Learning Probabilistic Sentential Decision Diagrams Under Logic Constraints by Sampling and Averaging

Renato Lui Geh[1] **Denis Deratani Mauá**[1]

[1]Department of Computer Science, Institute of Mathematics and Statistics, University of São Paulo, Brazil

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.,

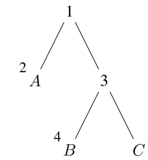
Accepted for the 37th Conference on Uncertainty in Artificial Intelligence (UAI 2021).

2016], or require the addition of auxiliary (latent) variables [Sinz, 2005]. Second, canonicity results in trivial representations for sub-circuits whose variables are not logically constrained, leading to unreasonable probabilistic independence assumptions. Besides some recent preliminary work on the sampling of PSDDs, little attention has been given to the generation of initial circuits from both logic formulae and data. Mattei et al. [2019] suggested a top-down approach to sampling PSDDs, yet no practical algorithm was fully formulated. Geh et al. [2020] recommended using a BDD to more efficiently represent the partial formulae required in the sampling of the circuit, however the generated circuit suffered from an exponential blow-up in size.

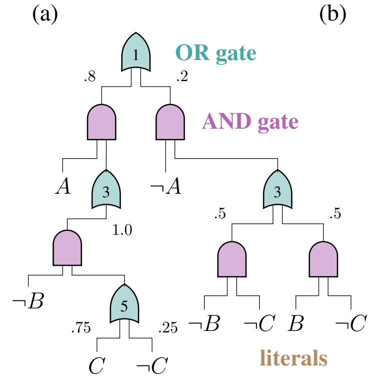
In this work, we show how the aforementioned problems can be mitigated by a simple learning method that is domain agnostic and scales to large amounts of data and intricate propositional formulae. The proposed method consists in sampling PSDDs in a way that balances model diversity, complexity and goodness-of-fit (Section 3). Our approach solves the intractability issues in the works of Mattei et al. [2019] and Geh et al. [2020] by partly relaxing logical constrains generated during the sampling. Experiments with previously used tasks show that, through simple ensemble strategies, the proposed method achieves competitive results against state-of-the-art methods (Section 4). We also present some conclusions and final remarks (Section 5). We start by reviewing the theory behind PSDDs (Section 2).

A	B	C	Pr
0	0	0	**0.1**
0	1	0	**0.1**
1	0	0	**0.2**
1	0	1	**0.6**

(a)



(b)



(c)

Figure 1: (a) A probability distribution whose support is given by $(A \to \neg B) \land (C \to A)$ and (c) its PSDD representation and (b) corresponding vtree.

2 BACKGROUND

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. $\mathbf{X}$, $\mathbf{x}$). 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 ϕ. The scope $\mathrm{Sc}(f)$ 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 $\phi(\mathbf{x}, \mathbf{y})$ be a Boolean function over disjoint sets of variables $\mathbf{X}$ and $\mathbf{Y}$, and $\mathcal{D} = \{(p_i, s_i)\}_{i=1}^k$ be a set of tuples where p_i and s_i are formulae over $\mathbf{X}$ and $\mathbf{Y}$, respectively, satisfying $p_i \land p_j \equiv \bot$ for each $i \neq j$ and $\vee_{i=1}^k p_i \equiv \top$. We say that $\mathcal{D}$ is an $(\mathbf{X}, \mathbf{Y})$-partition of ϕ iff $\phi \equiv \bigvee_{i=1}^k (p_i \land s_i)$. 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 ϕ. The variables appearing in a vtree with root v are denoted $\mathrm{Sc}(v)$. We distinguish the children of a node v of a vtree into a left child, denoted as $v^{\leftarrow}$, and a right child, denoted as $v^{\rightarrow}$. Intuitively, the left child contains the variables in the $\mathbf{X}$ part of an $(\mathbf{X}, \mathbf{Y})$-partition, while the right child contains the variables in $\mathbf{Y}$. 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,*[1] *or (ii) a convex combination $\sum_{i=1}^k \theta_i P_{p_i}(\mathbf{X}) P_{s_i}(\mathbf{Y})$ where P_{p_i} and P_{s_i} are PSDDs over variables $\mathbf{X}$ and $\mathbf{Y}$, respectively, which are also the variables in the left and right children of an inner node of v. We require that $\sum_i P_{p_i}(\mathbf{x}) = P_{p_j}(\mathbf{x})$ for some j for each assignment $\mathbf{x}$ of $\mathbf{X}$, and that $\theta_i = 0$ if $\max_{\mathbf{y}} P_{s_i}(\mathbf{y}) = 0$.*

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

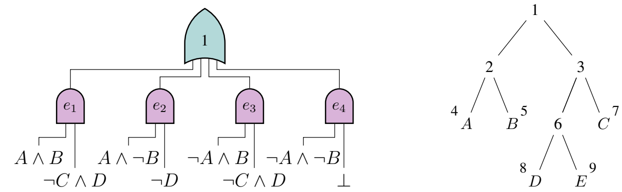
PSDDs are materialized as probabilistic circuits, that is, directed acyclic graphs whose leaves are univariate distributions and inner nodes represent convex combinations (OR gates) of products (AND gates) of their input. Logical constraints are embedded into a PSDD structure as the support of each sub-circuit. In fact, one can read off a logical expression/circuit of a PSDD by translating each convex combination $\sum_{i=1}^{k} \theta_i P_{p_i}(\mathbf{X}) P_{s_i}(\mathbf{Y})$ into a logical formula $\bigvee_{i=1}^{k} [\![P_{p_i}(\mathbf{X}) > 0]\!] \wedge [\![P_{s_i}(\mathbf{Y}) > 0]\!]$. This justifies naming convex combinations as OR gates and products as AND gates. Figure 1 shows an example of (a) a probability distribution whose support is the satisfying assignments of the formula $(A \to \neg B) \wedge (C \to A)$, (c) a PSDD representation of the same distribution and (b) the corresponding vtree. In the figure, the indicator functions are represented more clearly as literals; each θ_i of a convex combination is represented as a weighted edge connecting OR gates to AND gates. Note that OR gates can have more than one parent, hence the PSDD is not limited to tree structures as in the example.

By construction, the probabilistic circuit representation of a PSDD satisfies properties of *determinism* (each OR gate has at most one non-zero input when evaluated at a complete configuration of the literals), *smoothness* (all subcircuits of an OR gate have identical scope) and *structural decomposition* (AND gates have two children whose scopes are given by a vtree).

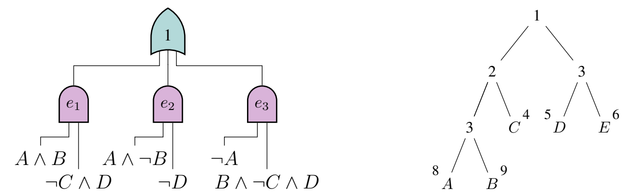
Once each leaf/literal of a PSDD has its value fixed, the corresponding probability value can be computed in linear time in the size of the circuit by visiting nodes from the leaves to the root and applying the function associated to each node (product for AND gates and convex combination for OR gates). One can also obtain in linear time the probability of some evidence $\mathbf{e}$ that specifies a partial assignment to the variables by bottom-up propagation: each leaf is assigned value 1 if the corresponding literal is satisfied by $\mathbf{e}$ or absent in $\mathbf{e}$ and 0 otherwise; and each inner node is assigned the value resulting of the corresponding function of the children values. Applying this procedure for the PSDD in Figure 1c we obtain $P(A = 1) = 0.8$. Conditional probabilities can thus be efficiently obtained as two computations of probability of evidences. Many other exact inferences can be performed efficiently in PSDDs by passing values through the graphical structure [Bekker et al., 2015, Mattei et al., 2020].

3 SAMPLING PSDDS

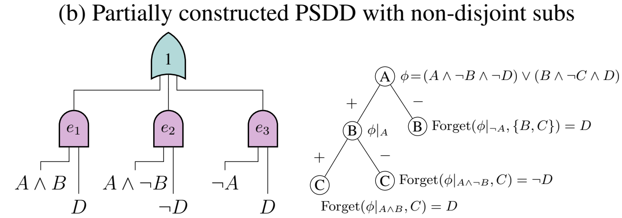
In this section, we describe a procedure for effectively learning PSDDs by sampling and averaging. Given a Boolean formula, our method generates a vtree uniformly at random, and then finds a PSDD structure that approximates ϕ as much as possible while maintaining the circuit size below some given threshold.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



(a) Exact partition



(b) Partially constructed PSDD with non-disjoint subs



(c) Partial partition

Figure 2: Exact, invalid and partial partitions.

Consider a Boolean formula ϕ which we want to decompose as an $(\mathbf{X}, \mathbf{Y})$-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 2a. 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 ϕ by the single assignment of $\mathbf{X}$ 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 $\mathbf{X}$. 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 $\phi(A, B, C, D) = (A \wedge \neg B \wedge \neg D) \vee (B \wedge \neg C \wedge D)$ 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 2b. The prime of e_1 is obtained as the disjunction of the primes $A \wedge B \wedge C$ and $A \wedge B \wedge \neg C$. Similarly, the prime of e_3 is the disjunction of all primes that contain $\neg A$. The subs are obtained by the restriction of ϕ 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 $\mathrm{Sc}(\phi)$ (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 $\phi(\mathbf{x}, \mathbf{y})$ be a Boolean function over disjoint sets of variables $\mathbf{X}$ and $\mathbf{Y}$, and $\mathcal{D} = \{(p_i, s_i)\}_{i=1}^{k}$ be a set of tuples where p_i and s_i are formulae over $\mathbf{X}$ and $\mathbf{Y}$, respectively, with $p_i \wedge p_j \equiv \bot$ for $i \neq j$ and $\vee_{i=1}^{k} p_i \equiv \top$. We say that $\mathcal{D}$ is a partial partition of ϕ if*

$$\left\langle \bigvee_{i=1}^{k} (p_i \wedge s_i) \right\rangle \geq \phi \,,$$

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 that tree representation to efficiently generate a partial $(\mathrm{Sc}(v^{\leftarrow}), \mathrm{Sc}(v^{\rightarrow}))$-partition of a Boolean formula ϕ 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 $\mathrm{Sc}(v^{\leftarrow})$. Starting 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 $\mathrm{Sc}(v^{\leftarrow})$, those formulae are not valid subs. To obtain valid subs, we apply the operation $\mathrm{Forget}(\psi, X) \equiv \psi|_X \vee \psi|_{\neg X}$ to each such formula ψ and each variable X in $\mathrm{Sc}(\psi) \cap \mathrm{Sc}(v^{\leftarrow})$. Note that by construction $\mathrm{Forget}(\psi, X) \geq \psi$, and by extent any PSDD constructed as such is a relaxation of its intended logic formula. 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 ϕ obtained by the algorithm using an ordering A, B, C and 3 primes. The Boolean functions ϕ in the algorithm are represented as Binary Decision Diagrams [Bryant, 1986], which allow for the efficient implementation of restriction and forget operations. This has the additional benefit of allowing efficient representations of logic constraints whose CNF/DNF representations are intractable.

Algorithm 1 SAMPLEPARTIALPARTITION

Input BDD ϕ, vtree node v, number of primes k
Output A set of sampled elements

Define $\mathbf{E}$ as an empty collection of sampled elements
Sample an ordering $X_1, \ldots, X_m$ of $\mathrm{Sc}(v^{\leftarrow}) \cap \mathrm{Sc}(\phi)$
Let $\mathbf{Q}$ be a queue initially containing $(\phi, 1, \{\})$
$j \leftarrow 1$ ▷ Counter of sampled elements
while $|\mathbf{E}| < k$ **do**
  Pop top item (ψ, i, p) from $\mathbf{Q}$
  if $j \geq k$ or $i > m$ or $\psi \equiv \top$ **then**
    Add $(p, \mathrm{Forget}(\phi|_p, \mathrm{Sc}(v^{\leftarrow})))$ to $\mathbf{E}$
    continue
  $\alpha \leftarrow \psi|_{X_i}$, $\beta \leftarrow \psi|_{\neg X_i}$
  if $\alpha \equiv \beta$ **then** enqueue $(\psi, i+1, p)$ in $\mathbf{Q}$
  else
    if $\alpha \not\equiv \bot$ **then** push $(\alpha, i+1, p \wedge X_i)$ to $\mathbf{Q}$
    if $\beta \not\equiv \bot$ **then** push $(\beta, i+1, p \wedge \neg X_i)$ to $\mathbf{Q}$
    $j \leftarrow j + 1$
return $\mathbf{E}$

We generate a PSDD structure by repeatedly applying SAMPLEPARTIALPARTITION over the previously generated primes and subs until there are only literals left. Alternatively, 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 ϕ. 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 $(p_i \vee p_j, s_i)$. This operation is known as *compression* [Darwiche, 2011]. Although compression does not change the formula of $\mathcal{D}$, it does alter the PSDD's underlying distribution. 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

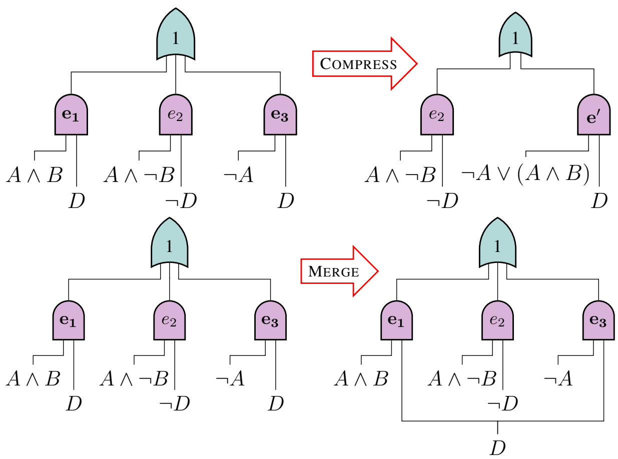


Figure 3: Examples of local transformations for circuit complexity reduction during learning, where elements in bold on the left are either compressed (top) or merged (bottom), resulting in the (incomplete) circuit on the right.

Algorithm 2 SamplePSDD

Input BDD ϕ, vtree node v, number of primes k
Output A sampled PSDD structure

```
1: if Sc(v) = 1 then
2:     if φ is a literal then return φ as a literal node
3:     else return a Bernoulli distribution from Sc(v)
4: else if φ ≡ ⊤ then
5:     return a fully factorized circuit over Sc(v)
6: E ← SamplePartialPartition(φ, Sc(v←), 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) ∈ E do
11:     l ← SampleExactPSDD(p, v←, k)
12:     r ← SamplePSDD(s, v→, k)
13:     Add an AND gate with inputs l and r as a child of S
14: return S
```

a PSDD circuit in a top-down fashion by recursively calling SamplePartialPartition. The recursion terminates when it is called on a vtree leaf node, in which case either a literal node is created if ϕ is a literal, or a Bernoulli distribution is learned if $\phi \equiv \top$ and $\mathrm{Sc}(v) = 1$. Note the special treatment for $\phi \equiv \top$ in Line 5. Since any partial partition of ϕ is a(n exact) partition, the algorithm is able to construct a random circuit from a tautology. Alternatively, we may associate some template circuit for such cases. For simplicity, we return a fully factorized distribution when recursing with a tautology in the pseudo-code.

To ensure prime mutual exclusivity and determinism, no relaxation is allowed at any subcircuit rooted at a prime. Consider Figure 2c as an example: suppose a recursive call on e_1's prime $A \wedge B$ relaxes it to B. This contradicts the partition definition, as B conflicts with e_3's prime $\neg A$ (because $B \wedge \neg A \not\equiv \bot$). To deal with this problem, we must make sure subcircuits rooted at primes only contain exact partitions. This is done with SampleExactPSDD (Line 9), which simply calls a SampleExactPartition equivalent during prime sampling. One can easily verify that, since we assume k as a bounded constant, these exact subcircuits will never suffer from an exponential blowup, as all of their subsequent exact partitions contain primes with at most $\lceil \log_2(k) \rceil$ variables and thus at most $2^{\lceil \log_2(k) \rceil}$ elements.

4 EXPERIMENTS

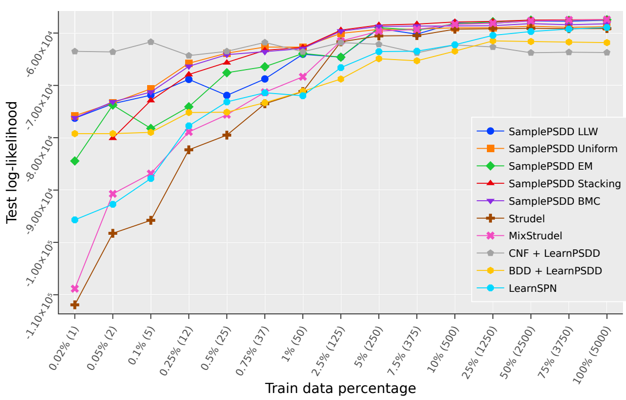
We evaluate the performance of SamplePSDD in three different tasks that combine logical constraints and data against LearnPSDD, Strudel, mixtures of Strudels [Dang et al., 2020], and LearnSPN [Gens and Domingos, 2013]. On each instance, we sample a fixed number n of PSDD structures, learning their parameters 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.[2]

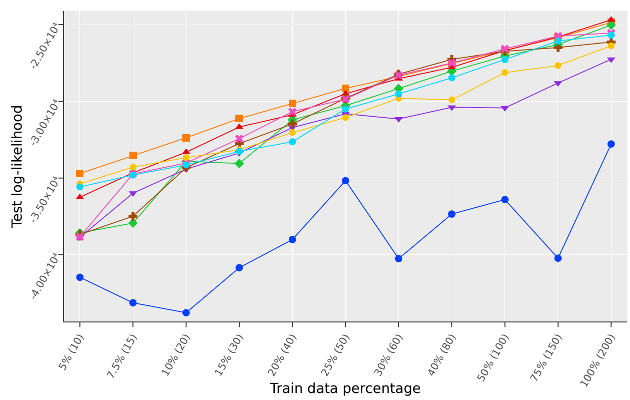
4.1 LED DISPLAY

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

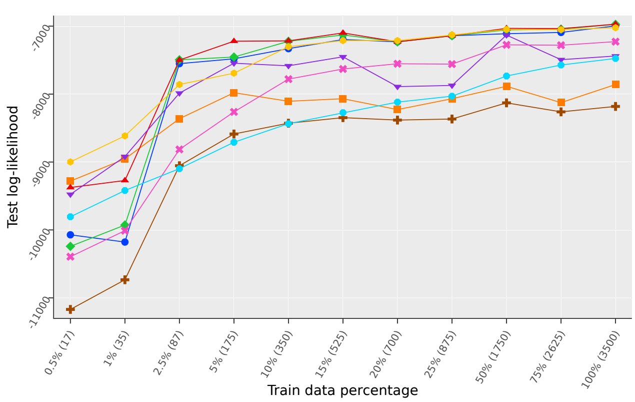
[2] https://gitlab.com/pgm-usp/pyspn



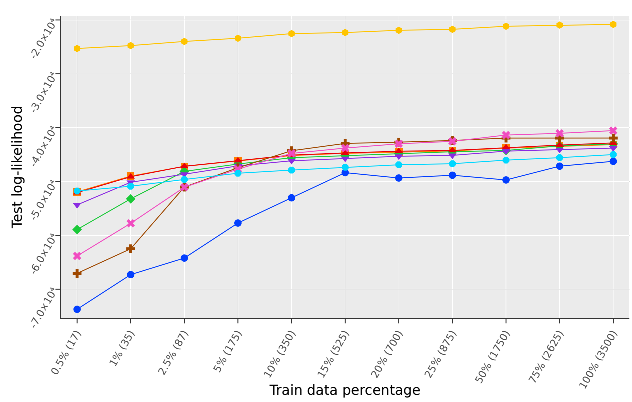
(a)



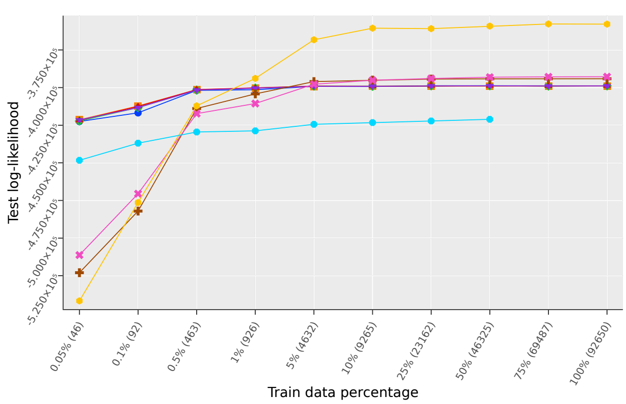
(b)



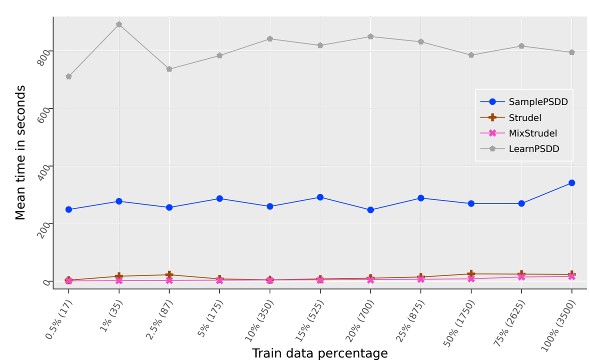
(c)



(d)



(e)



(f)

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.

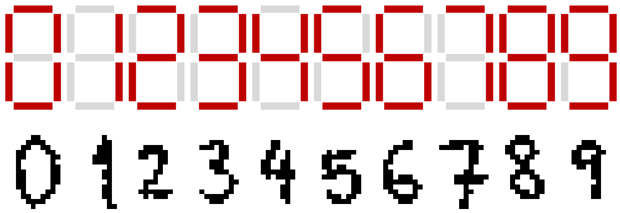


Figure 5: Digits represented by a LED display, with their image counterparts.

segment i (i.e. whether the segment is on or off) and X_i is the latent state of i. We randomly sampled a PSDD over variables X_i and Y_i whose support are the valid configurations of segments X_i representing the digits, and use that model to generate a dataset of 5000 training instances and 10000 test instances.

A more complex alternative configuration for the LED setting, `led-pixels`, is the interpretation of digits as images and segments as pixel regions. The segment constraints remain unchanged, but now pixels act as the latent variables. Figure 5 (bottom) shows the 10×15 black-and-white images for each digit. In this pixelized version, we do not explicitly describe, in the form of logical contraints, a one-to-one mapping of pixel regions as segments; instead, we visually identify key points where pixels are most often activated given a segment configuration. Let $\mathbf{R}_s$ be the pixel variables which are most often set to 1 when a segment s is on. We build a constraint for each segment: $\psi(s) = s \rightarrow \bigvee_{r \in \mathbf{R}_s} r$. We further recognize which pixels are always off given a valid digit segment configuration: $\phi(\mathbf{s}) = \left(\bigwedge_{p:p=0|\mathbf{s}} \neg p\right) \wedge \left(\bigwedge_{s \in \mathbf{s}} s\right)$. The full logic formula encoding the constraints is the conjunction of every possible ϕ and ψ.

Figure 4 (a) and (b) show how our approach fairs against competitors using different percentages of available training data on the unpixelized and pixelized versions of the dataset respectively. The labels on the x-axis indicate percentage and number of training instances. We sampled $n = 100$ circuits for both settings, with $k = 32$ and $k = 8$ for `led` and `led-pixels` respectively. Note how the use of logic constraints greatly improves performance even under extremely scarce data (with 1 or 2 datapoints!). Most of the SAMPLEPSDD approaches obtain the best sample efficiency among the methods that achieve best performance with the full dataset, and ranks amongst the best when data size is small.

4.2 CARDINALITY CONSTRAINTS

The `dota` dataset contains the results of 102,944 online matches of the Dota 2 videogame, made available at the UCI Repository. In this game, each team is composed of 5 players, with each player controlling a single character out of a pool of 113. Each character can be controlled by 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., $\sum_i C_j^{(i)} = 5$ for $j = 1, 2$). Adding the constraint that no character can be selected by both teams ($\neg(C_1^{(i)} \wedge C_2^{(i)})$) 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 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, LEARNPSDD initially obtains worse performance when compared to SAMPLEPSDD, but quickly picks up, outperforming other models by a large margin. SAMPLEPSDD ranks first for small data regimes, and is comparable to other algorithms (e.g. STRUDEL) for larger training datasets. LEARNSPN encountered problems scaling to more than 50k instances due to intensive memory usage.

We also compared methods on the `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 4c shows the log-likelihood for this data. LEARNPSDD obtains superior performance across some of the low sample sizes, but our approach was able to quickly pick up and tie with 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 `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 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.[3]

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 ≥ 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

[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.

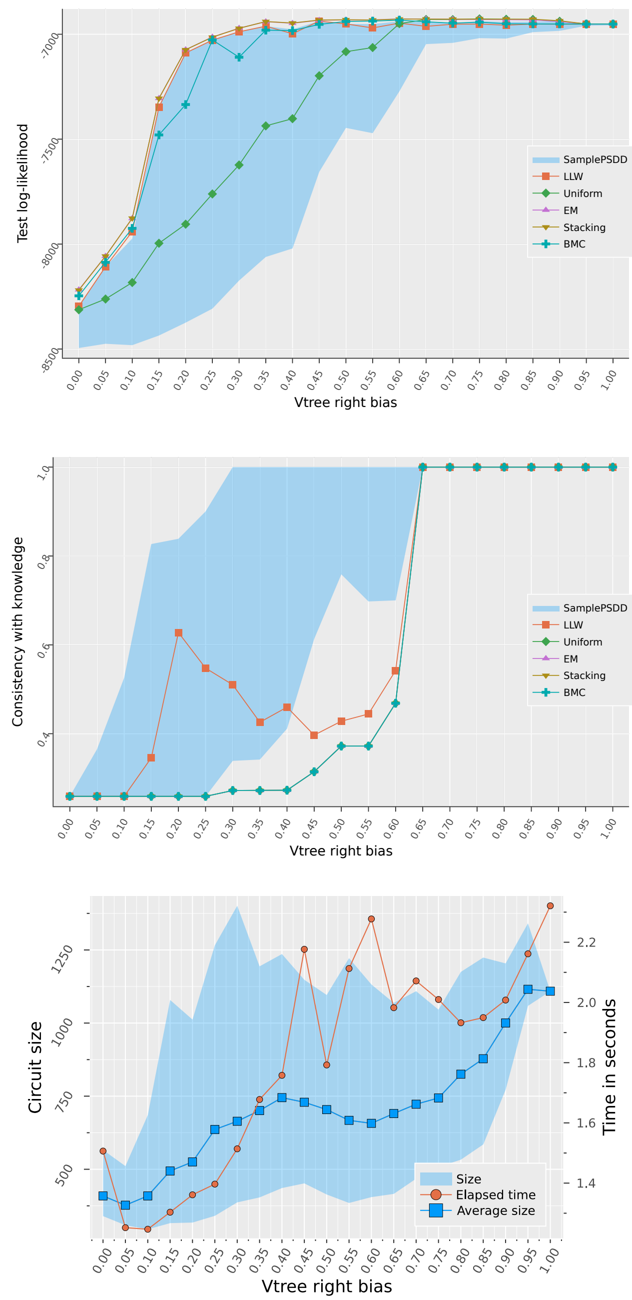


Figure 6: PSDD samples generated from different vtree biases and evaluated on data fitness (top), consistency with formula (middle) and circuit complexity (bottom).

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.

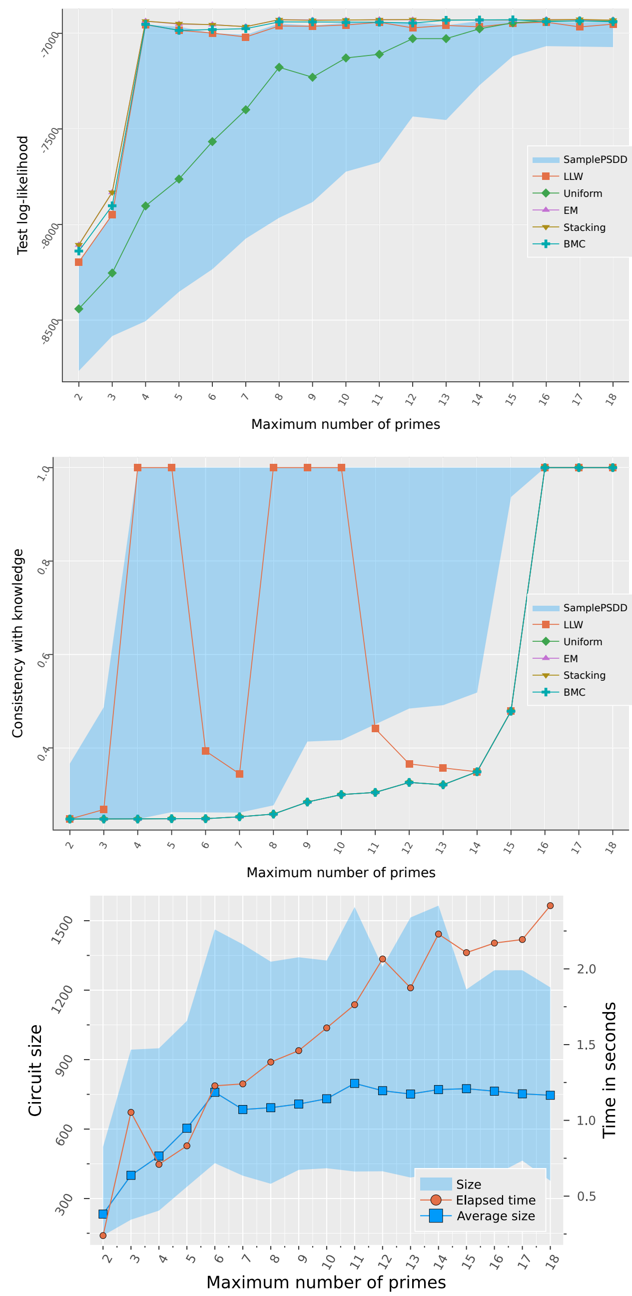


Figure 7: Performance of SAMPLEPSDD when varying k, evaluating log-likelihood (top), satisfiability with the original constraints (middle) and circuit complexity (bottom).

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

Jessa Bekker, Jesse Davis, Arthur Choi, Adnan Darwiche, and Guy Van den Broeck. Tractable learning for complex probability queries. In *Advances in Neural Information Processing Systems*, pages 2242–2250, 2015.

Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691, 1986.

Arthur Choi and Adnan Darwiche. Dynamic minimization of sentential decision diagrams. In *Proceedings of the Twenty-Seventh AAAI Conference on Artificial Intelligence*, pages 187–194, 2013.

Arthur Choi, Guy Van den Broeck, and Adnan Darwiche. Tractable learning for structured probability spaces: A case study in learning preference distributions. In *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence*, pages 2861–2868, 2015.

Arthur Choi, Nazgol Tavabi, and Adnan Darwiche. Structured features in naive Bayes classification. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*, pages 3233–3240, 2016.

Arthur Choi, Yujia Shen, and Adnan Darwiche. Tractability in structured probability spaces. In *Advances in Neural Information Processing Systems*, volume 30, pages 3477–3485, 2017.

Alvaro H. C. Correia, Robert Peharz, and Cassio de Campos. Joints in random forests. In *Advances in Neural Information Processing Systems 33*, 2020.

Meihua Dang, Antonio Vergari, and Guy Van den Broeck. Strudel: Learning structured-decomposable probabilistic circuits. *CoRR*, abs/2007.09331, 2020.

Meihua Dang, Pasha Khosravi, Yitao Liang, Antonio Vergari, and Guy Van den Broeck. Juice: A julia package for logic and probabilistic circuits. In *Proceedings of the 35th AAAI Conference on Artificial Intelligence (Demo Track)*, 2021.

Adnan Darwiche. A differential approach to inference in bayesian networks. *Journal of the ACM*, 50(3):280–305, 2003.

Adnan Darwiche. SDD: A new canonical representation of propositional knowledge bases. In *Proceedings of the Twenty-Second International Joint Conference on Artificial Intelligence*, pages 819–826, 2011.

Niklas Eén and Niklas Sörensson. Translating pseudo-boolean constraints into SAT. *Journal on Satisfiability, Boolean Modeling and Computation*, 2006.

Wolfgang Gatterbauer and Dan Suciu. Oblivious bounds on the probability of boolean functions. *ACM Transactions on Database Systems*, 39(1), 2014.

Renato Geh, Denis Mauá, and Alessandro Antonucci. Learning probabilistic sentential decision diagrams by sampling. In *Proceedings of the VIII Symposium on Knowledge Discovery, Mining and Learning*. SBC, 2020.

Robert Gens and Pedro Domingos. Learning the structure of sum-product networks. In *Proceedings of the 30th International Conference on Machine Learning*, pages 873–880, 2013.

Toshihiro Kamishima. Nantonac collaborative filtering: Recommendation based on order responses. In *Proceedings of the Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. Association for Computing Machinery, 2003.

Doga Kisa, Guy Van den Broeck, Arthur Choi, and Adnan Darwiche. Probabilistic sentential decision diagrams. *Knowledge Representation and Reasoning Conference*, 2014.

Yitao Liang, Jessa Bekker, and Guy Van den Broeck. Learning the structure of probabilistic sentential decision diagrams. In *Proceedings of the Thirty-Third Conference on Uncertainty in Artificial Intelligence*, 2017.

Lilith Mattei, Décio L. Soares, Alessandro Antonucci, Denis D. Mauá, and Alessandro Facchini. Exploring the space of probabilistic sentential decision diagrams. In *3rd Workshop of Tractable Probabilistic Modeling*, 2019.

Lilith Mattei, Alessandro Antonucci, Denis Deratani Mauá, Alessandro Facchini, and Julissa Villanueva Llerena. Tractable inference in credal sentential decision diagrams. *International Journal of Approximate Reasoning*, 125: 26–48, 2020.

Kristine Monteith, James L. Carroll, Kevin Seppi, and Tony Martinez. Turning bayesian model averaging into bayesian model combination. In *The 2011 International Joint Conference on Neural Networks*, 2011.

Masaaki Nishino, Norihito Yasuda, Shin ichi Minato, and Masaaki Nagata. Zero-suppressed sentential decision diagrams. In *AAAI Conference on Artificial Intelligence*, 2016.

Umut Oztok and Adnan Darwiche. A top-down compiler for sentential decision diagrams. In *Proceedings of the 24th International Conference on Artificial Intelligence*, pages 3141–3148, 2015.

Hoifung Poon and Pedro Domingos. Sum-product networks: A new deep architecture. In *Proceedings of the Twenty-Seventh Conference on Uncertainty in Artificial Intelligence*, pages 337–346, 2011.

Tahrima Rahman, Prasanna Kothalkar, and Vibhav Gogate. Cutset networks: A simple, tractable, and scalable approach for improving the accuracy of chow-liu trees. In *Proceedings of the 2014th European Conference on Machine Learning and Knowledge Discovery in Databases*, pages 630–645, 2014.

Yujia Shen, Arthur Choi, and Adnan Darwiche. Tractable operations for arithmetic circuits of probabilistic models. In *Advances in Neural Information Processing Systems*, 2016.

Yujia Shen, Arthur Choi, and Adnan Darwiche. A tractable probabilistic model for subset selection. In *Proceedings of the Thirty-Third Conference on Uncertainty in Artificial Intelligence*, 2017.

Carsten Sinz. Towards an optimal CNF encoding of boolean cardinality constraints. In *Proceedings of the 11th International Conference on Principles and Practice of Constraint Programming*, 2005.

Padhraic Smyth and David Wolpert. Stacked density estimation. In M. Jordan, M. Kearns, and S. Solla, editors, *Advances in Neural Information Processing Systems*, volume 10. MIT Press, 1998.

Antonio Vergari, YooJung Choi, Anji Liu, Stefano Teso, and Guy Van den Broeck. A compositional atlas of tractable circuit operations: From simple transformations to complex information-theoretic queries. *CoRR*, abs/2102.06137, 2021.