens, J., Verbaeten, S., Bruynooghe, M.: Logic programs with annotated disjunctions. In: Demoen, B., Lifschitz, V. (eds.) ICLP 2004. LNCS, vol. 3131, pp. 431–445. Springer (2004)