Structured Sum-Product Networks

George Chichirim
Keble College
University of Oxford

Supervisor: Prof. Dan Olteanu

3rd Year Project Report
Honour School of Computer Science - Part B
Trinity 2020

Abstract

This thesis introduces a new class of relational Sum-Product Network models learned over multi-relational data called Structured Sum-Product Networks, or SSPNs for short. It puts forward a novel approach to computing the network structure and parameters by exploiting the semantics and structure of the underlying multi-relational databases, as conveyed by the join dependencies of the data, which may significantly improve the construction time and accuracy of such models. The thesis details the design and implementation of a fully relational system for managing SSPNs, and puts forward algorithms that encode SSPN learning and inference as plain SQL queries. It also introduces a user-friendly scripting language to aid the development of SSPN models. This thesis is the first to investigate the structure-aware learning of deep generative models that capture the joint distribution of the underlying multi-relational data. On the MovieLens dataset, SSPNs have better accuracy in terms of MAE and RMSE metrics than 8/10 and 9/10, respectively, best known discriminative models for this task. SSPNs have also significantly better accuracy results than SPFlow, a state-of-the-art system for SPNs. Finally, the SSPN system is three orders of magnitude faster than the SPFlow system for model construction/learning and has two orders of magnitude better throughput for inference queries, where the throughput measure is computed as inference time per network size.

Acknowledgements

First of all, I would like to express my gratitude to my supervisor Prof. Dan Olteanu for his support throughout the project, for the many productive meetings of brainstorming, and for the valuable feedback he has given during this year.

I would also like to thank Fabian Peternek for participating in all our discussions, and for helping with the implementation of the scripting language parser. Moreover, I would like to thank Maximilian Schleich for his suggestions and feedback for this research project.

Finally, I would like to thank my family for their continuous love and support, and for always encouraging me to do my best, ever since I can remember.

Table of Contents

Chapter 1

Introduction

In this thesis, we investigate Sum-Product Networks (SPNs) and the problem of learning both their structure and parameters over multi-relational databases. We introduce a new class of *structure-aware deep probabilistic models* which we will call **Structured Sum-Product Networks**, or **SSPNs** for short. In contrast to standard SPNs, SSPNs are modelled on the relational structure of the underlying training dataset. This can lead to smaller SPNs that can be trained faster and can have higher accuracy. Moreover, we create a **fully relational framework** through which we can model, learn, and use SSPNs to answer inference queries.

1.1 Motivation

1.1.1 Why SPNs?

Probabilistic Graphical Models (PGMs) [16] can compactly represent many complex distributions, but probabilistic inference is generally intractable for them [32]. Deep architectures with many layer of hidden variables are expressive too, but again inference in them is difficult [4].

Sum-Product Networks were introduced by Poon and Domingos [30] as both tractable probabilistic models and deep architectures. They are in fact deep networks with probability distributions as leaves, and products or weighted sums of sub-SPNs as inner nodes. While these restrictions limit their expressiveness, they allow for clear semantics in the sense that sub-SPNs rooted at each node compute a joint probability distribution over the variables from their scope.

The most important advantage of SPNs over other types of PGMs, including Bayesian Networks (BNs) and Markov Networks (MNs), is that exact inference

can be achieved in time linear in the size of the network. In contrast, in both BNs and MNs, probabilistic inference is #P-complete [32], and therefore it needs to be approximated in order to ensure tractability. This advantage of SPNs has generated a lot of interest since inference is often a core task for both structure and parameter learning. SPNs have actually achieved impressive results in image completion [30, 8, 27], classification [9], and speech and language modelling [28, 5].

1.1.2 Why Relational Data?

According to a 2017 Kaggle survey on the state of data science and machine learning [15], the majority of practical data science tasks involve relational data: in Retail, 86% of used data is relational; in Insurance, it is 83%; in Marketing, it is 82%; and in Finance it is 77%. Furthermore, relational data benefit from the investment of many human hours for cleaning and normalization and are rich with knowledge of the underlying domain modelled using database constraints.

However, as Schleich et al. [33] pointed out, the current state of affairs in building ML models over relational data largely ignores the structure and rich semantics readily available in relational databases.

By far the most common approach to learning over relational data is to use two distinct systems: the data systems for managing the training dataset and the ML library for model training. The data system first computes the training dataset as the result of a *feature extraction query* and exports it as one table. The ML library then imports the training dataset in its own format and learns the desired model. This approach is called *data structure-agnostic* and is depicted with red in Figure 1.1. The key disadvantage of this is the non-trivial time spent on materialising, exporting and importing the training dataset, which is commonly orders of magnitude larger than the input database.

An alternative approach is to migrate the statistical package into the database system space. Here, each machine learning task is implemented as a distinct user-defined aggregate function (UDAF) inside the database system. This allows for better runtime performance since it does not need to export and import the (usually large) training dataset. A prime example of this approach is MADLib [13] that extends PostgreSQL with a comprehensive library of machine learning UDAFs. Furthermore, the UDAFs can be pushed into the feature extraction query exploiting the structure of the data, thus reducing the complexity and drastically improving the runtime performance of the learning process. This approach is called *data structure-aware* and is depicted with green in Figure 1.1. It first

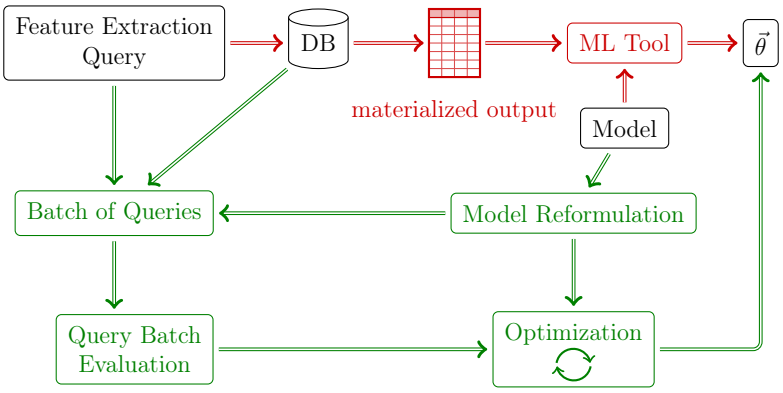


Figure 1.1: Taken from [33]. Relational structure-aware (green) versus relational structure-agnostic (red) learning.

compiles the model specification into a set of aggregates (model reformulation). Data dependencies such as functional dependencies can be used to reparametrise the model, so a model over a smaller set of functionally determining features is learned instead and then mapped back to the original model. Join dependencies, such as those prevalent in feature extraction queries that put together several input tables, are exploited to avoid redundancy in the representation of join results and push the model aggregates past joins. The model aggregates over the feature extraction query define a batch of queries. The result of a query batch is then the input to an optimizer such as a gradient descent method that iterates until the model parameters converge.

1.2 Contributions

So far, to the best of our knowledge, *no work considered structure-aware learning of deep probabilistic models such as Sum-Product Networks.* There is available expertise in the AI and DB communities on either SPNs or structure-aware computation, but not both [34, 2, 3]. The contributions of this thesis can be divided into two major parts.

Firstly, we introduce a new class of SPNs that we call *Structure-aware (or simply Structured) Sum-Product Networks.* These models will succinctly and accurately capture joint probability distributions learned from *large multi-relational databases* and allow for *tractable probability inference* and *efficient learning* orders of magnitude faster than what is possible with the current technology. The key conjecture of this project is that the learning time and accuracy of such models can be drastically improved by exploiting the *semantics and structure* of the

underlying multi-relational database, including: domain expertise expressed as logical rules, database dependencies and normal forms, human-added semantics in the form of concept hierarchies, and feature extraction queries that construct the training dataset from the input database.

SSPNs draw on a novel combination of prior work from the AI community on Sum-Product Networks [30, 34], and from DB community on Factorised Databases (FDBs) [25, 26], which are succinct lossless representations of relational data. FDBs exploit laws of relational algebra, in particular the distributivity of the Cartesian product over union that underlies algebraic factorisation, and data together with computation sharing to reduce redundancy in the representation and computation of query results [24].

SSPNs inherit the best of SPNs and FDBs. Like Sum-Product Networks, SSPNs are representations of joint probability distributions learned from the input data and support tractable inference. Like Factorised Databases, SSPNs exploit the structure of the underlying relational data to achieve small sizes and fast construction. Learning Sum-Product Networks is notoriously challenging, with existing systems reporting significant runtimes for moderately small data sizes. One reason for this is that they do not exploit the existing structure of the underlying data and set out to re-discover it using expensive clustering and independence testing [10].

The ability to learn deep models over relational data with less computing resources than the state of the art can also significantly improve model accuracy. Firstly, we can train SSPNs over larger amounts of relational data. It also means that models can be quickly refreshed so they continue to represent accurately the domain even in the presence of high throughput of new data evidence. It also means that more models can be trained within the time budget allocated to train one model using the existing technology, so that we can choose the one with the highest accuracy.

Secondly, we introduce a *fully relational system* for modelling, learning, and using SSPNs. Generally, SPNs can be realised in a multitude of ways. Our design decision is to create SSPNs that are defined by the joins of several tables stored in a relational DBMS. Learning and prediction can then happen purely relational. There are tremendous benefits of this approach:

1. Since relational DBMSs are designed to deal with large tables means that this approach can deal with truly large training datasets and learned SSPNs that may not fit entirely in the main memory of a commodity machine.

2. The learned SSPNs can be made persistent to disk. This contrasts with all existing approaches that represent SPNs in main memory as graphs that are only kept for as long as prediction is performed and are discarded afterwards. Subsequent need of the SPNs means learning them again! Our relational representation of the SSPNs can be seen as a relational normalisation of the graph representation of the SPN.

3. Prediction using SSPNs is mapped purely to relational queries. More importantly, learning SSPNs is also defined using relational queries. The overall system is thus purely relational.

4. This approach is portable across DBMSs, since the relational queries are given in a standard query language called SQL, which is supported by virtually all major DBMSs. This means our approach is highly usable, adding the fact that DBMSs are literally everywhere (e.g., SQLite is one of the most used software libraries along with zlib, libpng, and libjpeg).

There is a price to pay for this high usability and scalability: One can implement an SPN-specific learning and prediction engine that would have lower footprint and higher runtime performance than a general-purpose relational DBMS. Nevertheless, we consider that the benefits of our approach outweigh by a large margin this shortcoming. In experiments we show that our approach achieves better accuracy and is much faster than prior SPN systems.

Besides model storage and evaluation, we consider the problem of maintaining SSPN models incrementally under changes (tuple insertions and deletions) to the input database. We consider a novel approach that breaks down the construction of the network into three components: the construction of the factorised join that gives the structure of the network (structure estimation), the computation of the weights on the children of a sum node (parameter estimation), and the computation of the probability distributions at the leaves of the network (pdf estimation). We introduce algorithms that efficiently maintain all three components under updates.

Last but not least, we introduce a user-friendly scripting language to aid the development of SSPN models, from the preparation of the training dataset, to the definition of the SSPN structure and probability distributions, and finally to the inference (conditional and most probable explanation) queries over SSPNs.

Chapter 2

Preliminaries

2.1 Factorised Databases

Factorised Databases (FDB) were first introduced by Olteanu and Závodnỳ [25, 26]. In this section we will briefly explain them and present the results that will make us understand better their contribution to this thesis.

The idea is to represent relations symbolically as relational algebra expressions consisting of Unions, Cartesian Products, and singleton relations. Such representations are called *factorised representations* or *f-representations*, since they use algebraic factorisation to nest products and unions, and hence express combinations of values symbolically. Further compression can be achieved by introducing symbolic references into the representations, so that repeated subexpressions can be defined only once and be referred to several times. These kind of factorised representations are called *d-representations*.

2.1.1 Factorised Data Representations

Definition 2.1. A *factorised representation with definitions*, or d-representation, is a list of expressions $(D_1, ..., D_n)$ where each D_k is a relational algebra expression over a schema Σ_k. An expression over schema Σ from above has one of the following forms:

- $\varnothing$, representing the empty relation over Σ,
- $\langle\rangle$, representing the relation consisting of the nullary tuple, if $\Sigma = \varnothing$,
- $\langle A : a\rangle$, representing the singleton relation with one tuple (a), if $\Sigma = \{A\}$ and $a \in Dom(A)$,

- $(E_1 \cup ... \cup E_n)$, representing the union of the relations represented by E_i, where each E_i is an expression over Σ,
- $(E_1 \times ... \times E_n)$, representing the Cartesian product of the relations represented by E_i, where each E_i is an expression over schema Σ_i such that Σ is the disjoint union of all Σ_i,
- **a reference $^{\uparrow}E$ to an expression E over Σ.**

Any expression E that doesn't contain references is an f-representation. We write $[\![E]\!]$ for the relation over schema Σ represented by the expression E over Σ. The schema of a d-representation is the schema of its root expression D_1. Each expression D_i can contain references to D_j only for $j > i$, and it must be referenced at least once if $i > 1$.

Any d-representation D over schema Σ consisting of expressions $D_1, ..., D_n$ represents a relation $[\![D]\!]$ over Σ . For i from n to 1, we can define expression E_i to be the expression D_i with all references $^{\uparrow}D_j$ $(j > i)$ replaced by E_j. E_1 is now an f-representation, called the *traversal* of D, equivalent to D, and therefore representing the same relation $[\![D]\!]$. Moreover, any f-representation E is also a d-representation (by definition).

Theorem 2.2. *Factorised representations are a complete representation system for relational data.*

Proof. Any relation has at least one f-representation, the *flat* f-representation. This is a (possibly empty) union of products of singletons, where each product of singletons represents a distinct tuple in the relation. □

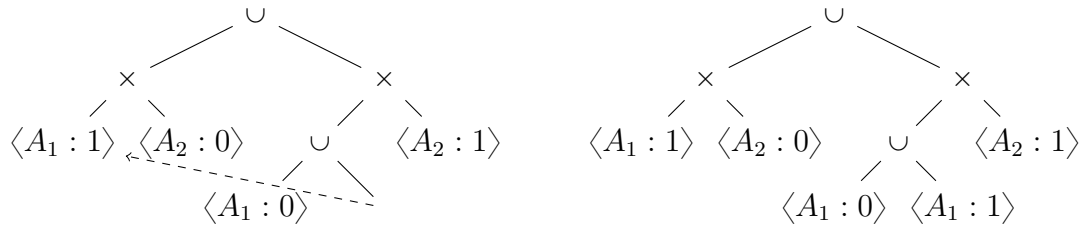
Any f-representation has a parse tree whose internal nodes are unions and products, and whose leaves are singletons or empty relations. Similarly, any d-representation has a directed acyclic parse graph (the graph is acyclic due to the restriction that each D_i can contain a reference to D_j only for $j > i$).

Example 2.3. Consider the relation $\boldsymbol{R}_n$ over schema $\{A_1, ..., A_n\}$ whose tuples are all binary sequences $(a_1, ..., a_n)$ with no two consecutive zeros. The d-representation consisting of

$$D_{1,0} = \langle A_1 : 0 \rangle, \quad D_{1,1} = \langle A_1 : 1 \rangle,$$

$$D_{k,0} = {}^{\uparrow}D_{k-1,1} \times \langle A_k : 0 \rangle, \quad D_{k,1} = ({}^{\uparrow}D_{k-1,0} \cup {}^{\uparrow}D_{k-1,1}) \times \langle A_k : 1 \rangle \quad \text{for } k \in \overline{2, n},$$

and root $D = {}^{\uparrow}D_{n,0} \cup {}^{\uparrow}D_{n,1}$, represents the relation $\boldsymbol{R}_n$.[1] This can be seen by showing inductively over k that $D_{k,b}$ represents the relation $\sigma_{A_k=b}(\boldsymbol{R}_k)$. Below are the parse graph of the above d-representation, and the parse tree of its equivalent f-representation, for the case $n = 2$:



Definition 2.4. The size $|D|$ of a d-representation D is the total number of its singletons, empty set symbols, unions, products, and occurrences of references. The number of singletons in D is denoted by $\|D\|$.

Even though any relation has a flat f-representation, nested f-representations can be exponentially smaller than their equivalent flat f-representations, where the exponent is the size of the schema.

Example 2.5. The f-representation $(\langle A_1 : 0\rangle \cup \langle A_1 : 1\rangle) \times \ldots \times (\langle A_n : 0\rangle \cup \langle A_n : 1\rangle)$ has $2n$ singletons, while any equivalent flat f-representation has $n * 2^n$ singletons.

By caching common subexpressions, d-representations can be exponentially smaller than their equivalent f-representations.

Example 2.6. The d-representation from Example 2.3 has size $O(n)$, while the size of its equivalent f-representation is exponential in n since only the singleton $\langle A_1 : 1\rangle$ occurs F_n times (where F_n is the n^{th} Fibonacci number).

2.1.2 Factorised Joins Over Multi-Relational Databases

So far we have seen how to represent relational data more efficiently using factorised representations. The same idea can be extended to Join Queries over Multi-Relational Databases.

Definition 2.7. The size $|Q|$ of a join query Q is the number n of its relation symbols.

[1] In Definition 2.1 the expressions D_i are indexed by natural numbers $i \in \overline{1,n}$, but any partial order with a least element is sufficient because it can be re-indexed by consecutive natural numbers that satisfy the two restrictions.

Definition 2.8. Given a join query Q, two variables A and B are *conditionally independent given* a set of variables S if, for any database $\boldsymbol{D}$, A's assignments do not constrain B's assignments given assignments for S in $\boldsymbol{D}$; otherwise, A and B are dependent.

Example 2.9. In the join query $R_1(A, B) \bowtie R_2(A, C)$, the variables B and C are independent given variable A, whereas A and B are dependent on each other as imposed by relation R_1.

Definition 2.10. Given a join query Q, a *variable order* Δ for Q is a pair (T, key), where:

- T is a rooted tree with one node per variable in Q such that the variables of each relation symbol in Q lie along the same root-to-leaf path in T,
- The function key maps each variable A to the subset of its ancestor variables in T on which the variables in the subtree rooted at A depend, i.e., for every variable B that is a child of a variable A, $key(B) \subseteq key(A) \cup \{A\}$.

Variable orders define the nesting structure of the factorised join queries, they guide the grounding process that computes the factorised representation of a join query, and they define the asymptotic size bounds for factorised queries and the time complexity to compute them.

The conditional independence of variables is modelled in a variable order by branching: two variables A and B on different branches in a variable order Δ are conditionally independent given their common ancestors. The first constraint in Definition 2.10 states that all variables of a relation symbol are dependent and, therefore, they cannot lie on different root-to-leaf paths in Δ, since that would mean they are independent.

Example 2.11. Taken from [24]. Figure 2.1(a) depicts three relations and their natural join. The join result exhibits a high degree of redundancy. The value l_1 occurs in 12 tuples, each value c_1 and c_2 occurs in six tuples and they are paired with the same tuples of values for the other attributes.

Since l_1 is paired in **Competition** with c_1 and c_2, and in **Branch** with p_1 and p_2, the Cartesian product of $\{c_1, c_2\}$ and $\{p_1, p_2\}$ occurs in the join result. We can represent this product symbolically as $\{c_1, c_2\} \times \{p_1, p_2\}$, instead of materialising it. If we systematically apply this observation, we obtain an equivalent *factorised representation* (Figure 2.1(d)) of the entire join result that is much more compact than its flat representation.

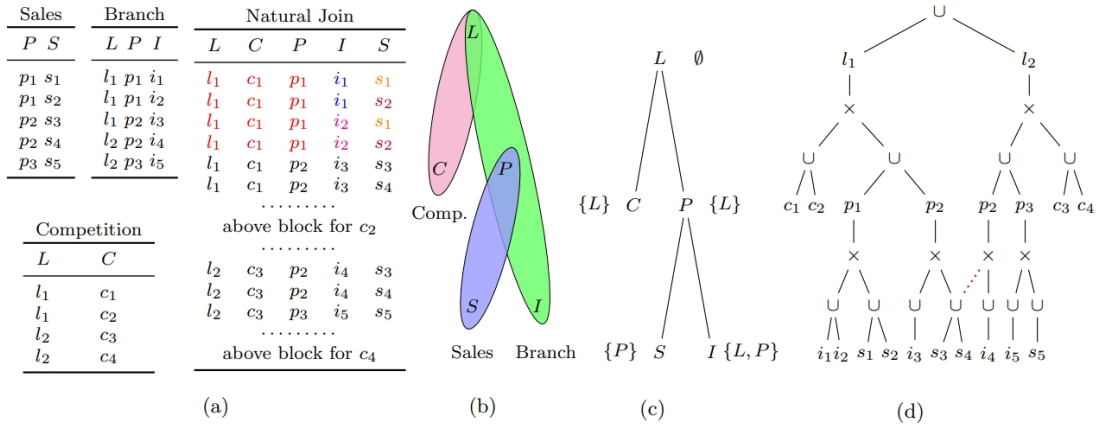


Figure 2.1: Taken from [24]. (a) Database with relations Branch(Location, Product, Inventory), Competition(Location, Competitor), Sales(Product, Sale), where the attribute names are abbreviated; (b) Hypergraph of the natural join of the relations; (c) Variable order Δ defining one possible nesting structure of the factorised join result given in (d). The union $s_3 \cup s_4$ is cached under the first occurrence of p_2 and referenced (via a dotted edge) from the second occurrence of p_2.

Definition 2.12. The *d-trees* are the variable orders from Definition 2.10 and represent the nesting structures of the d-representations of join query results.

Remark. In case $key(A)$ is not the set of all ancestors of variable A in a d-tree Δ, then in a d-representation over Δ, the same factorisation fragments rooted at A-values may be repeated for every tuple of values for ancestors not in $key(A)$. Here is where the referencing is used: we store this fragment only once and we refer to it instead of repeatedly copying it.

Definition 2.13. The *f-trees* are the d-trees where $key(A)$ is the set of all ancestors of A, for all variables A. The f-trees are the nesting structures of f-representations.

2.1.3 Size Bounds of Factorised Joins

Given a join query Q, there may be several variable orders for Q and they define factorised representations of the result query of different sizes.

Definition 2.14. For a join query Q, we define $H(Q) = (V, E)$ as the hypergraph that has one node in V per query variable in Q, and one hyperedge in E per relation in Q. Figure 2.1(b) depicts the hypergraph of the natural join of the three relations from Figure 2.1.

Definition 2.15. An edge cover of $H(Q)$ is a subset of edges of $H(Q)$ such that each node appears in at least one edge. The minimum edge cover problem is the problem of finding an edge cover of minimum size.

The edge cover problem can be formulated as an integer programming problem by assigning to each edge R_i a variable x_{R_i} that can be 1 if R_i is part of the cover, and 0 otherwise.

The Cartesian product of the relations in an edge cover includes the query result, so for any database $\boldsymbol{D}$ we have:

$$|Q(D)| \leqslant |R_1|^{x_{R_1}} \times ... \times |R_n|^{x_{R_n}} \leqslant N^{\sum_{i=1}^{n} x_{R_i}} \tag{2.1}$$

By minimising the size of the edge cover, we can obtain a more accurate upper bound for the size of the query result. This bound becomes tight for fractional solutions [1].

Definition 2.16 ([1])**.** Given a join query Q over a database $\boldsymbol{D} = (R_1, ..., R_n)$, the *fractional edge cover number* $\rho^*(Q)$ is the cost of an optimal solution to the linear program with variables $\{x_{R_i}\}_{i=1}^{n}$:

$$\begin{aligned} &\text{minimise} \sum_{i=1}^{n} x_{R_i} \\ &\text{subject to} \sum_{R_i \in rel(A)} x_{R_i} \geqslant 1 \text{ for each query variable } A \\ &\qquad x_{R_i} \geqslant 0 \qquad\qquad \text{for each } 1 \leqslant i \leqslant n \end{aligned} \tag{2.2}$$

Theorem 2.17 ([26])**.** *For an f-representation over an f-tree Δ of a join query Q, the number s_A of values of a variable A is bounded by the fractional edge cover number of the join query that is a $(key(A) \cup A)$-restriction of Q.*

Corollary 2.18. *An upper bound on the size of the f-representation over Δ is then the maximum over all variables in Δ of the number of values of A:*

$$s(\Delta) = max\{\rho^*(Q_{key(A)\cup\{A\}}) \mid A \text{ is a variable in } \Delta\} \tag{2.3}$$

Corollary 2.19. *The f-tree width $s(Q)$ and d-tree width $s^{\uparrow}(Q)$ are then the minimum of the previous upper bound over all f-trees and d-trees, respectively:*

$$\begin{aligned} s(Q) &= min\{s(\Delta) \mid \Delta \text{ is an f-tree of } Q\} \\ s^{\uparrow}(Q) &= min\{s(\Delta) \mid \Delta \text{ is a d-tree of } Q\} \end{aligned} \tag{2.4}$$

Theorem 2.20 ([26]). *The d-tree width $s^{\uparrow}(Q)$ is equal to the fractional hypertree width of the join query fhtw(Q).*

We know that $1 \leqslant s^{\uparrow}(Q) \leqslant s(Q) \leqslant \rho^*(Q) \leqslant |Q|$. The gap between $s(Q)$ and $\rho^*(Q)$ can be as large as $|Q|$ (e.g., for hierarchical queries), whereas the gap between $s^{\uparrow}(Q)$ and $s(Q)$ can be as large as $log|Q|$ (e.g., for path queries). Clique queries (e.g., triangles) are the pathological cases for which factorisation bring no asymptotic saving.

Proposition 2.21 ([26, 1]). *Given a join query Q, for any database $\boldsymbol{D}$, the join query result $Q(\boldsymbol{D})$ admits*

- *a flat representation of size $O(|\boldsymbol{D}|^{\rho^*(Q)})$,*
- *an f-representation over f-trees of size $O(|\boldsymbol{D}|^{s(Q)})$,*
- *a d-representation over d-trees of size $O(|\boldsymbol{D}|^{fhtw(Q)})$.*

There are classes of databases for which the size bounds in Proposition 2.21 are asymptotically tight. Furthermore, there are algorithms that compute the join query result of the three representations in worst-case optimal time (i.e., the computation time is the same as the size bound modulo log factors in the size of the input relations) [26].

2.2 Sum-Product Networks

Sum-Product Networks (SPNs) were first introduced by Poon and Domingos [30]. In this section we will briefly explain them and present the results that will make us understand better their contribution to this thesis.

SPNs are deep neural networks containing only summations and multiplications. While these restrictions limit their expressiveness, they allow us to attribute clear semantics to each node in the network in the sense that sub-SPNs rooted at each node compute a joint probability distribution over the variables from their scope. In this sense, SPNs can also be seen as a tractable probabilistic graphical model (PGM). In fact, SPNs are equivalent to Arithmetic circuits (ACs) [6, 31], and they can be converted into equivalent traditional probabilistic graphical models such as Bayesian networks (BNs) and Markov networks (MNs) by treating sum nodes as hidden variables [35]. An important advantage of SPNs over BNs and MNs is that marginal inference can be done without any approximation in linear time in the size of the network.

We use the following notation adapted from Peharz et al. [29] throughout the thesis. Random variables (RVs) are denoted by X, Y, Z, etc. The set of values (states) a RV X can take is $val(X)$, and its value is denoted by $x \in val(X)$. Sets of RVs are denoted by boldface letters, e.g., $\mathbf{X} = \{X_1, ..., X_N\}$. We define the Cartesian product $\mathbf{val}(\mathbf{X}) = \times_{i=1}^{N} val(X_i)$, and we use $\mathbf{x}$ for elements of $\mathbf{val}(\mathbf{X})$. For $X \in \mathbf{X}$ we define $\mathbf{x}[X]$ to be the projection of $\mathbf{x}$ onto X. An element $\mathbf{x} \in \mathbf{val}(\mathbf{X})$ represents a *complete evidence* (or complete state), assigning to each RV in $\mathbf{X}$ a value.

Partial evidence about X is represented as a subset $\mathcal{X} \subseteq val(X)$, which is an element of the sigma-algebra $\mathcal{A}_X$ induced by the RV X. More precisely, for discrete RV X, we have that $\mathcal{A}_X = \mathcal{P}(val(X))$ (i.e., the power-set of $val(X)$). For continuous RV X, in this thesis we always have that $val(X) = \mathbb{R}$, thus $\mathcal{A}_X$ is the set of all subsets $\mathcal{X} \subseteq \mathbb{R}$ that can be obtained by performing a countable union of intervals from $\mathbb{R}$. For sets of RVs $\mathbf{X} = \{X_1, ..., X_N\}$, we define the product sets $\mathcal{H}_{\mathbf{X}} = \{\times_{i=1}^{N} \mathcal{X}_i \mid \mathcal{X}_i \in \mathcal{A}_{X_i}\}$ to represent partial evidence about $\mathbf{X}$, and we use $\boldsymbol{\mathcal{X}}$ for elements of $\mathcal{H}_{\mathbf{X}}$. For $X \in \mathbf{X}$ and $\boldsymbol{\mathcal{X}} \in \mathcal{H}_{\mathbf{X}}$, we define $\boldsymbol{\mathcal{X}}[X] = \{\mathbf{x}[X] \mid \mathbf{x} \in \boldsymbol{\mathcal{X}}\}$.

We use the following abbreviations. For example, for RVs X and Y with $val(X) = \mathbb{N}$ and $val(Y) = \mathbb{R}$, a complete evidence can be $X = 1, Y = 2.5$, while partial evidence can be $X = 1, Y \in (1, 2)$ or $X \in \{3, 4, 5\}$. Notice that in

the second example the absence of Y is an abbreviation for the nonrestrictive evidence $Y \in val(Y)$. We also abbreviate the evidence $\boldsymbol{\mathcal{X}} = \times_{i=1}^{N} val(X_i)$ with $*$.

2.2.1 Network Polynomials

Darwiche [6] introduced *network polynomials* (NPs) for Bayesian networks over random variables $\mathbf{X}$ with finitely many states. Poon and Domingos [30] built on this idea and generalised them to unnormalised distributions (i.e., any non-negative function $\Phi(\mathbf{x})$). Let $\lambda_{X=x} \in \mathbb{R}$ be the indicator variables (IVs) for each random variable X and each state $x \in val(X)$.

Definition 2.22. Let Φ be an unnormalised probability distribution over RVs $\mathbf{X}$ with finitely many states, and let $\boldsymbol{\lambda}$ be their IVs. The *network polynomial* f_Φ of Φ is defined as:

$$f_\Phi(\boldsymbol{\lambda}) = \sum_{\mathbf{x} \in \mathbf{val}(\mathbf{X})} \Phi(\mathbf{x}) \times \prod_{X \in \mathbf{X}} \lambda_{X=\mathbf{x}[X]} \tag{2.5}$$

The NP represents the distribution Φ in the following sense. Restrict the IVs to $\{0, 1\}$, and as functions of $\mathbf{x} \in \mathbf{val}(\mathbf{X})$, where:

$$\lambda_{X=x}(\mathbf{x}) = \begin{cases} 1 & \text{if } x = \mathbf{x}[X] \\ 0 & \text{otherwise} \end{cases} \tag{2.6}$$

Let $\boldsymbol{\lambda}(\mathbf{x})$ be the corresponding vector-valued function, collecting all $\lambda_{X=x}(\mathbf{x})$. When we input $\boldsymbol{\lambda}(\mathbf{x})$ to f_Φ for a complete evidence $\mathbf{x} \in \mathbf{val}(\mathbf{X})$, all but one of the terms of the sum evaluate to 0, leaving us with $f_\Phi(\boldsymbol{\lambda}(\mathbf{x})) = \Phi(\mathbf{x})$.

The unnormalised probability of partial evidence $\boldsymbol{\mathcal{X}}$ is $f_\Phi(\boldsymbol{\lambda}(\boldsymbol{\mathcal{X}}))$, where:

$$\lambda_{X=x}(\boldsymbol{\mathcal{X}}) = \begin{cases} 1 & \text{if } x \in \boldsymbol{\mathcal{X}}[X] \\ 0 & \text{otherwise} \end{cases} \tag{2.7}$$

The NP compactly describes marginalisation over arbitrary domains of the RVs by simply setting the corresponding IVs to 1. In particular, when the evidence is $*$ and thus $\lambda_{X=x} = 1$ for all X and x, $f_\Phi(\boldsymbol{\lambda}(*))$ is the normalisation constant of Φ.

A direct implementation of the NP is obviously not practical due to the exponentially many terms.

Example 2.23. The network polynomial f_Φ of the unnormalised probability

X	Y	Φ
0	0	0.2
0	1	0.5
1	0	0.3
1	1	0.7

(a)

Evidence $\boldsymbol{\mathcal{E}}$	$\lambda_{X=0}$	$\lambda_{X=1}$	$\lambda_{Y=0}$	$\lambda_{Y=1}$
$X=0, Y=1$	1	0	0	1
$X=1, Y=1$	0	1	0	1
$X=1$	0	1	1	1
$*$	1	1	1	1

(b)

Figure 2.2: (a) Probability table for Φ; (b) Examples of evidence indicators.

distribution Φ over the Boolean RVs X and Y from Figure 2.2(a) is:

$$f_\Phi(\lambda_{X=0}, \lambda_{X=1}, \lambda_{Y=0}, \lambda_{Y=1}) = \\ 0.2\lambda_{X=0}\lambda_{Y=0} + 0.5\lambda_{X=0}\lambda_{Y=1} + 0.3\lambda_{X=1}\lambda_{Y=0} + 0.7\lambda_{X=1}\lambda_{Y=1}$$

Figure 2.2(b) depicts some examples of evidence and the corresponding values of the indicator variables.

2.2.2 Finite State Sum-Product Networks

Poon and Domingos [30] initially defined SPNs over Boolean Random Variables with a straightforward extension to finite multi-valued discrete RVs.

For some node n in an acyclic directed graph, we denote with $ch(n)$ the set of children of n, and with $desc(n)$ the set of descendants of n.

Definition 2.24. Let $\mathbf{X} = \{X_1, X_2, ..., X_N\}$ be a set of finite multi-valued discrete random variables, and let $\boldsymbol{\lambda}$ be their IVs. A *Sum-Product Network* (SPN) over the variables in $\mathbf{X}$ is a rooted directed acyclic graph whose internal nodes are sums and products, and whose leaves are the IVs. The edges linking each sum node s to its children $c \in ch(s)$ are labeled with non-negative weights $w_{s,c}$.

Let $\boldsymbol{\mathcal{X}}$ be an evidence. We define $V_n(\boldsymbol{\mathcal{X}})$ to be the value computed in a bottom up pass by a node n based on evidence $\boldsymbol{\mathcal{X}}$:

$$V_n(\boldsymbol{\mathcal{X}}) = \begin{cases} \lambda_{X=x}(\boldsymbol{\mathcal{X}}) & \text{if } n \text{ is the leaf node } \lambda_{X=x} \\ \sum_{c \in ch(n)} w_{n,c} \times V_c(\boldsymbol{\mathcal{X}}) & \text{if } n \text{ is a sum node} \\ \prod_{c \in ch(n)} V_c(\boldsymbol{\mathcal{X}}) & \text{if } n \text{ is a product node} \end{cases} \tag{2.8}$$

We denote the sum-product network S as a function of the IVs $\boldsymbol{\lambda}$ by $S(\boldsymbol{\lambda})$. We define the value of S based on evidence $\boldsymbol{\mathcal{X}}$ to be $S(\boldsymbol{\lambda}(\boldsymbol{\mathcal{X}})) = V_{root}(\boldsymbol{\mathcal{X}})$. We

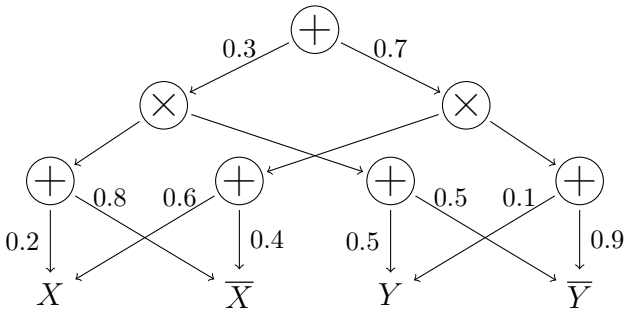


Figure 2.3: SPN S over two Boolean RVs. We denote their IVs with $X, \overline{X}, Y$, and $\overline{Y}$, respectively.

abbreviate this value as $S(\mathbf{x})$ for complete evidence $\mathbf{x}$, and as $S(\boldsymbol{\mathcal{X}})$ for partial evidence $\boldsymbol{\mathcal{X}}$.

Definition 2.25 (Scope)**.** The *scope* of a node n, denoted by $sc(n)$, is the set of variables that appear in the sub-network rooted at n:

$$sc(n) = \begin{cases} \{X\} & \text{if } n \text{ is some leaf node } \lambda_{X=x} \\ \bigcup_{c \in ch(n)} sc(c) & \text{otherwise} \end{cases} \tag{2.9}$$

The sub-network rooted at an arbitrary node n in the SPN S is itself an SPN, which is over $sc(n)$ and which we denote by S_n.

Example 2.26. For the SPN S in Figure 2.3 we have that $S(x, \bar{x}, y, \bar{y}) = 0.3 \times (0.2x + 0.8\bar{x}) \times (0.5y + 0.5\bar{y}) + 0.7 \times (0.6x + 0.4\bar{x}) \times (0.1y + 0.9\bar{y})$. As in Figure 2.2(b) from NPs, we also have here that if a complete state $\mathbf{x}$ is $X = 0, Y = 1$, then $S(\mathbf{x}) = S(0, 1, 1, 0) = 0.148$. If the evidence $\boldsymbol{\mathcal{X}}$ is $X = 1$, then $S(\boldsymbol{\mathcal{X}}) = S(1, 0, 1, 1) = 0.48$. And finally, $S(*) = S(1, 1, 1, 1) = 1$.

The values $S(\mathbf{x})$ for all $\mathbf{x} \in \mathbf{val}(\mathbf{X})$ define an unnormalised probability distribution over $\mathbf{X}$. The unnormalised probability of an arbitrary evidence $\boldsymbol{\mathcal{X}}$ under this distribution is $\Phi_S(\boldsymbol{\mathcal{X}}) = \sum_{\mathbf{x} \in \boldsymbol{\mathcal{X}}} S(\mathbf{x})$. The partition function (or normalisation constant) of the distribution defined by $S(\mathbf{x})$ is $Z_S = \sum_{\mathbf{x} \in \mathbf{val}(\mathbf{X})} S(\mathbf{x})$. Therefore, we can define the probability distribution of an SPN as:

Definition 2.27 (SPN distribution)**.** Let S be an SPN over RVs $\mathbf{X}$. The *probability distribution* represented by S is:

$$P_S(\mathbf{x}) = \frac{S(\mathbf{x})}{Z_S} = \frac{S(\mathbf{x})}{\sum_{\mathbf{x}' \in \mathbf{val}(\mathbf{X})} S(\mathbf{x}')} \tag{2.10}$$

Inference in structurally unconstrained SPNs is in general intractable. Therefore, Poon and Domingos [30] introduced the notion of *validity*, and two conditions that ensure the validity of an SPN, *completeness* and *consistency*.

Definition 2.28 (Validity)**.** An SPN S over $\mathbf{X}$ is *valid* if $S(\boldsymbol{\mathcal{X}}) = \Phi_S(\boldsymbol{\mathcal{X}})$ for all evidence $\boldsymbol{\mathcal{X}} \in \mathcal{H}_{\mathbf{X}}$. In other words, an SPN is valid if it always computes correctly the probability of an evidence. In particular, if S is a valid SPN then $S(*) = Z_S$ (meaning that all RVs are marginalised).

Definition 2.29 (Completeness)**.** A sum node s in SPN S is *complete* if $sc(c) = sc(c'), \forall c, c' \in ch(s)$. The SPN S is complete if every sum node in S is complete.

Definition 2.30 (Consistency)**.** A product node p in SPN S is *consistent* if for all $c, c' \in ch(p)$ with $c \neq c'$ it holds that $\lambda_{X=x} \in desc(c) \implies \forall x' \neq x : \lambda_{X=x'} \notin desc(c')$. The SPN S is consistent if every product node in S is consistent.

Theorem 2.31 ([30])**.** *If an SPN S is both complete and consistent, then S is also valid.*

Note that completeness and consistency are sufficient but not necessary for validity. However, they are necessary for the stronger property that every subnetwork S_n of S is valid [30].

When the SPN S is valid, it ensures that the value computed by S based on some evidence $\boldsymbol{\mathcal{X}}$ is exactly the unnormalised probability of that evidence. Therefore, the probability of an assignment $\mathbf{X} = \mathbf{x}$ (i.e., complete evidence) can be computed as:

$$P(\mathbf{X} = \mathbf{x}) = P_S(\mathbf{x}) = \frac{S(\mathbf{x})}{S(*)} \tag{2.11}$$

The marginal probability of a partial evidence $\boldsymbol{\mathcal{X}}$ can also be computed using the equation 2.11. Furthermore, conditional probabilities can also be computed by evaluating two partial evidences. Let A and B be two events, and let $\boldsymbol{\mathcal{X}}_A$ and $\boldsymbol{\mathcal{X}}_B$ be the corresponding partial evidences. We define $\boldsymbol{\mathcal{X}}_A \cap \boldsymbol{\mathcal{X}}_B$ to be the piecewise intersection. Then:

$$P(A \mid B) = \frac{P(A \cap B)}{P(B)} = \frac{S(\boldsymbol{\mathcal{X}}_A \cap \boldsymbol{\mathcal{X}}_B)}{S(\boldsymbol{\mathcal{X}}_B)} \tag{2.12}$$

Therefore, in a valid SPN, joint, marginal, and conditional probability queries can all be answered by two bottom up evaluations of the network, and hence, exact inference in valid SPNs takes linear time in the size of the network. Alternatively, Bayesian and Markov networks may take exponential time in the size of the network.

2.2.3 Generalised Sum-Product Networks

So far, we have considered SPNs over finite states RVs using IVs. However, SPNs can be generalised by replacing the IVs with probability distributions over arbitrary large scopes [30, 29]. From now on we assume that each RV from $\mathbf{X}$ is either continuous or discrete, where the latter can also have countably infinitely many states.

Poon and Domingos [30] also proposed a more restrictive, and hence easier to achieve, condition for the product nodes that implies consistency:

Definition 2.32 (Decomposability)**.** A product node p in SPN S is *decomposable* if, for all $c, c' \in ch(p)$ with $c \neq c'$, it holds that $sc(c) \cap sc(c') = \varnothing$. The SPN S is decomposable if every product node in S is decomposable.

Definition 2.33 (Locally normalised)**.** In valid SPNs, the normalisation constant $Z_S = \sum_{\mathbf{x} \in \mathbf{val}(\mathbf{X})} S(\mathbf{x})$ can be computed efficiently by a single bottom up evaluation of $S(*)$. We call SPNs with $Z_S = 1$ *normalised* SPNs, for which we have $P_S(\mathbf{x}) = S(\mathbf{x})$. When the weights are normalised for each sum node s (i.e., $\forall s : \sum_{c \in ch(s)} w_{s,c} = 1$), the SPN is automatically normalised [29]. We call such SPNs *locally normalised* SPNs.

Peharz et al. [29] showed that locally normalised SPNs are not a weaker class of models than non-normalised SPNs (i.e., any distribution represented by an SPN with some structure can be represented by a locally normalised SPN with the same structure). Moreover, they also showed that the class of consistent SPNs is not exponentially more compact than the class of decomposable SPNs. They showed that any distribution that can be encoded by a consistent SPN using polynomially many arithmetic operations in $|\mathbf{X}|$ can also be polynomially encoded by a decomposable SPN.

Definition 2.34 (Generalised SPNs)**.** We define generalised SPNs as in Definition 2.24, but now the leaves are distributions over the RVs from $\mathbf{X}$. The scope of a node becomes now:

$$sc(n) = \begin{cases} \mathbf{Y} & \text{if } n \text{ is a distribution over RVs } \mathbf{Y} \subseteq \mathbf{X} \\ \bigcup_{c \in ch(n)} sc(c) & \text{otherwise} \end{cases} \tag{2.13}$$

Ideally, we would want to have the same results with generalised SPNs as with valid finite state SPNs. Therefore, we will require that all sum nodes are

complete and all product nodes are decomposable. Clearly, each node represents a probability distribution over its scope. Actually, we can interpret generalised SPNs as hierarchical mixture models, since each sum node can be seen as a mixture of the distributions encoded by its children, where the probability of each child component is proportional to the weight of the edge between the sum node and that child. Completeness ensures that each child is a distribution over the same set of variables. Similarly, we can think of each product node as factoring a distribution into a product of marginal distributions. Decomposability ensures that the marginal distributions are independent.

Evaluation of $P_S(\mathbf{x}) = S(\mathbf{x})$ for complete state $\mathbf{x}$ clearly works in the same way as for finite state SPNs. Marginalisation also works in a similar way. For partial evidence $\boldsymbol{\mathcal{X}}$ we need to compute:

$$S(\boldsymbol{\mathcal{X}}) = \int_{x_1 \in \mathcal{X}_1} \dots \int_{x_N \in \mathcal{X}_N} S(x_1, \dots, x_N) dx_n \dots dx_1 \tag{2.14}$$

where integrals have to be replaced by sums for discrete RVs. The main observation is that we can pull the integrals from 2.14 over all sums and products down to the input distributions. At sum nodes, we can interchange the integrals with the sum due to completeness. At product nodes, we can interchange the integrals with the product due to decomposability (i.e., since the factors are functions over disjoint variable sets). Therefore, the value $V_n(\boldsymbol{\mathcal{X}})$ becomes now:

$$V_n(\boldsymbol{\mathcal{X}}) = \begin{cases} p_n(\boldsymbol{\mathcal{X}}) & \text{if } n \text{ is a distribution node} \\ \sum_{c \in ch(n)} w_{n,c} \times V_c(\boldsymbol{\mathcal{X}}) & \text{if } n \text{ is a sum node} \\ \prod_{c \in ch(n)} V_c(\boldsymbol{\mathcal{X}}) & \text{if } n \text{ is a product node} \end{cases} \tag{2.15}$$

where $p_n(\boldsymbol{\mathcal{X}})$ is the result of marginalising the distribution at node n over the corresponding scope $sc(n)$ with respect to the partial evidence $\boldsymbol{\mathcal{X}}$.

In particular, if node n is a univariate distribution over RV X, then:

$$p_n(\boldsymbol{\mathcal{X}}) = \begin{cases} \sum_{x \in \boldsymbol{\mathcal{X}}[X]} \mathrm{pmf}_X(x) & \text{if } X \text{ is a discrete RV} \\ \int_{x \in \boldsymbol{\mathcal{X}}[X]} \mathrm{pdf}_X(x) dx & \text{if } X \text{ is a continuous RV} \end{cases} \tag{2.16}$$

Therefore, given that marginalisation at input distributions can be computed efficiently, complete and decomposable generalised SPNs can answer inference queries in linear time in the size of the network, exactly as finite state SPNs do.

Chapter 3

Structured Sum-Product Networks

We have seen in the introduction that, most of the time, the training data that Machine Learning Tools need to import is expected to lay in memory as one big table. However, in many cases, this big table is not straight there, and it is rather the output of a Feature Extraction Query over several much smaller tables. Usually, a Relational Database Management Systems is used for evaluating the query that computes and materialises the output table, which is then exported to the ML library. Our goal is to avoid this separation and build a structure-aware system that tries to benefit to the maximum from the knowledge that can be inferred by exploiting the semantics and structure of the underlying multi-relational databases. In particular, the conditional independence in the training dataset, as conveyed by the join dependencies in the data, or equivalently, by the joins in the feature extraction query.

The key observation that builds the foundation of this project is the similarity between FDBs and SPNs. In both of them, the underlying algebraic structure is a semiring (i.e., similar to a ring but without the requirements that each element must have an additive inverse).

In FDBs, the addition is the set union $\bigcup$ and the multiplication is the Cartesian product $\times$. While in SPNs, the addition and multiplication are the $+$ and $\times$ operators from $\mathbb{R}$.

Both FDBs and SPNs exploit the distributivity of multiplication over addition law to succinctly represent the relational data in one case, and the joint probability distribution of several random variables in the other case.

3.1 Structure Learning

Similar to how variable orders define the nesting structure of the factorised join queries, they are the guidelines for the nesting structure of SSPNs too.

Let Δ be a variable order for the natural join query Q over the database $\boldsymbol{D} = (R_1, ..., R_n)$. The query Q represents the Feature Extraction Query that constructs the training dataset. We restrict ourselves to variable orders Δ that satisfy the following:

- All inner nodes of Δ must be joining nodes (i.e., they are part of more than one relation),
- All leaves of Δ must not be joining nodes (i.e., they are part of exactly one relation). They represent univariate random variables, each variable having a distribution type given by the user (i.e, human-added semantics). We call these nodes distribution nodes.

We talk later about these restrictions and about the fact that they do not constrain the expressiveness of SSPNs.

Definition 3.1. Given a variable order Δ, let $\mathcal{D}(\Delta)$ be the leaves of Δ (i.e., the distribution nodes), let $\mathcal{J}(\Delta)$ be the inner nodes of Δ (i.e., the joining nodes), and let $\mathcal{V}(\Delta)$ be the set of all nodes from variable order Δ.

Definition 3.2 (SSPN)**.** A *Structured Sum-Product Network* (SSPN) over variable order Δ is a generalised SPN over the random variables of Δ.

The method we propose for learning SSPN's nesting structure is summarised in Algorithm 1 and illustrated in Figure 3.1. In particular, it is an alternating structure between sum and product nodes, with univariate distributions as leaves, that accurately and compactly models the training data. It contains a sum node s_A (or a distribution node d_A, respectively) for each variable $A \in \mathcal{J}(\Delta)$ (or $A \in \mathcal{D}(\Delta)$, respectively) and each distinct combination of values for the variables in $key(A)$. It also contains a product node p_A for each variable $A \in \mathcal{J}(\Delta)$ and each distinct combination of values for the variables in $key(A) \cup \{A\}$.

BuildSSPN is a (depth-first) algorithm that, given a variable order Δ, computes the SSPN nesting structure. It also computes some counts that are used for computing the weights (discussed later). The relations $R_1, ..., R_n$ are assumed sorted on their attributes following a depth-first pre-order traversal of Δ. The

Algorithm 1 Constructs the sub-SSPN rooted at A

function BUILDSSPN(A, $(start_i, end_i)_{i \in \overline{1,n}}$, $ancVals$)
  $context \leftarrow \pi_{key(A)}(ancVals)$
  if $cache_A[context]$ is not NULL **then**
    return $cache_A[context]$
  end if
  if $ch(A) = \varnothing$ **then**
    let $i \in \overline{1,n}$ s.t. $A \in schema(R_i)$
    $dNode \leftarrow$ new DistributionNode(A, $\pi_A(R_i[start_i, end_i])$)
    $count(dNode) \leftarrow end_i - start_i + 1$
    if $key(A) \neq anc(A)$ **then**
      $cache_A[context] \leftarrow dNode$
    end if
    return $dNode$
  else
    $sNode \leftarrow$ new SumNode()
    $count(sNode) = 0$
    for $a \in \bigcap_{i \in \overline{1,n}, A \in schema(R_i)} \pi_A(R_i[start_i, end_i])$ **do**
      for $i \in \overline{1,n}$ **do**
        find new ranges $R_i[start'_i, end'_i] \subseteq R_i[start_i, end_i]$ s.t.
          $\pi_A(R_i[start'_i, end'_i]) = a$
      end for
      $pNode \leftarrow$ new ProductNode()
      $count(pNode) \leftarrow 1$
      for $i \in \overline{1,n}$ s.t. A is the last attribute in R_i **do**
        $count(pNode) \leftarrow count(pNode) \times (end'_i - start'_i + 1)$
      end for
      $berNode \leftarrow$ new BernoulliNode(A, $\{a\}$)
      $pNode \rightarrow addChild(berNode)$
      for $B \in ch(A)$ **do**
        $chNode \leftarrow$ BUILDSPN(B, $(start'_i, end'_i)_{i \in \overline{1,n}}$, $ancVals \cup \langle A : a \rangle$)
        $pNode \rightarrow addChild(chNode)$
        $count(pNode) \leftarrow count(pNode) \times count(chNode)$
      end for
      $sNode \rightarrow addChild(pNode)$
      $count(sNode) \leftarrow count(sNode) + count(pNode)$
    end for
    if $key(A) \neq anc(A)$ **then**
      $cache_A[context] \leftarrow sNode$
    end if
    return $sNode$
  end if
end function

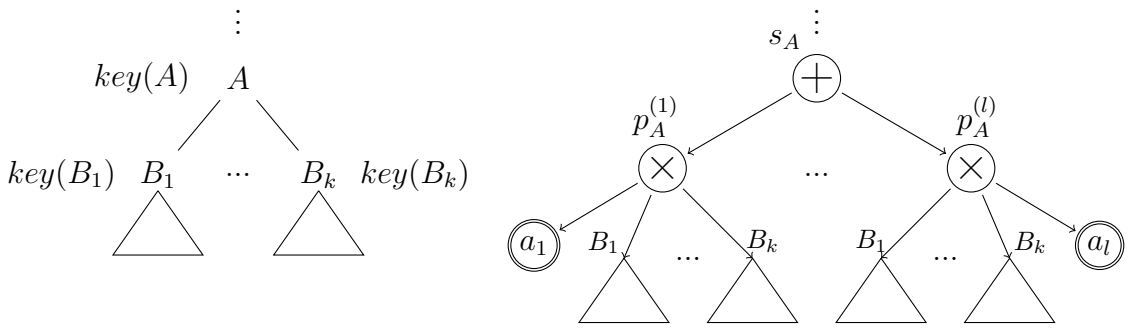


Figure 3.1: Sketch of the current step of Algorithm 1. The left hand side represents the part of the variable order used for the current step of the algorithm (the triangles represent the subtrees rooted at $B_1, ..., B_k$). The right hand side represents the sub-SSPN constructed in the current step of the algorithm. The triangles represent the sub-SSPNs that are the children of the product nodes (note that they may be just references). The double circled nodes are the Bernoulli nodes corresponding to each value a that A can map to.

algorithm takes as parameters the current node $A \in \mathcal{V}(\Delta)$, an array of ranges defined by *start* and *end* indices in each relation, and $ancVals$ mapping that keeps track of the current values each variable in $anc(A)$ maps to. Initially, A is the root of Δ, the ranges span the relations entirely, and $ancVals$ is empty.

We first check whether the sub-SSPN we are about to compute has been already computed. If so, we return a reference to it instead, where the key for the cache is just the context of A (i.e., the current mapping of the variables in $key(A)$). Caching is particularly useful when $key(A) \subset anc(A)$ (i.e., strictly contained), since this means that the sub-SSPNs over the variable order rooted at A are repeated for every distinct combination of values for the variables in $anc(A) \backslash key(A)$. This makes the nesting structure of SSPNs to be rooted directed acyclic graphs instead of rooted trees, and has a huge impact on the size of the resulting SSPN.

If A is a leaf node, then we construct a distribution node d_A using the values that A can map to in the current context (i.e., the current range for R_i) for estimating its parameters (discussed later). Note that at line 7, there is exactly one i such that $A \in schema(R_i)$ due to the restrictions imposed on Δ.

If A is an inner node (i.e., joining variable), then we first create a sum node s_A corresponding to variable A and its current context. For each mapping a in the intersection of possible A-values from the relations that contain attribute A we do the following. We first compute the new ranges that are narrowed down to those tuples with value a for attribute A. This can be done using a parallel iterators

style algorithm in time linear in the size of the smallest range (modulo number of relations factor). Note that the narrowed ranges are mutually exclusive for each value a that A can map to. We then create a product node p_A corresponding to the augmented context (i.e., the values each variable from $key(A)$ maps to, plus the fresh new mapping $\langle A : a \rangle$) that becomes a child of the sum node s_A (line 34). For each child B of A, we recurse using the new ranges and by setting the current value a of A in $ancVals$. The resulted sub-SSPN becomes a child of the product node p_A. At lines 27-28, we also add a Bernoulli node constructed from the value a to the children of the product node p_A. The purpose of this additional distribution node is to allow us to condition on the joining variable A when doing inference. The Bernoulli node simply returns 1 if $a \in \boldsymbol{\mathcal{X}}[A]$, and 0 otherwise. This gives us the additional power of taking into consideration only a subset of the sub-SSPNs when doing inference.

At the end, the distribution node d_A, or the sum node s_A, respectively, is added to the cache if that is the case (i.e., $key(A) \neq anc(A)$), and it is returned.

Lemma 3.3 (Scope)**.** *Let $sc_\Delta(A)$ be the set of RVs from the subtree rooted at node A in Δ (i.e., its leaves). Then, for any SSPN S, $sc(n) = sc_\Delta(A)$ for any node n in S corresponding to the variable $A \in \mathcal{V}(\Delta)$ (i.e., s_A, p_A, or d_A).*

Proof. The proof is based on structure induction on Δ. By definition, $sc(d_A) = \{A\}$ for any distribution node d_A corresponding to the leaf node A from Δ. By the induction hypothesis, $sc(p_A) = \bigcup_{c \in ch(p_A)} sc(c) = \bigcup_{B \in ch(A)} sc_\Delta(B) = sc_\Delta(A)$ for any product node p_A corresponding to value a of A from Δ. And finally, $sc(s_A) = \bigcup_{p_A \in ch(s_A)} sc(p_A) = sc_\Delta(A)$. □

Theorem 3.4 (Completeness)**.** *Given a variable order Δ, any SSPN S over Δ is a complete SPN over the RVs from $\mathcal{D}(\Delta)$.*

Proof. This follows immediately from Lemma 3.3 since $sc(p_A) = sc(p'_A) = sc_\Delta(A)$ for any children p_A and p'_A of any sum node s_A corresponding to variable $A \in \mathcal{J}(\Delta)$. □

Theorem 3.5 (Decomposability)**.** *Given a variable order Δ, any SSPN S over Δ is a decomposable SPN over the RVs from $\mathcal{D}(\Delta)$.*

Proof. By Lemma 3.3, we have that $sc(c_i) = sc_\Delta(B_i), \forall c_i \in ch(p_A)$ for any product node p_A corresponding to variable $A \in \mathcal{J}(\Delta)$. Since Δ is a tree, we have that $sc_\Delta(B_i) \cap sc_\Delta(B_j) = \varnothing, \forall B_i \neq B_j$, and thus finishing the proof. □

Corollary 3.6 (Validity)**.** *By Theorems 3.4 and 3.5, we have that, given a variable order Δ, any SSPN S over Δ is a **valid** SPN over the RVs from $\mathcal{D}(\Delta)$.*

Corollary 3.7 (Exact Inference)**.** *Hence, exact inference in S can be answered in time linear in $|S|$.*

The structure of SSPNs from Algorithm 1 resulted naturally from the conditional independence and factorisation modelled by the variable order Δ. Given the join query Q, two children B_1 and B_2 of A are conditionally independent given $key(A) \cup \{A\}$. This leads naturally to setting B_1's and B_2's corresponding sum nodes as children of the product node corresponding to the value a of A in the current context.

3.2 Weights Learning

Once the structure of an SPN is fixed, one needs to estimate the weights labelling each outgoing edge of a sum node. The most popular parameter learning algorithms that have been proposed are Gradient Descent (GD) [9, 30] and Expectation Maximisation (EM) [9, 29, 30].

3.2.1 Overview of Existing Methods

More precisely, given a valid SPN S over $\mathbf{X}$ and a training data set $\mathcal{D} = \{\mathbf{x}^1, .., \mathbf{x}^L\}$, the objective is to maximise the log-likelihood:

$$\begin{aligned} &\text{maximise } \log \mathcal{L} = \sum_{l=1}^{L} \log S(\mathbf{x}^l) \\ &\text{subject to } \sum_{c \in ch(s)} w_{s,c} = 1, \text{ for all sum nodes } s \\ &\qquad\qquad w_{s,c} \geqslant 0, \text{ for all sum nodes } s \text{ and } c \in ch(s) \end{aligned} \tag{3.1}$$

We assume without loss of generality that the sum and product nodes from S are arranged in alternating layers. The partial derivatives can be computed using backpropagation by first evaluating $S_n(\mathbf{x})$ in an upward pass, and then the derivatives in a downward pass as following:

$$\begin{aligned}\frac{\partial S}{\partial S_n}(\mathbf{x}) &= \begin{cases} \sum_{p \in pa(n)} \frac{\partial S}{\partial S_p}(\mathbf{x}) \prod_{k \in ch_{-n}(p)} S_k(\mathbf{x}) & \text{if } n \text{ is a sum node} \\ \sum_{p \in pa(n)} w_{p,n} \times \frac{\partial S}{\partial S_p}(\mathbf{x}) & \text{if } n \text{ is a product node} \end{cases} \\ \frac{\partial S}{w_{n,c}}(\mathbf{x}) &= \frac{\partial S}{\partial S_n}(\mathbf{x}) \times S_c(\mathbf{x}) \end{aligned} \tag{3.2}$$

where $ch_{-n}(p)$ are the children of parent p of n, excluding n itself. Using the above partial derivatives, one can also compute the marginals of all nodes (including the hidden variables corresponding to the sum nodes) [6].

Hence, the derivative of the log-likelihood from 3.1 is:

$$\frac{\partial \log \mathcal{L}}{\partial w_{n,c}} = \sum_{i=1}^{L} \frac{\partial \log S}{w_{n,c}}(\mathbf{x}^l) = \sum_{l=1}^{L} \frac{1}{S(\mathbf{x}^l)} \times \frac{\partial S}{\partial S_n}(\mathbf{x}^l) \times S_c(\mathbf{x}^l) \tag{3.3}$$

giving us the following steepest ascent update (where η is the step size):

$$\mathbf{w} \leftarrow \text{project}(\mathbf{w} + \eta \nabla \log \mathcal{L}) \tag{3.4}$$

where project means the projection of the new weights into the feasible region (i.e., positive and locally normalised weights). Alternatively, one can let unconstrained $Z_S = S(*)$ and optimise $P_S(\mathbf{x}) = \frac{S(\mathbf{x})}{S(*)}$ instead, but keep in mind that projection onto the positive axes is still necessary.

SPN weights can also be learned using Expectation Maximisation (EM) [7] by considering each sum node as the marginalisation of a hidden variable in a mixture model. The E step computes the marginals of the hidden variables corresponding to the sum nodes, and the M step updates the weights by adding each marginal value to its corresponding sum from the previous iterations and renormalising to obtain the new weights.

However, both methods suffer from the gradient diffusion problem, especially when learning deep networks. The gradient signal or the EM updates, respectively, may rapidly vanish as more layers are added to the network.

Therefore, Poon and Domingos [30] proposed to overcome this problem by using hard EM, i.e., by replacing marginal inference with Most Probable Explanation inference. MPE can be found by replacing the sum nodes with max nodes. We first do a bottom-up evaluation of the network, and then we start from the root and follow all children of the product nodes, and the (or a) child with the highest weighted value for the max nodes. Hard EM keeps track of the number

of times each weight edge was part of such an MPE path. The final weights are obtained by normalising the counts (i.e., $w_{s,c} = count_{s,c} / \sum_{c' \in ch(s)} count_{s,c'}$). This avoids the gradient diffusion problem since all updates are of unit sizes.

3.2.2 Computing Weights for SSPNs

Our approach for learning the weights uses the same idea from hard EM, yet in a completely different manner. Given a non-leaf node A from Δ, we can efficiently compute the size of the join query result represented by the factorised representation rooted at A. This query result represents the fraction of the data used for learning the sub-SSPN rooted at the sum node s corresponding to node A. This is further divided into smaller fractions of data that are used for learning the sub-SSPNs corresponding to each product child p. This leads naturally to the training data points that increment the count corresponding to the weight $w_{s,p}$ in the hard EM method.

Therefore, the weights of SSPNs are defined as:

$$w_{s,p} = \frac{count(p)}{\sum_{c \in ch(s)} count(c)} = \frac{count(p)}{count(s)}, \ \forall \text{ sum node } s \text{ and } p \in ch(s) \tag{3.5}$$

where $count(n)$ is the number of tuples in the join sub-query result from the subtree rooted at the corresponding node A of n from Δ.

Algorithm 1 shows how to compute these counts. If A is a leaf node, then the count of the corresponding distribution node is just the size of the range (line 9). When A is an inner node, the algorithm creates the sum node $sNode$, and a product node $pNode$ for each possible mapping a for A. The count of each $pNode$ is the product of the counts of its children (line 32). We also need to multiply this with the range sizes of all relations R_i that "end" in node A (i.e., A is the deepest attribute of R_i, lines 24-26). Finally, the count of $sNode$ is just the sum of the counts of its children (line 35).

Corollary 3.8 (Locally normalised)**.** *Since all weights of SSPN S are normalised, we have that S is locally normalised, and thus, $Z_S = 1$, or more importantly, $P_S(\mathbf{x}) = S(\mathbf{x})$.*

This has a significant impact in the efficiency of SSPNs since the time it takes to answer an inference query is now half the initial time (i.e., inference queries now require only one bottom-up evaluation of the network).

3.3 Distribution Parameters Learning

When Algorithm 1 inputs a leaf node A, it must create a distribution node that accurately models the corresponding fraction of training data (i.e., the current range for R_i). The distribution type used for this leaf is given by the user, however, its parameters need to be estimated. In order to estimate them accurately, we will use the Maximum Likelihood Estimation (MLE) method. This method estimates the probability distribution parameters by maximising a likelihood function, so that the observed data is most probable. In many cases the likelihood function is differentiable, and thus, the derivative test for determine the maxima can be applied.

3.4 Relational SSPNs

As discussed in the Introduction, relational DBMSs bring a lot of advantages, hence, our design decision aimed for a relational representation of our SSPNs in order to take advantage of these benefits.

Each sum node s_A or distribution node d_A from SSPN S is uniquely identified by the values each variable from $key(A)$ maps to. Similarly, each product node p_A from S is uniquely identified by the values each variable from $key(A) \cup \{A\}$ maps to. Building on this observation, we can represent S relationally by defining the following three sets of relations:

- $A_{distr}(key_A, \theta_1, \theta_2, ..., \theta_k)$, for each distribution variable $A \in \mathcal{D}(\Delta)$,
- $A_{sumcnt}(key_A, A_{sc})$, for each variable $A \in \mathcal{V}(\Delta)$,
- $A_{cnt}(key_A, A, A_c)$, for each variable $A \in \mathcal{V}(\Delta)$.

where key_A denotes the sequence of variables from $key(A)$ ordered increasingly by their depth in Δ (i.e., first parent and then child), where $\boldsymbol{\theta} = \theta_1, ..., \theta_k$ denotes the sequence of parameters of the distribution encoded by the variable A, and where A_{sc} and A_c denote the counts of the sum nodes and product nodes, respectively, that these two sets of relations represent.

Given a variable $A \in \mathcal{V}(\Delta)$, let K1, K2, ..., Km be the (possibly empty) sequence key_A, and let B1, B2, ..., Bk be the (possibly empty) sequence of children $ch(A)$. If A is a leaf node, then let R be the unique relation such that $A \in schema(R)$, otherwise let R1, R2, ..., Rt be the (possibly empty) sequence of relations that "end" in node A.

The three sets of tables can be computed using joins, group by's, and simple aggregates, following a reversed depth-first pre-order traversal of Δ. For each variable $A \in \mathcal{V}(\Delta)$ following that order, we have that if A is a leaf node, then:

Algorithm 2 Computes the count table corresponding to leaf node A

```
CREATE TABLE A_cnt AS
SELECT K1, ..., Km, A, COUNT(*) AS A_c
FROM R
GROUP BY K1, ..., Km, A;
```

And if A is an inner node, then:

Algorithm 3 Computes the count table corresponding to inner node A

```
CREATE TABLE A_cnt AS
SELECT K1, ..., Km, A, SUM(B1_sc * ... * Bk_sc) AS A_c
FROM B1_sumcnt NATURAL JOIN ... NATURAL JOIN Bk_sumcnt
     NATURAL JOIN R1 ... NATURAL JOIN Rt
GROUP BY K1, ..., Km, A;
```

The sum count table can be simply computed in both cases as:

Algorithm 4 Computes the sum count table corresponding to node A

```
CREATE TABLE A_sumcnt AS
SELECT K1, ..., Km, SUM(A_c) AS A_sc
FROM A_cnt
GROUP BY K1, ..., Km;
```

In case A is a leaf node, we also need to compute the distribution table. This can be done using aggregates that match the formulas of the MLE parameters.

For example, if A encodes a Categorical distribution, then the probability of each category is just its normalised frequency, thus:

Algorithm 5 Computes the Categorical distribution table corresponding to A

```
CREATE TABLE A_distr AS
SELECT K1, ..., Km, A, (A_c / A_sc) AS _freq
FROM A_cnt NATURAL JOIN A_sumcnt;
```

If A encodes a Gaussian distribution, then its MLE parameters are the mean and standard deviation of the training data, thus:

Algorithm 6 Computes the Gaussian distribution table corresponding to A

```
1: CREATE TABLE A_distr AS
2: SELECT K1, ..., Km, AVG(A) AS _mean, VAR_POP(A) AS _var
3: FROM R
4: GROUP BY K1, ..., Km;
```

Inference queries can be computed in a similar manner using the three sets of relations we have just created. Following the reversed depth-first pre-order traversal of Δ, we compute a temporary relation $A_{tmp}(key_A, A_p)$ for each variable $A \in \mathcal{V}(\Delta)$ that keeps track of the intermediate probability value of the sub-SSPN rooted at the corresponding sum node s_A.

Given partial evidence $\boldsymbol{\mathcal{X}}$, we have that if A is a leaf node, then the value of attribute A_p must equal $p_A(\boldsymbol{\mathcal{X}})$ (i.e., the result of marginalising the distribution at node d_A with respect to $\boldsymbol{\mathcal{X}}$). This can be accomplished using pre-defined aggregate functions that compute the probability mass function (pmf) or cumulative density function (cdf) of different probability distributions.

For example, if A encodes a Categorical distribution and $\boldsymbol{\mathcal{X}}[A] = \{a, b\}$, then

Algorithm 7 Temporary table corresponding to Categorical RV A

```
1: WITH
2: ...
3: A_tmp AS (
4:     SELECT K1, ..., Km, SUM(_freq) AS A_p
5:     FROM A_distr
6:     WHERE A IN (a, b)
7:     GROUP BY K1, ..., Km
8: ),
9: ...
```

If A encodes a Gaussian distribution and $\boldsymbol{\mathcal{X}}[A] = (a, b), a < b$, then we use gaussian_getCDF() pre-defined function to compute $cdf(x) = \frac{1}{2}[1 + \text{erf}(\frac{x-\mu}{\sigma\sqrt{2}})]$:

Algorithm 8 Temporary table corresponding to Gaussian RV A

```
1: WITH
2: ...
3: A_tmp AS (
4:     SELECT K1, ..., Km, gaussian_getCFD(_mean, _var, b) -
       ↪ gaussian_getCDF(_mean, _var, a) AS A_p
5:     FROM A_distr
6: ),
7: ...
```

In the trivial case when variable A is not conditioned at all in partial evidence $\boldsymbol{\mathcal{X}}$, the attribute A_p must be set to 1, and thus:

Algorithm 9 Temporary table corresponding to unconditioned RV A

```
WITH
...
A_tmp AS (
    SELECT [DISTINCT] K1, ..., Km, 1 AS A_p
    FROM A_distr
),
...
```

where the DISTINCT keyword must be used if A encodes a Categorical distribution because there are several tuples that define a distribution node (i.e., one per category) and we need only one tuple per distribution node in the resulting temporary table.

If A is an inner node, the value of the sub-SSPN rooted at sum node s_A is then the weighted sum of the product children, where the value of each product child is the product of the values of its sum children:

Algorithm 10 Temporary table corresponding to inner node A

```
WITH
...
A_tmp AS (
    SELECT K1, ..., Km,
           SUM(B1_p * .. * Bk_p * (A_c / A_sc)) AS A_p
    FROM A_cnt NATURAL JOIN A_sumcnt
         NATURAL JOIN B1_tmp ... NATURAL JOIN Bk_tmp
    [WHERE A IN (...)]
    GROUP BY K1, ..., Km
),
...
```

where the WHERE clause must be added if the joining variable A is constrained in the partial evidence $\boldsymbol{\mathcal{X}}$. In the end, the value of the inference query is the value of the probability attribute corresponding to the root variable of Δ.

The same approach as the one described above can be used in order to compute the SQL script that answers conditional probability or most probable explanation (MPE) queries.

3.5 Maintaining SSPNs under Updates

As discussed in the introduction, SSPNs have the ability to be efficiently maintained under updates (tuple insertions and deletions) to the input database. We have seen above that the construction of SSPNs S is divided into three components. The network structure and its weights are represented relationally by the count and sum count tables. The parameters of the probability distributions found at the leaves of S are represented relationally by the distribution tables. We see next how to efficiently maintain each of these three sets of tables under tuple updates.

Suppose that one wants to insert or delete a tuple from relation $R_i(A^{(1)}, ..., A^{(r)})$ that is part of the join query Q. The attributes $A^{(1)}, ..., A^{(r)}$ are ordered increasingly by their depth in Δ (i.e., first parent and then child).

3.5.1 Network Structure and Weights Maintenance

Analysing the method SSPNs S are constructed, we can determine the minimal changes required to maintain S up to date. More precisely, the only relations that need to be updated are $A_{sumcnt}(key_A, A_{sc})$ and $A_{cnt}(key_A, A, A_c)$ for all variables $A \in \mathcal{V}(\Delta)$ that are ancestors of $A^{(r)} \in schema(R_i)$ (including $A^{(r)}$). Moreover, the only tuples from these relations that need to be updated are those that agree on the common attributes from $\{A^{(1)}, ..., A^{(r)}\}$. Therefore, we can maintain these relations by following the reversed depth-first pre-order traversal of Δ and updating the tuples that agree on $A^{(1)}, ..., A^{(r)}$ from the above relations.

If A is a leaf node, then updating $A_{sumcnt}(key_A, A_{sc})$ and $A_{cnt}(key_A, A, A_c)$ means incrementing (if one performs an insertion) or decrementing (if one performs a deletion) the attributes A_{sc} and A_c.

Otherwise, updating $A_{sumcnt}(key_A, A_{sc})$ and $A_{cnt}(key_A, A, A_c)$ means recalculating the attributes A_{sc} and A_c using the corresponding formulas described in the previous section.

3.5.2 Distribution Parameters Maintenance

When $A^{(r)}$ is a leaf node, we also need to update the tuples from the distribution table $A^{(r)}_{distr}(key_{A^{(r)}}, \theta_1, ..., \theta_k)$ that agree on $key(A^{(r)}) = \{A^{(1)}, ..., A^{(r-1)}\}$. Distribution tables can be efficiently maintained under updates by keeping track of intermediate sum and count aggregates from which the corresponding MLE pa-

rameter formulas can be evaluated. Many well known probability distributions can be maintained in time O(1).

For example, the MLE parameters for Categorical distribution are the frequency of each category. Instead of keeping track of the exact frequencies, we can keep track of the count of each category and the total count. The frequency is then the corresponding count divided by the total count, and an update consists of just two incrementations or decrementations to the count corresponding to $A^{(r)}$ and to the total count.

The MLE parameters for Gaussian distribution are the mean $\mu = \frac{1}{n}\sum_{i=1}^{n} x_i$ and the standard deviation $\sigma^2 = \frac{1}{n}\sum_{i=1}^{n}(x_i-\mu)^2 = \frac{1}{n}\sum_{i=1}^{n}(x_i^2+\mu^2-2x_i\mu)$. Thus, by keeping track of n, $\sum_{i=1}^{n} x_i$, and $\sum_{i=1}^{n} x_i^2$, one can compute μ and σ^2 in O(1). Moreover, these three values can be easily updated in O(1) when inserting or deleting a new data point x.

Using the same approach, we can easily maintain the mean μ MLE parameter for Poisson distribution and the rate $\lambda = \frac{n}{\sum_{i=1}^{n} x_i}$ MLE parameter for Exponential distribution in O(1).

3.6 Discussion

At the beginning of the chapter we have imposed a couple of restrictions to the variable orders Δ used for SSPNs. The first restriction constrains the attributes of the relations to lay down in their natural order on Δ (i.e., the joining variables before the distribution ones). As stated, this is natural and is not much of a constraint. However, the second restriction allows only for relations with exactly one distribution variable (which is the leaf of the corresponding path in Δ). We overcome this restriction by decomposing the relations with more than one distribution variables into multiple relations, each having exactly one distribution variable. Basically, we make these distribution variables siblings hanging at the bottom of the corresponding path from Δ. However, this decomposition yields to wrong counts used for computing the weights. Therefore, we adjusted the algorithm to take into consideration this decomposition, and thus, setting the corresponding count to the actual number of distribution variable data points, instead of raising this number to the power of the number of siblings, which is what the initial algorithm would do.

Example 3.9. Figure 3.2 depicts the SSPN S constructed over the variable order Δ taken from Example 2.1 in the FDB preliminary chapter. The joining

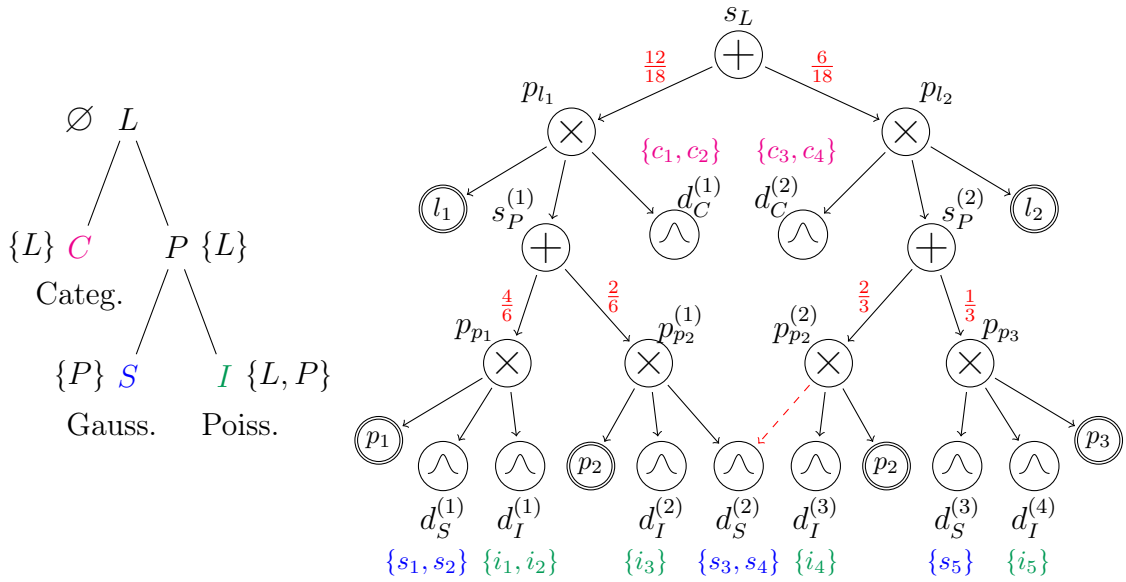


Figure 3.2: SSPN S (right hand side) constructed over the variable order Δ (left hand side) from Figure 2.1.

variables are L and P, while the distribution ones are C, S, and I, each having its distribution type. The structure of S follows the Algorithm 1, the double circled nodes being the helping Bernoulli distribution nodes (each with their own value), and the nodes with the bell shaped function inside the circle being the distribution nodes corresponding to variables C, P, and I. Next to each distribution node, it is depicted the set of values from which the corresponding MLE parameters are computed. We have that $count(d_S^{(1)}) = count(d_I^{(1)}) = 2$, and thus, $count(p_{p_1}) = 2 \times 2 = 4$. Similarly, $count(d_I^{(2)}) = 1$ and $count(d_S^{(2)}) = 2$, and thus, $count(p_{p_2}^{(1)}) = 1 \times 2 = 2$. By these two, we have that $count(s_P^{(1)}) = 4 + 2 = 6$, and thus, $w_{s_P^{(1)}, p_{p_1}} = \frac{4}{6}$ and $w_{s_P^{(1)}, p_{p_2}^{(1)}} = \frac{2}{6}$. The rest of the counts and weights are computed in the same way. Also, in the variable order Δ we have that $key(S) = \{P\} \neq \{L, P\} = anc(S)$. Therefore, some sharing occurs and it can be seen in Figure 3.2 as the dotted red edge between $p_{p_2}^{(2)}$ and $d_S^{(2)}$.

In the relational representation of S, the following tables are computed:

- $I_{distr}(L, P, mean)$, $I_{cnt}(L, P, I, I_c)$, and $I_{sumcnt}(L, P, I_{sc})$,
- $S_{distr}(L, P, mean, var)$, $S_{cnt}(L, P, S, S_c)$, and $S_{sumcnt}(L, P, S_{sc})$,
- $P_{cnt}(L, P, P_c)$ and $P_{sumcnt}(L, P_{sc})$,
- $C_{distr}(L, C, freq)$, $C_{cnt}(L, C, C_c)$, and $C_{sumcnt}(L, C_{sc})$,
- $L_{cnt}(L, L_c)$ and $L_{sumcnt}(L_{sc})$.

Chapter 4

Scripting Language for SSPNs Modelling

As mentioned in Introduction, apart from introducing the theory behind SSPNs and implementing the system that builds the relational representation of them, we have also created a user-friendly scripting language for modelling SSPNs. This is basically another level of abstraction for our system which helps the user prepare the training data, define the variable order Δ, define the inference queries, and finally run the experiment much more easily.

The workflow is as follows. The system first parses the specification script and then calls the corresponding back-end functions with the corresponding arguments. These functions will generate several SQL files that define the relational representation of SSPN S and the inference queries described in the specification. Then, it connects to the DBMS engine and runs the corresponding SQL files. The SQL code corresponding to inference queries is generated so that the results together with the values of the conditioned variables are inserted into a table. At the end, the system also inserts the run time statistics into a table.

Figure 4.1 depicts the grammar used for the Scripting Language. Figure 4.2 depicts a specification script example for the SSPN S from Figure 3.2. Between lines 8 and 24 small examples are shown for each of the data manipulation statements our system supports.

A typical session in our system runs as follows. We start by providing the schema of all tables (lines 1-5). We then do some data operations. We then set each of the variables a distribution type, where joining variables must be of type Bernoulli (lines 26-29). We then construct the variable order and the SPN (lines 31-36). We finally define the queries (lines 38-43).

```
1:  spec ::= "Schema:" schema "Definitions:" def* execute?
2:
3:  schema ::= relation+
4:  def ::= decompose | drop | join | cluster | project | pdf | var_order |
    ↪  spn | bucket | inference | mpe
5:
6:  decompose ::= relation {", " relation}+ "= decompose" rel_name
7:  drop ::= rel_name "= drop" var "from" rel_names
8:  join ::= relation "= join" rel_name ", " rel_name
9:  cluster ::= rel_name "= cluster" rel_name "as" var "on" vars "using" num
    ↪  "means"
10: project ::= relation "= project" rel_name
11:
12: pdf ::= dist_type ":" vars;
13: var_order ::= vo_name "= construct variable order from" term "using
    ↪  relations" rel_names
14: term ::= var {"(" {term ","}+ ")"}?
15:
16: spn ::= spn_name "= construct spn with variable order" vo_name
17:
18: bucket ::= "bucket:" var "[" (set_value ",")+ "]"
19:
20: inference ::= query_name "= query" spn_name {"from" db_name "."
    ↪  rel_name}? ": P(" events {"|" events}? ")"
21: mpe ::= query_name "= query" spn_name {"from" db_name "." rel_name}? ":
    ↪  MPE(" var {"|" events}? ")"
22:
23: events ::= {{bin_condition | set_condition} ","}+
24: bin_condition ::= var {"<" | ">" | "="} value
25: set_condition ::= var {"in" | "between"} set_value
26: value ::= double | "'" string "'" | col_name
27: set_value ::= "[" {value ","}+ "]" | col_name
28:
29: execute ::= "execute" db_name {"out =" filename}? "statistics"?
30:
31: relation ::= rel_name vars
32: vars ::= "(" {var ","}+ ")"
33: dist_type ::= "bernoulli" | "categorical" | "gaussian" | "poisson" |
    ↪  "exponential"
34: rel_names ::= "(" {rel_name ","}+ ")"
35: rel_name, vo_name, spn_name, query_name, col_name, db_name, filename,
    ↪  var ::= string
36: num ::= int
```

Figure 4.1: The grammar for the Scripting Language.

```
1:  Schema:
2:  Branch(L, P, I)
3:  Competition(L, C)
4:  Sales(P, S)
5:  testset(I)
6:
7:  Definitions:
8:  /*
9:  Rel. R(J1, J2, D1, D2) with multiple distr. vars. must be decomposed:
10: R1(J1, J2, D1), R2(J1, J2, D2) = decompose R
11:
12: We can drop a variable.
13: Relation R becomes the natural join of R1, R2, R3, with J dropped:
14: R = drop J from (R1, R2, R3)
15:
16: We can join two relations:
17: R(A, B, C) = join R1, R2
18:
19: We can use k-means to cluster some variables:
20: R_cl = cluster R as K on (A, B, C) using 5 means
21:
22: We can project away attributes from relation R(A, B, C):
23: R_p(A, B) = project R
24: */
25:
26: bernoulli: (L, P)
27: categorical: (C)
28: gaussian: (S)
29: poisson: (I)
30:
31: vo_ex = construct variable order from
32:     L(C, P(S, I))
33: using relations
34:     (Branch, Competition, Sales)
35:
36: spn_ex = construct spn with variable order vo_ex
37:
38: inference1 = query spn_ex: P(C in ['c1', 'c2'], S < 2)
39: inference2 = query spn_ex: P(C = 'c1' | S < 2, I between [2, 5])
40:
41: bucket: C [['c1'], ['c2'], ['c3'], ['c4']]
42: mpe_query1 = query spn_ex: MPE(C | I < 5)
43: mpe_query2 = query spn_ex from mydb.testset: MPE(C | I = I)
44:
45: execute mydb statistics
```

Figure 4.2: The specification script used for the example from Figure 3.2.

We can define inference queries (with or without evidence - lines 38-39) or MPE queries (lines 42-43). For an MPE query we also need to provide the value buckets. The system issues a query per bucket and the one with the highest probability is the answer of the MPE query. We can also use a database table to issue multiple queries with the same definition (i.e., a query per tuple), provided that the table's columns match the variables we condition on (line 43). We can also execute everything by just one command (line 45). The system automatically creates a table per query definition (with the same name as the query name) where the results will be inserted. It also creates a runtime statistics table where it inserts the runtimes for each SPN construction and each query.

Our system connects to PostgreSQL and runs the queries that define SSPN S, but due to the fact that all generated queries are written in the Standard Query Language (SQL), our system can effectively run in any DBMS on any platform.

Chapter 5

Experiments

We have used the real MovieLens-100K [12] data set to test the accuracy of SSPNs. The data set consists of 100,000 ratings on 1,682 movies by 943 users. Each rating is an integer between 1 and 5. More details about the data set can be seen in Figure 5.1.

5.1 Summary of Findings

The task is to predict unknown ratings given pairs of UserIDs and MovieIDs. We evaluate the models in terms of accuracy (MAE and RMSE metrics) and runtime performance (wall-clock time). We compare ourselves against the best known models for the given task and also against SPFlow, a state-of-the-art system for SPNs. Our experimental findings are summarised as follows:

- Our proposed SSPN system outperformed 8 out of 10 competitor models in terms of the MAE accuracy metric, and 9 out of 10 models in terms of the RMSE accuracy metric. Moreover, our system reported significantly better accuracy results than the state-of-the-art SPFlow system for this data set.
- In terms of runtime performance for learning the model, SSPN system was three orders of magnitude faster than the competitor SPFlow system, even though the input database for SSPN system was much bigger due to the incorporated similarity knowledge between users and movies.
- Even though the size of the SSPN network is two orders of magnitude bigger than the size of the SPFlow learned network, the average runtimes for answering one expected rating query are rather similar: SPFlow is just 1.095 times faster than our system.

MovieLens-100K data set	
No. of users	943
No. of movies	1682
No. of ratings	100000
No. of ratings per user	106.04
No. of ratings per movie	59.45
Rating Sparsity	93.7%

User Features	Movie Features
UserID	MovieID
Age	Title
Gender	Genres
Occupation	

Figure 5.1: Description of MovieLens-100k data set. Rating Sparsity is the portion of the missing entries from the full ratings matrix.

5.2 Evaluation Metric

We use the Mean Absolute Error (MAE) and the Root Mean Square Error (RMSE) to evaluate the accuracy of the predicted ratings compared to the actual ratings:

$$\begin{aligned} MAE =& \frac{1}{|\mathcal{T}|} \sum_{r_{u,m} \in \mathcal{T}} |r_{u,m} - \hat{r}_{u,m}| \\ RMSE =& \sqrt{\frac{1}{|\mathcal{T}|} \sum_{r_{u,m} \in \mathcal{T}} (r_{u,m} - \hat{r}_{u,m})^2} \end{aligned} \tag{5.1}$$

where $\hat{r}_{u,m}$ is the predicted rating of user u for movie m, and $r_{u,m}$ is the actual rating from the testing set $\mathcal{T}$, and where $|\mathcal{T}|$ is the size of $\mathcal{T}$ (i.e., the number of UserID and MovieID pairs). Smaller values of MAE and RMSE means better performance. Note that RMSE is more sensitive to large errors than MAE as it computes the sum of square and not absolute values.

5.3 Competitor Algorithms

We compare SSPNs with the state-of-the-art discriminative models from the Simple Python Recommendation System Engine (Surprise) [14] website: the SVD algorithm [19], an extension called SVD++, which takes into account implicit ratings [17], NMF algorithm, which is a collaborative filtering algorithm based on Non-negative Matrix Factorisation [21], Slope One algorithm [20], k-NN, Centered k-NN, and k-NN Baseline, which are several variants of the k-nearest neighbours algorithm [18], Co-Clustering, which is a collaborative filtering algorithm based on co-clustering [11], and Baseline, which is the baseline estimate computed

Algorithm	MAE	RMSE
SVD	0.737	0.934
SVD++	**0.722**	**0.92**
NMF	0.758	0.963
Slope One	0.743	0.946
k-NN	0.774	0.98
Centered k-NN	0.749	0.951
k-NN Baseline	**0.733**	0.931
Co-Clustering	0.753	0.963
Baseline	0.748	0.944
Random	1.215	1.514
SPFlow	0.9566	1.1435
SSPN (ours)	**0.735**	**0.930**

Figure 5.2: MAE and RMSE performance comparison of the algorithms from the Surprise website [14].

from the mean and the biases of each user and movie [18].

We also compare the SSPN system with the state-of-the-art SPN model: Mixed Sum-Product Networks (or MSPNs for short) [22] using their own Python library called SPFlow [23].

5.4 Experimental Setup

We run all experiments on an Intel(R) Core(TM) i5-6500 CPU @ 3.20GHz, 32GB RAM, with Ubuntu 18.04.4 LTS computer. We use PostgreSQL 10.12 for SSPN system and Python 3.6.9 for SPFlow system.

To evaluate the performance of the algorithms we used the 5-fold cross-validation procedure. The ratings table was split into 5 random sets, each having 20,000 rating tuples, and 5 experiments were conducted, each having 80,000 rating tuples in the training data set and 20,000 rating tuples in the testing dataset. We use the Root Mean Square Error (RMSE) from above for computing the error bars (i.e., to measure the differences between different runs).

5.5 Experimental Results

Table 5.2 shows that Structured Sum-Product Networks have better performance than all but two state-of-the-art discriminative models when using the MAE accuracy metric, and that we have better performance than all but one competitor

model when using the RMSE metric. More precisely, the MAE and RMSE values of SSPNs on the MovieLens-100K data set are 0.735 ± 0.006 and 0.930 ± 0.007, respectively.

This is an outstanding result taking into consideration that the competitor algorithms are the state-of-the-art discriminative models that are specialised for predictions in recommender systems. The only task these systems can solve is to predict the rating given a pair of existing UserID and MovieID. In contrast, our approach is much more general and can answer not only the most probable rating given an existing pair of UserID and MovieID, but also other kind of inference or conditional probability queries. After all, SSPNs model the entire join probability distribution in the input data and treat all input variables as random variables, and not only the rating label such as for discriminative models. In particular, SSPNs can also predict ratings for new users or movies that don't exist in the data by conditioning on their features (i.e., age, gender, genres, etc.) rather than on their (non-existent) IDs. Another query that is possible with SSPNs, but not with the above models, is, for example, computing the age distribution for users who give ratings above 3 to science fiction movies.

5.6 Model Accuracy

To obtain the results above, we built an ensemble of two SSPNs S_1 and S_2 over the variable orders Δ_1 and Δ_2, respectively. We also added knowledge by computing some similarity metrics between users and movies as additional two relations that were used as input for SSPN learning together with the original relations. SSPN S_1 integrates the similarities between users, while SSPN S_2 integrates the similarities between movies. The weight of S_1 is $1 - \lambda$, while the weight of S_2 is λ, where the parameter $0 \leqslant \lambda \leqslant 1$ needs to be learned from the training data set. The prediction is then the weighted sum of the predictions of the two SSPNs, rounded to the first natural number if required. Basically, our SSPNs ensemble is a linear regression model, where the features are computed using non-linear models (in this case SSPNs). In both Δ_1 and Δ_2, we set leaf *age* to be Gaussian, and the others to be Categorical. More details about the variable orders Δ_1 and Δ_2 can be seen in Figure 5.3.

We tried several similarity measures for our model, such as the Cosine similarity (Equation 5.2) and the Mean Square Difference (MSD) similarity (Equation 5.4). However, both of them give poor results on the MovieLens-100K data set

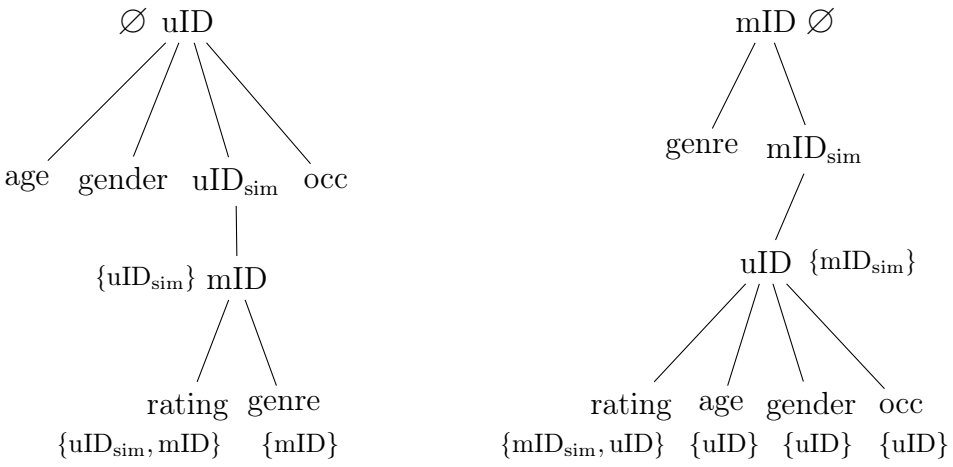


Figure 5.3: The variable orders Δ_1 (left) and Δ_2 (right) over which SSPNs S_1 and S_2 were constructed. All variables from the second layer have the root as their key in both Δ_1 and Δ_2.

in terms of both MAE and RMSE.

$$\begin{aligned}\text{cosine\_sim}(u_1, u_2) =& \frac{\sum_{m \in M_{u_1,u_2}} r_{u_1,m} \times r_{u_2,m}}{\sqrt{\sum_{m \in M_{u_1,u_2}} r_{u_1,m}^2} \times \sqrt{\sum_{m \in M_{u_1,u_2}} r_{u_2,m}^2}} \\ \text{cosine\_sim}(m_1, m_2) =& \frac{\sum_{u \in U_{m_1,m_2}} r_{u,m_1} \times r_{u,m_2}}{\sqrt{\sum_{u \in U_{m_1,m_2}} r_{u,m_1}^2} \times \sqrt{\sum_{u \in U_{m_1,m_2}} r_{u,m_2}^2}}\end{aligned} \tag{5.2}$$

$$\begin{aligned}msd(u_1, u_2) =& \frac{1}{|M_{u_1,u_2}|} \sum_{m \in M_{u_1,u_2}} (r_{u_1,m} - r_{u2,m})^2 \\ msd(m_1, m_2) =& \frac{1}{|U_{m_1,m_2}|} \sum_{u \in U_{m1,m2}} (r_{u,m_1} - r_{u,m_2})^2\end{aligned} \tag{5.3}$$

with the MSD similarity defined as:

$$\begin{aligned}\text{msd\_sim}(u_1, u_2) =& \frac{1}{msd(u_1, u_2) + 1} \\ \text{msd\_sim}(m_1, m_2) =& \frac{1}{msd(m_1, m_2) + 1}\end{aligned} \tag{5.4}$$

where M_{u_1,u_2} is the set of movies both u_1 and u_2 have rated in the training data set, and U_{m_1,m_2} is the set of users that have rated both m_1 and m_2 in the training data set.

We noticed that, when using the above similarity measures, our model tends to predict the average ratings for the users and the movies, respectively.

Therefore, we proposed a third similarity metric (Equation 5.5) that tries to

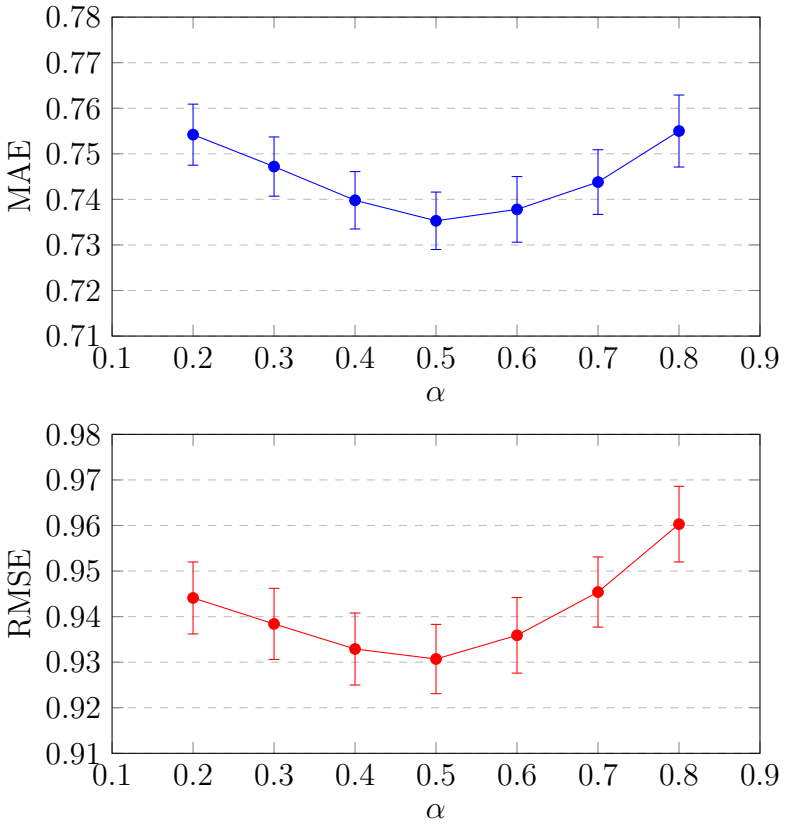


Figure 5.4: MAE and RMSE performance when we vary the parameter α and the optimal value $\lambda = 0.6$ is set.

offer much more weight to those users u_2 (or movies m_2, respectively) that are more similar to u_1 (or m_1, respectively):

$$\text{users\_sim}(u_1, u_2) = \begin{cases} Eq(u_1, u_2) & \text{if } Eq(u_1, u_2) > \alpha \times Diff(u_1, u_2) \\ 0 & \text{otherwise} \end{cases} \tag{5.5}$$

where $Eq(u_1, u_2)$ and $Diff(u_1, u_2)$ are the number of movies users u_1 and u_2 have rated the same and different, respectively:

$$\begin{aligned} Eq(u_1, u_2) &= |\{m \in M_{u_1,u_2} : r_{u_1,m} = r_{u_2,m}\}| \\ Diff(u_1, u_2) &= |\{m \in M_{u_1,u_2} : r_{u_1,m} \neq r_{u_2,m}\}| \end{aligned} \tag{5.6}$$

with $\text{movies\_sim}(m_1, m_2)$ defined analogous, and where the parameter α needs to be learned from the training data set.

By varying both similarity parameter α and the ensemble weights parameter λ, and by computing the average and root mean square error for the 5-fold cross

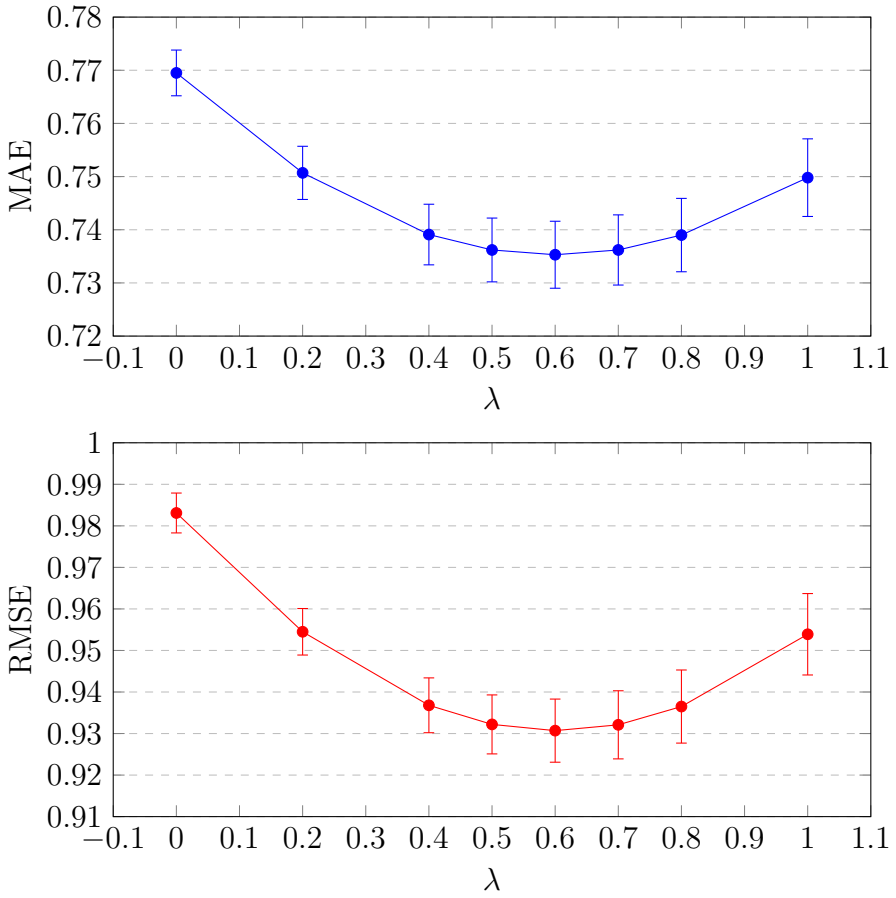


Figure 5.5: MAE and RMSE performance when optimal value $\alpha = 0.5$ is set and we vary the parameter λ.

validation, we obtained that the best values for these parameters are $\alpha = 0.5$ and $\lambda = 0.6$ for both MAE and RMSE accuracy metrics. That is, for a pair (u_1, u_2), we take into account the number of movies u_2 rated the same as u_1 if and only if this number is at least a third from the total number of movies u_1 and u_2 have rated. And analogous for the movies. And we give a contribution of 0.4 to the user similarity SSPN (i.e., S_1), and a contribution of 0.6 to the movie similarity SSPN (i.e., S_2). Intuitively, it is correct to give slightly more contribution to the movie similarity SSPN S_2 as the task is to give ratings about movies and so ratings given to similar movies are more important than the ratings given by the similar users.

Figure 5.4 depicts the MAE and RMSE values when we vary the parameter α and the optimal value $\lambda = 0.6$ is set. Figure 5.5 depicts the MAE and RMSE values when the optimal value $\alpha = 0.5$ is set and we vary the parameter λ.

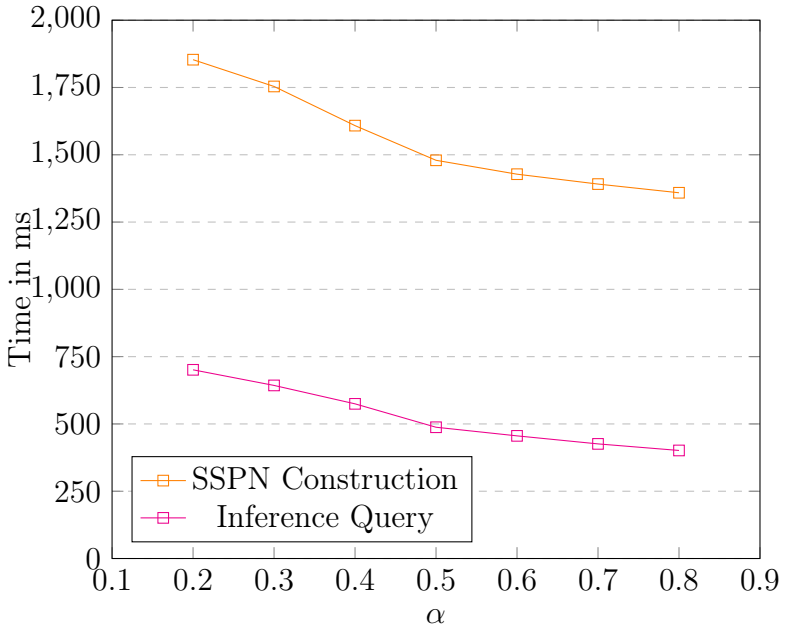


Figure 5.6: Time performance when varying the parameter α.

5.7 Runtime Performance

The choice of the similarity measure from Equation 5.5 has also an impact on the run time performance. The pairs for which the similarity is 0 are not stored in the tables, and thus, as the parameter α increases, more similarity pairs become 0, and so the sizes of the SSPNs decrease. Figure 5.6 shows the time performance of our model when varying α. In particular, for the optimal model (i.e., $\alpha = 0.5$), the accumulated SSPNs S_1 and S_2 construction times is 1479.51 millisecond, and the average time spent on answering one query is 487.72 milliseconds. It is worth mentioning that we didn't use any indexes on the tables. This is likely to further improve the runtime performance and is subject to future work.

5.8 Network Size

When the optimal value $\alpha = 0.5$ is set, the users similarity table has around 300,000 tuples, the movies similarity table has around 625,000 tuples, and the ratings table has 80,000 tuples. Taking into account the 943 users and the 1,682 movies, together with their features, the total size of our model is 1.1 million nodes broke down as:

- $|S_1| = 383,077$ nodes with 3,568 sum nodes, 294,998 product nodes, and

84,511 leaves, where S_1 is the user similarity based SSPN,

- $|S_2| = 717,093$ nodes with 4,307 sum nodes, 628,275 product nodes, and 84,511 leaves, where S_2 is the movie similarity based SSPN.

The optimisation induced by variable orders by caching the sub-SSPNs rooted at sum nodes s_A whenever $key(A) \subset anc(A)$ (i.e., strictly contained) is crucial here. The variables *mID*, *rating*, and *genre* from Δ_1, and the variables *uID*, *rating*, *age*, *gender*, and *occ* from Δ_2 have their key sets strictly contained in their ancestor sets, and thus, all sub-SSPNs corresponding to these variables are cached. If we didn't cache and didn't reuse these sub-SSPNs, the total size of our model would have been around 350 million nodes.

5.9 SPN Competitor

We also trained and tested a state-of-the-art SPN model called Mixed Sum-Product Networks [22]. We computed the full join of the user features, movie features, and the ratings tables. We one-hot encoded the Occupation and Genres tables. We trained the MSPN setting the UserID and MovieID as Discrete type, the Rating and Age as Real type, and the Gender together with all Occupation and Genres one-hot features as Binary type. Whereas for SSPN we consider an ensemble of two models, SPFlow constructs their model solely based on the materialised natural join of the input tables (as expected by that system). By joining with the similarities tables as well, the query result size would be around 140 million tuples, each with over 40 attributes. This is impractical for SPFlow, and thus, incorporating similarities to their model is unfeasible.

Even though the training data set has 80,000 rows and over 40 features, the size of the computed MSPN is rather small: 6,252 total nodes with 497 sum nodes, 1,772 product nodes, and 3,983 leaves. The MAE and RMSE values of this model are 0.956 and 1.143, respectively. Undoubtedly, our SSPNs significantly outperformed the MSPN model on the MovieLens-100K data set in terms of both MAE and RMSE, with our scores being just 0.735 and 0.930, respectively.

As stated in the Introduction chapter, existing SPN systems including SPFlow report significant runtimes as they do not exploit the existing structure of the underlying data expressed by the join relationships and set out to re-discover it using expensive clustering and independence testing (time quadratic in the size of the join result). In contrast, for acyclic queries like the MovieLens dataset, SSPNs

are constructed in time linear in the input database size (including the additional relations storing the similarities between users and movies). The runtime for constructing the MSPN was 469 seconds, which is indeed much slower compered to our SSPN construction time of just 1.48 seconds.

Even though the network sizes are of different magnitude order (i.e., 6,252 nodes compared to 1.1 million nodes), the average times of one inference query are rather similar: 445.34 milliseconds (for MSPN) compared to 487.72 milliseconds (for SSPN - ours). Hence, SSPN system has two orders of magnitude better throughput than SPFlow, where the throughput measure is computed as inference time per network size. This remarkable result validates our initial design decision of implementing a fully relational system for modelling SSPNs.

Chapter 6

Conclusion

6.1 Summary

In this thesis, we introduced a new class of structure-aware relational Sum-Product Network models learned over multi-relational databases called Structured Sum-Product Networks. We put forward a novel approach to computing the structure and the parameters of these models by exploiting the semantics and structure of the underlying multi-relational databases induced by the join relationships and possible orders on the join variables. This was the key conjecture of the thesis that had a significant impact on the runtime performance and accuracy of SSPNs. We introduced the relational representation of SSPNs, and put forward algorithms that translate SSPN learning and inference to plain SQL queries. We implemented a fully relational framework, together with a user-friendly scripting language, that aid the development of SSPN models. We also explained how SSPNs can efficiently be maintained under input data updates, as, in contrast to the general approach, is possible with our models due to the relational representation.

The experiments validated the reasons behind the decision to investigate structure-aware learning and relational representation of SSPNs. Our proposed model outperformed 8 out of 10 (when using the MAE metric) and 9 out of 10 (when using the RMSE metric) best known discriminative models for the MovieLens-100K data set. Moreover, SSPNs reported significantly better accuracy results than the state-of-the-art SPFlow system in terms of both MAE and RMSE metrics. Last but not least, our relational system for SSPNs was three orders of magnitude faster than SPFlow library for model construction/learning and had two orders of magnitude better throughput for inference queries.

6.2 Future Work

Modelling and Experiments If I had more time, I would have liked to test the accuracy and runtimes of SSPN models over other data sets. Furthermore, I would have analysed how the runtime performance can be used to improve the accuracy of the models by training SSPNs over larger amounts of relational data or by training several models within the given time budget, and then choose the one with highest accuracy. The scripting language has been designed specifically to allow for easy modelling and deployment of SSPN models. In our search for the SSPN used in the experiments, we tried with a range of simpler SSPNs that considered different similarity measures, that created new join variables by grouping on the various properties of users and movies. They all reported worse accuracy and were not reported in the report for lack of space.

Optimisation of Inference If a set of conditional probability queries share the same value for a variable from the evidence, further runtime optimisation can be achieved by constructing a pruned SSPN where the above variable is marginalised. Thus, because the set of queries are evaluated on a smaller SSPN, the total runtime improves. This is akin to view materialisation in databases: One precomputes a subquery that is common to many queries to speed up the evaluation of all these queries. Most probably, I will implement this feature during the summer.

Optimisation of Learning Another idea that I would have liked to investigate, and which will most probable do so in the future, is to analyse how various in-database optimisations can be applied to our SSPN system. For example, since we know exactly how the tables that define the model are joined together, further runtime optimisation may be achieved by using indexes and particular choices of physical join implementations such as sort-merge joins as opposed to hash joins.

Implementation of Updates Another feature that will most probable be added to the system in the near future is to implement the algorithms that maintain the SSPNs under input data updates (tuples insertions and deletions).

Further direction A more general direction for future work can also consist of investigating and applying the structure-aware learning and in-database representations approaches for other types of Machine Learning tasks.

Bibliography

[1] A. Atserias, M. Grohe, and D. Marx. Size bounds and query plans for relational joins. In *2008 49th Annual IEEE Symposium on Foundations of Computer Science*, pages 739–748. IEEE, 2008.

[2] N. Bakibayev, D. Olteanu, and J. Závodnỳ. Fdb: A query engine for factorised relational databases. *arXiv preprint arXiv:1203.2672*, 2012.

[3] N. Bakibayev, T. Kočiskỳ, D. Olteanu, and J. Závodnỳ. Aggregation and ordering in factorised databases. *arXiv preprint arXiv:1307.0441*, 2013.

[4] Y. Bengio et al. Learning deep architectures for ai. *Foundations and trends® in Machine Learning*, 2(1):1–127, 2009.

[5] W.-C. Cheng, S. Kok, H. V. Pham, H. L. Chieu, and K. M. A. Chai. Language modeling with sum-product networks. In *Fifteenth Annual Conference of the International Speech Communication Association*, 2014.

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

[7] A. P. Dempster, N. M. Laird, and D. B. Rubin. Maximum likelihood from incomplete data via the em algorithm. *Journal of the Royal Statistical Society: Series B (Methodological)*, 39(1):1–22, 1977.

[8] A. Dennis and D. Ventura. Learning the architecture of sum-product networks using clustering on variables. In *Advances in Neural Information Processing Systems*, pages 2033–2041, 2012.

[9] R. Gens and P. Domingos. Discriminative learning of sum-product networks. In *Advances in Neural Information Processing Systems*, pages 3239–3247, 2012.

[10] R. Gens and D. Pedro. Learning the structure of sum-product networks. In *International conference on machine learning*, pages 873–880, 2013.

[11] T. George and S. Merugu. A scalable collaborative filtering framework based on co-clustering. In *Fifth IEEE International Conference on Data Mining (ICDM'05)*, pages 4–pp. IEEE, 2005.

[12] F. M. Harper and J. A. Konstan. The movielens datasets: History and context. *Acm transactions on interactive intelligent systems (tiis)*, 5(4):1–19, 2015.

[13] J. Hellerstein, C. Ré, F. Schoppmann, D. Z. Wang, E. Fratkin, A. Gorajek, K. S. Ng, C. Welton, X. Feng, K. Li, et al. The madlib analytics library or mad skills, the sql. *arXiv preprint arXiv:1208.4165*, 2012.

[14] N. Hug. Surprise, a Python library for recommender systems. http://surpriselib.com, 2017.

[15] Kaggle. The State of Data Science and Machine Learning, 2017. URL https://www.kaggle.com/surveys/2017.

[16] D. Koller and N. Friedman. *Probabilistic graphical models: principles and techniques*. MIT press, 2009.

[17] Y. Koren. Factorization meets the neighborhood: a multifaceted collaborative filtering model. In *Proceedings of the 14th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 426–434, 2008.

[18] Y. Koren. Factor in the neighbors: Scalable and accurate collaborative filtering. *ACM Transactions on Knowledge Discovery from Data (TKDD)*, 4 (1):1–24, 2010.

[19] Y. Koren, R. Bell, and C. Volinsky. Matrix factorization techniques for recommender systems. *Computer*, 42(8):30–37, 2009.

[20] D. Lemire and A. Maclachlan. Slope one predictors for online rating-based collaborative filtering. In *Proceedings of the 2005 SIAM International Conference on Data Mining*, pages 471–475. SIAM, 2005.

[21] X. Luo, M. Zhou, Y. Xia, and Q. Zhu. An efficient non-negative matrix-factorization-based approach to collaborative filtering for recommender systems. *IEEE Transactions on Industrial Informatics*, 10(2):1273–1284, 2014.

[22] A. Molina, A. Vergari, N. Di Mauro, S. Natarajan, F. Esposito, and K. Kersting. Mixed sum-product networks: A deep architecture for hybrid domains. In *Thirty-second AAAI conference on artificial intelligence*, 2018.

[23] A. Molina, A. Vergari, K. Stelzner, R. Peharz, P. Subramani, N. D. Mauro, P. Poupart, and K. Kersting. Spflow: An easy and extensible library for deep probabilistic learning using sum-product networks, 2019.

[24] D. Olteanu and M. Schleich. Factorized databases. *ACM SIGMOD Record*, 45(2):5–16, 2016.

[25] D. Olteanu and J. Závodnỳ. Factorised representations of query results: size bounds and readability. In *Proceedings of the 15th International Conference on Database Theory*, pages 285–298, 2012.

[26] D. Olteanu and J. Závodnỳ. Size bounds for factorised representations of query results. *ACM Transactions on Database Systems (TODS)*, 40(1):1–44, 2015.

[27] R. Peharz, B. C. Geiger, and F. Pernkopf. Greedy part-wise learning of sum-product networks. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, pages 612–627. Springer, 2013.

[28] R. Peharz, G. Kapeller, P. Mowlaee, and F. Pernkopf. Modeling speech with sum-product networks: Application to bandwidth extension. In *2014 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 3699–3703. IEEE, 2014.

[29] R. Peharz, S. Tschiatschek, F. Pernkopf, and P. Domingos. On theoretical properties of sum-product networks. In *Artificial Intelligence and Statistics*, pages 744–752, 2015.

[30] H. Poon and P. Domingos. Sum-product networks: A new deep architecture. In *2011 IEEE International Conference on Computer Vision Workshops (ICCV Workshops)*, pages 689–690. IEEE, 2011.

[31] A. Rooshenas and D. Lowd. Learning sum-product networks with direct and indirect variable interactions. In *International Conference on Machine Learning*, pages 710–718, 2014.

[32] D. Roth. On the hardness of approximate reasoning. *Artificial Intelligence*, 82(1-2):273–302, 1996.

[33] M. Schleich, D. Olteanu, M. Abo-Khamis, H. Q. Ngo, and X. Nguyen. Learning models over relational data: A brief tutorial. In *International Conference on Scalable Uncertainty Management*, pages 423–432. Springer, 2019.

[34] A. Vergari. Awesome Sum-Product Networks, October 2019. URL https://github.com/arranger1044/awesome-spn.

[35] H. Zhao, M. Melibari, and P. Poupart. On the relationship between sum-product networks and bayesian networks. In *International Conference on Machine Learning*, pages 116–124, 2015.