Abstract

Probabilistic Sentential Decision Diagrams (PSDDs) are effective tools for combining uncertain knowledge in the form of (learned) probabilities and certain knowledge in the form of logical constraints. Despite some promising recent advances in the topic, very little attention has been given to the problem of effectively learning PSDDs from data and logical constraints in large domains. In this paper, we show that a simple strategy of sampling and averaging PSDDs leads to state-of-the-art performance in many tasks. We overcome some of the issues with previous methods by employing a top-down generation of circuits from a logic formula represented as a BDD. We discuss how to locally grow the circuit while achieving a good trade-off between complexity and goodness-of-fit of the resulting model. Generalization error is further decreased by aggregating sampled circuits through an ensemble of models. Experiments with various domains show that the approach efficiently learns good models even in very low data regimes, while remaining competitive for large sample sizes.

1 INTRODUCTION

Probabilistic Circuits (PCs) are generative models with neural network-like semantics capable of tractably answering several advanced probabilistic queries. Conceptually, PCs unify a wide range of tractable probabilistic models such as arithmetic circuits [Darwiche, 2003], sum-product networks [Poon and Domingos, 2011], (mixtures of) cutset networks [Rahman et al., 2014], and generative forests [Correia et al., 2020].

Probabilistic Sentential Decision Diagrams (PSDDs) are a particularly interesting subclass of PCs, as tractability covers a broader spectrum of exact queries in such models [Kisa et al., 2014, Bekker et al., 2015, Shen et al., 2016, Mattei et al., 2020, Vergari et al., 2021]. Also, the parameters of PSDDs have a clear probabilistic interpretation, which allows for closed-formula parameter learning and the injection of domain knowledge in the form of propositional logic formulae expressed in the network structure [Darwiche, 2011, Kisa et al., 2014]. This allows PSDDs to be efficiently learned from a combination of data and logic constraints, increasing sample efficiency and allowing the easy representation of combinatorial objects such as hierarchies, rankings, routes, etc [Choi et al., 2016, 2015, 2017, Shen et al., 2017].

Most existing approaches to learning PSDDs from intricate constraints and data are limited to specific logic formulae [Choi et al., 2015, 2017, Shen et al., 2017]. To our knowledge, the only existing algorithms that take arbitrary constraints and data are LEARNPSDD [Liang et al., 2017] and STRUDEL [Dang et al., 2020]. Both are centered on the idea of iteratively applying structural transformations that preserve the logical constraints represented in the incumbent model and increase goodness-of-fit or some proxy measure. While LEARNPSDD performs a costly local search requiring several evaluations through the whole circuit at every iteration, STRUDEL makes use of fast heuristic local searches to achieve similar performance at much lower computational cost. The question of how to initialize the structure search is left mostly unaddressed by Liang et al. [2017], while Dang et al. [2020] suggest compiling a circuit from a Chow-Liu Tree learned from data that ignores logical constraints.

Logical restrictions can be incorporated in learning by compiling a CNF formula into an initial logic circuit, usually chosen to minimize size [Choi and Darwiche, 2013, Oztok and Darwiche, 2015]. The compiled circuit is canonical, in that it is the unique representation of the formula for a given partial ordering of the variables, and is devoid of any probabilistic meaning. Such an approach presents two major shortcomings. First, some logical constraints (e.g. cardinality constraints) that can be efficiently represented as PCs have intractable CNF representation [Nishino et al., 2020].
We first review basic concepts of PSDDs, and fix some notation. Random variables are written in upper case (e.g. $X$) and their values in lower case (e.g. $x$). We identify propositional variables with 0/1-valued random variables, and use them interchangeably. Sets of variables and their joint values are written in boldface (e.g. $X$, $x$). Given a Boolean formula $f$, we write $\langle f \rangle$ to denote its semantics, that is, the Boolean function represented by $f$. For Boolean formulas $f$ and $g$, we write $f \equiv g$ if they are logically equivalent, that is, if $\langle f \rangle = \langle g \rangle$; we abuse notation and write $\phi \equiv f$ to indicate that $\phi = \langle f \rangle$ for a Boolean function $\phi$. The scope $\text{Sc}(f)$ of a formula $f$ is the set of variables that appear in it. The partitions of a Boolean formula are an important concept to understand PSDDs [Darwiche, 2011]:

**Definition 1.** Let $\phi(x, y)$ be a Boolean function over disjoint sets of variables $X$ and $Y$, and $\mathcal{D} = \{(p_i, s_i)\}_{i=1}^{k}$ be a set of tuples where $p_i$ and $s_i$ are formulae over $X$ and $Y$, respectively, satisfying $p_i \land p_j \equiv \bot$ for each $i \neq j$ and $\lor_{i=1}^{k} p_i \equiv \top$. We say that $\mathcal{D}$ is an $(X, Y)$-partition of $\phi$ iff $\phi \equiv \lor_{i=1}^{k} (p_i \land s_i)$. Each $p_i$ and $s_i$ is called a prime and sub, respectively; together they form an element of the partition.

Each prime and sub can be further decomposed into new partitions, recursively, until they contain only literals. This process generates a logic circuit as the one in Figure 1c (if one disregards the weights at the edges). In the figure we draw the prime $p_i$ of an element as the left child (input) of an AND gate, and the sub as its right child. Such a recursive decomposition is often guided by a vtree, which is a rooted binary tree whose leaves are the variables in both the data and $\phi$. The variables appearing in a vtree with root $v$ are denoted $\text{Sc}(v)$. We distinguish the children of a node $v$ of a vtree into a left child, denoted as $v^+$, and a right child, denoted as $v^-$. Intuitively, the left child contains the variables in the $X$ part of an $(X, Y)$-partition, while the right child contains the variables in $Y$. A vtree is said to be right-linear (resp., left-linear) when every left (resp., right) child is a leaf. A vtree defines a total variable ordering as a left-right traversal. Figure 1b shows a right-linear vtree for the circuit in Figure 1c; the vtree defines the total order $(A, B, C)$.

PSDDs represent probability distributions subject to logical constraints by associating weights to the logic circuit of a recursive decomposition of a formula [Kisa et al., 2014].

**Definition 2 (PSDD).** Fix a vtree $v$. A PSDD is either (i) an indicator function of a literal $[X]$ or $[\neg X]$ where $X$ is a variable associated to a leaf of $v$, or (ii) a convex combination $\sum_{i=1}^{k} \theta_i P_i(X) P_s(y)$ where $P_i$ and $P_s$ are PSDDs over variables $X$ and $Y$, respectively, which are also the variables in the left and right children of an inner node of $v$. We require that $\sum P_i(x) = P_s(x)$ for each assignment $x$ of $X$, and that $\theta_i \geq 0$ if $\max_x P_i(y) = 0$.

The notation $[\phi]$ denotes the Iverson Bracket, which is a function that returns 1 if $\phi$ is true and 0 otherwise.

---

![Figure 1](image-url)
we obtain \( P(\text{product for AND gates and convex combination for OR values. Applying this procedure for the PSDD in Figure 1c value resulting of the corresponding function of the children e variables by bottom-up propagation: each leaf is assigned } \phi \text{ and then finds a PSDD structure that approximates formula, our method generates a vtree uniformly at random, ing PSDDs by sampling and averaging. Given a Boolean In this section, we describe a procedure for effectively learn-}

3 SAMPLING PSDDS

In this section, we describe a procedure for effectively learning PSDDs by sampling and averaging. Given a Boolean formula, our method generates a vtree uniformly at random, and then finds a PSDD structure that approximates \( \phi \) as much as possible while maintaining the circuit size below some given threshold. Approximating the logical constraints allows us to trade consistency for circuit complexity, and scale to large domains.

Consider a Boolean formula \( \phi \) which we want to decompose as an \( (X, Y) \)-partition according to some vtree. We therefore must produce elements such that their primes are mutually exclusive, exhaustive and their disjunction is valid. To simplify the problem, we consider only partitions where the primes are conjunctions of literals, as in the example in Figure 1. In this particular case, generating an element consists in producing a conjunction of literals to serve as the prime \( p \), with the respective sub obtained as the restriction of \( \phi \) by the single assignment of \( X \) consistent with \( p \). We denote the latter operation as \( \phi|_p \). A naïve implementation of that strategy, however, scales poorly, as the number of elements in a partition grows exponentially in the cardinality of \( X \). To counter this blow up, we instead sample a constant number of mutually exclusive and exhaustive primes, leading to an upper approximation of the Boolean formula.

To motivate the approach taken, and illustrate the difficulties it overcomes, consider generating a partition of the formula \( \phi(A, B, C, D) = (A \land \neg B \land \neg D) \lor (B \land \neg C \land D) \) according to the root node \( v = 1 \) of the vtree in Figure 2b, using at most 3 elements whose primes are conjunctions of literals. An exact decomposition requires \( 2^3 \) elements whose primes are all combinations of positive and negative literals of the variables \( \{A, B, C\} \). We can reduce that number down to our limit of 3 elements by grouping primes that share literals. For example, take the partially constructed circuit in Figure 2c. The prime of \( e_1 \) is obtained as the disjunction...
of the primes $A \land B \land C$ and $A \land B \land \neg C$. Similarly, the
prime of $e_3$ is the disjunction of all primes that contain $\neg A$. The subs are obtained by the restriction of $\phi$ by the
corresponding prime. As the example shows, the result is not a proper $(\{A, B, C\}, \{D, E\})$-partition, as the subs of
e_1 and e_3 both contain the variable C in the scope. Further
note that, despite $E$ not appearing in $\text{Sc}(\phi)$ (i.e. there is no
logical restriction on $E$), it may nonetheless retain some
probabilistic influence derived from the data, and as such
may not be removed from the circuit.

To efficiently decompose a formula, we resort to a weaker
definition of a partition that relaxes the logical constraints.

**Definition 3 (Partial partition).** Let $\phi(x, y)$ be a Boolean
function over disjoint sets of variables $X$ and $Y$, and $\mathcal{D} = \{(p_i, s_i)\}_{i=1}^k$ be a set of tuples where $p_i$ and $s_i$ are formulae
over $X$ and $Y$, respectively, with $p_i \land p_j \equiv \bot$ for $i \neq j$ and
$\lor_i^k p_i \equiv \top$. We say that $\mathcal{D}$ is a partial partition of $\phi$ if
\[
\bigvee_{i=1}^k (p_i \land s_i) \geq \phi,
\]
where the inequality is taken coordinate-wise.

Partial decompositions are similar in spirit to oblivious bounds in probabilistic databases [Gatterbauer and Suciu 2014], where a probability computation is made more tractable by relaxing a formula in such a way that the approximate probabilities provide an upper bound which does not depend on the actual probability.

The set of conjunctive primes in a partial partition can be
represented as a binary decision tree where each node is
labeled as a variable and the outgoing edges denote positive
and negative literals over that variable, as in the example
Figure 2c (right). A path from the root to a leaf represents
all literals in a prime. We use this tree representation to
efficiently generate a partial $(\text{Sc}(v^+), \text{Sc}(v^-))$-partition
of a Boolean formula $\phi$ according to a vtree node $v$ with
at most $k$ primes as follows. First, randomly sample an
ordering $(X_1, \ldots, X_m)$ of the variables in $\text{Sc}(v^-)$. Start-
ing from the root node labeled as $X_1$, repeatedly expand a
leaf labeled $X_i$ with two children labeled as $X_{i+1}$ until the
number of leaves is between $k-1$ and $k$ (so that further
expanding a leaf would violate the bound on the number of
primes). When expanding a leaf, generate restrictions $\phi|_{X_i}$
and $\phi|_{\neg X_i}$, and associate them with the left and right children,
respectively. Now, it may happen that $\phi|_{X_i} \equiv \phi|_{\neg X_i}$.
In this case, relabel the node as $X_{i+1}$ and re-expand it with
children $X_{i+2}$. When the process terminates we have at
most $k$ conjunctive primes $p$ represented by paths in the
tree, associated with formulae $\phi|_p$. Now, because the primes
do not contain all variables in $\text{Sc}(v^-)$, those formulae are
not valid subs. To obtain valid subs, we apply the operation
Forget$(\psi, X) \equiv \psi|_X \land \neg \psi|_{\neg X}$ to each such formula
$\psi$ and each variable $X$ in $\text{Sc}(\psi) \cap \text{Sc}(v^-)$. Note that by
construction Forget$(\psi, X) \geq \psi$, and by extent any PSDD
constructed as such is a relaxation of its intended logic for-
ma. The described procedure is more formally visualized in
the pseudocode of Algorithm 1. Figure 2c displays an example
of a partial partition of the formula $\phi$ obtained by the algorithm using an ordering $A, B, C$ and 3 primes.

We generate a PSDD structure by repeatedly applying
**Algorithm 1** SAMPLEPARTIALPARTITION over the previously generated
primes and subs until there are only literals left. Alterna-
tively, we may apply a compression or merge operation in
order to penalize the circuit size. Let $\mathcal{D} = \{(p_i, s_i)\}_{i=1}^k$ be a
partial partition of a Boolean function $\phi$. If there are primes
$p_i$ and $p_j$ in $\mathcal{D}$ such that $s_i = s_j$, for $i \neq j$, then we can
obtain a smaller circuit representing the same logic formula
by replacing elements $(p_i, s_i)$ and $(p_j, s_j)$ with a new
element $(p_i \lor p_j, s_i)$. This operation is known as compression
[Darwiche 2011]. Although compression does not change the
formula of $D$, it does alter the PSDD’s underlying dis-
tribution. A merge operation, as opposed to compression,
preserves the structure of primes, and reduces the circuit by
connecting identical subs into a single sub-circuit. Figure 3
shows examples of compression and merging. Note that the
merge operation is the only one to generate gates with more
than a single parent.

**Algorithm 2** describes the SAMPLEPSDD algorithm for
sampling PSDD structures. Starting with the full, original
formula and root vtree node, the algorithm essentially grows

---

**Algorithm 1** SAMPLEPARTIALPARTITION

**Input** BDD $\phi$, vtree node $v$, number of primes $k$

**Output** A set of sampled elements

1: Define $E$ as an empty collection of sampled elements
2: Sample an ordering $X_1, \ldots, X_m$ of $\text{Sc}(v^-) \cap \text{Sc}(\phi)$
3: Let $Q$ be a queue initially containing $(\phi, 1, \{\})$
4: $j \leftarrow 1$
5: while $|E| < k$ do
6: Pop top item $(\psi, i, p)$ from $Q$
7: if $j \geq k$ or $i > m$ or $\psi \equiv \top$ then
8: Add $(p, \text{Forget}(\phi|_p, \text{Sc}(v^-)))$ to $E$
9: continue
10: $\alpha \leftarrow \psi|_{X_i}$, $\beta \leftarrow \psi|_{\neg X_i}$
11: if $\alpha \equiv \beta$ then enqueue $(\psi, i+1, p)$ in $Q$
12: else
13: if $\alpha \not\equiv \bot$ then push $(\alpha, i+1, p \land X_i)$ to $Q$
14: if $\beta \not\equiv \bot$ then push $(\beta, i+1, p \land \neg X_i)$ to $Q$
15: $j \leftarrow j + 1$
16: return $E$
Figure 3: Examples of local transformations for circuit complexity reduction during learning, where elements in bold on the left are either compressed (top) or merged (bottom), resulting in the (incomplete) circuit on the right.

Algorithm 2 SAMPLEPSDD

Input BDD \( \phi \), vtree node \( v \), number of primes \( k \)
Output A sampled PSDD structure

1: if \( \text{Sc}(v) = 1 \) then
2:  \( \text{if } \phi \text{ is a literal then return } \phi \text{ as a literal node} \)
3:  \( \text{else return a Bernoulli distribution from Sc}(v) \)
4:  else if \( \phi \equiv \top \) then
5:  \( \text{return a fully factorized circuit over Sc}(v) \)
6:  \( E \leftarrow \text{SAMPLEPARTIALPARTITION}(\phi, \text{Sc}(v^\top), k) \)
7:  Create an OR gate \( S \)
8:  Randomly compress elements in \( E \) with equal subs
9:  Randomly merge elements in \( E \) with equal subs
10: for each element \( (p, s) \in E \) do
11:  \( l \leftarrow \text{SAMPLEEXACTPSDD}(p, v^\top, k) \)
12:  \( r \leftarrow \text{SAMPLEPSDD}(s, v^\top, k) \)
13:  Add an AND gate with inputs \( l \) and \( r \) as a child of \( S \)
14: return \( S \)

We start with a toy problem. A seven-segment LED display is a(n exact) partition, the algorithm is able to construct a (incomplete) circuit on the right.

4 EXPERIMENTS

We evaluate the performance of SAMPLEPSDD in three different tasks that combine logical constraints and data against LEARNPSDD, STRUDEL, mixtures of STRUDELS [Dang et al., 2020], and LEARNSPN [Gens and Domingos, 2013]. On each instance, we sample a fixed number \( n \) of PSDD structures, learning their parameters through closed-form smoothed maximum-likelihood estimation [Kisa et al., 2014]. We then use these \( n \) PSDDs as a weighted mixture of models, optimizing weights through several strategies: (1) likelihood weighting (LLW), where each component’s weight is proportional to its train likelihood; (2) uniform weights, (3) Expectation-Maximization (EM), (4) stacking [Smyth and Wolpert, 1998], and (5) Bayesian Model Combination (BMC) [Monteith et al., 2011]. Due to the nature of SAMPLEPSDD, circuit sampling can easily be parallelized, significantly decreasing run time. In several instances, compiling an initial circuit from a CNF for LEARNPSDD was intractable, in which case we compiled a BDD into a PSDD to use in LEARNPSDD.

Experiments were run on an Intel i7-8700K 3.70 GHz machine with 12 cores and 64GB. We limited LEARNPSDD to at most 100 iterations, and STRUDEL to 100 iterations (we include runs with 1000 iterations in the supplementary material). STRUDEL circuits were generated using the Juice probabilistic circuits library [Dang et al., 2021], which was also used for the SAMPLEPSDD and LEARNPSDD implementations. For LEARNSPN, we used the PySPN library.

4.1 LED DISPLAY

We start with a toy problem. A seven-segment LED display consists of LED light segments which are separately turned on or off in order to represent a digit. Figure 5 (top) shows all digits represented by a seven-segment display. Each digit is associated with a local constraint on the values of each segment. We adapted the approach by Mattei et al. [2020], and generated a led dataset of faulty observations of the segments as follows. Each segment is represented by a pair of variables \((X_i, Y_i)\), where \(Y_i\) is the observable state of

https://gitlab.com/pgm-usp/pyspn
Figure 4: (a) Log-likelihoods for the unpixelized led, (b) led-pixels, (c) sushi 10-choose-5, (d) sushi ranking, and (e) dota datasets. (f) Mean average in seconds of each PSDD learning algorithm.
At most one single player in a match. We represent the domain by 2 groups of 113 Boolean variables \( C_{1}^{(i)} \) and \( C_{2}^{(i)} \), denoting whether the \( i \)-th character was selected by the first or second team, respectively. We then encode 113 choose 5 cardinality constraints on the selection of each team (i.e., \( \sum_{j} C_{j}^{(i)} = 5 \) for \( j = 1, 2 \)). Adding the constraint that no character can be selected by both teams \((-C_{1}^{(i)} \land C_{2}^{(i)})\) made the BDD representation of the formula intractable, and was ignored. Since the CNF representation of cardinality constraints is intractable, we used a PSDD compiled from the BDD to generate an initial circuit for \textsc{LearnPSDD} (as BDDs can efficiently encode such constraints [Eén and Sörensson, 2006]).

The plot in Figure 4e shows the log-likelihood of the tested approaches. Despite accurately encoding logical constraints, \textsc{LearnPSDD} initially obtains worse performance when compared to \textsc{SamplePSDD}, but quickly picks up, outperforming other models by a large margin. \textsc{SamplePSDD} ranks first for small data regimes, and is comparable to other algorithms (e.g. \textsc{StruDEL}) for larger training datasets. \textsc{LearnSPN} encountered problems scaling to more than 50k instances due to intensive memory usage.

We also compared methods on the \textit{sushi} dataset [Kamishima, 2003], using the setting proposed in Shen et al. [2017]. The data contains a collection of 5,000 rankings of 10 different types of sushi. For each ranking we create 10 Boolean variables denoting whether an item was ranked among the top 5, and ignore their relative position. The logic constraints represent the selection of 5 out of 10 items. We split the dataset into 3,500 instances for training and 1,500 for the test set and evaluated the log-likelihood on both tasks. The plot in Figure 4e shows the log-likelihood for this data. \textsc{LearnPSDD} obtains superior performance across some of the low sample sizes, but our approach was able to quickly pick up and tie with \textsc{LearnPSDD} when using the LLW, stacking and EM strategies.

4.3 PREFERENCE LEARNING

We also evaluated the methods on the original task of ranking the items on the \textit{sushi} dataset. We adopt the same encoding and methodology as [Choi et al., 2015], where each ranking is encoded by a set of Boolean variables \( X_{ij} \) indicating whether the \( i \)-th item was ranked in the \( j \)-position. The test log-likelihood performance of the methods is shown in Figure 4d. The results are qualitatively similar to the previous experiments, with the added exception that \textsc{LearnPSDD} ranked first by a large margin compared to others.
4.4 PERFORMANCE AND SAMPLING BIAS

The approximation quality of SAMPLEPSDD is highly dependent on both the vtree and maximum number of primes. In this section, we compare the impact of both in terms of performance and circuit complexity. We assess performance by the log-likelihoods in the test set, as well as consistency with the original logical constraints. The latter is measured by randomly sampling 5,000 (possibly invalid) instances and evaluating whether the circuit correctly decides their satisfiability. A set of the top 100 sampled PSDDs (in terms of log-likelihood in the train set) are selected out of 500 circuits learned on the 10-choose-5 sushi dataset to compose the ensemble. Circuit complexity is estimated in terms of both time taken to sample all 500 circuits and graph size (i.e. number of nodes) of each individually generated PSDD.

It is quite clear that the structure of the vtree is strongly linked to the structure of a PSDD. This is even more so in the context of circuits of the SAMPLEPSDD form and given the need to approximate a logic formula. For instance, (near) right vtrees keep the number of primes fixed and require no approximation, while (near) left vtrees discard a large number of primes. In order to evaluate the effect of the type of vtree on the quality of sampled structures, we compared the performance of SAMPLEPSDD as we vary the bias towards generation of right-leaning vtrees.

Figure 6 shows the log-likelihood (top), consistency (middle) and circuit complexity (bottom) when varying the type of vtrees used for guiding the PSDD construction. The blue shaded area represents the interval of values for individual circuits. To verify consistency, we evaluate the PSDDs in terms of satisfiability of a given example. An ensemble returned a configuration as satisfiable if any of its models attributed some nonzero probability to it; and unsatisfiable if all models gave zero probability. This evaluation gives a lower bound to consistency, which means all models eventually unanimously agreed on satisfiability when vtree right bias $\geq 0.65$. Alternatively, since SAMPLEPSDD is a relaxation of the original formula, an upper bound on consistency could be achieved by evaluating whether any model within the ensemble gave a zero probability to the example. Interestingly, we note that the likelihood weighting strategy (LLW) dominates over other strategies on consistency. This is because LLW often degenerates to a few models, giving zero probability to lower scoring PSDDs, which means only a small subset of circuits decide on satisfiability, and thus a more relaxed model is less likely to disagree with the consensus. On the other hand, this does not translate to better data fitness on the general case, as we can clearly see from Figure 4.

Overall, Figure 6 shows that performance greatly increases

\textsuperscript{3}Given a parameter $p$, we grow a vtree in a top-down manner where at each node we independently assign each variable to the right child with probability $p$. 
as we move to more right-leaning vtrees, since we are more likely to take advantage of available prior knowledge. Figure 6 (bottom) shows that this comes at a cost to complexity, however, as more right-leaning vtrees mean smaller decompositions and fewer relaxations, resulting in larger circuits and higher learning times.

The maximum number of sampled primes \( k \) also plays a big role in providing good approximations. As previously mentioned, a sufficiently high \( k \) reduces partial partitions to exact partitions, although this is obviously generally not feasible for larger formulae or datasets. We evaluate how this impacts performance in the same manner as in the vtree experiment; although, in this setting, we considered uniformly random vtrees and only varied \( k \). Figure 7 shows a similar picture to the previous comparison: higher \( k \)’s translate to higher performance at a cost to circuit complexity.

Finally, we compare the time performance of PSDD learning algorithms on the same task. Figure 4f displays the mean time, in seconds, of each method as a function of training instances in the sushi ranking dataset. For SAMPLEPSDD, we measured the total time of learning 100 circuits in parallel. We observe that, although SAMPLEPSDD is much slower than STRUDEL, it is the result of 100 learned circuits. Meanwhile, LEARNPSDD is orders of magnitude slower compared to even SAMPLEPSDD, and outputs a single PSDD.

5 CONCLUSION

We proposed a new approach for learning PSDDs from logical constraints and data by a random top-down expansion on a propositional formula. Our method trades-off complexity and goodness-of-fit by learning a relaxation of the formula. We then leverage the diversity of samples by employing several different ensemble strategies. We empirically showed that this approach achieves state-of-the-art performance, often surpassing competitors when under very low data regimes. Finally, we reveal that PSDDs sampled from right leaning vtrees are better formula approximators and have increased log-likelihood performance, albeit at an increase of circuit complexity.

Acknowledgements

This work was supported by the Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq) grants # 133787/2019-2 and 304012/2019-0, and CAPES Finance Code 001. We thank the anonymous reviewers for their useful suggestions. We also thank the maintainers of the Juice package for quickly answering our questions.
References


