Feature Attribution via SHAP on d-DNNFs and Sentential Decision Diagrams

Esther Ng Xin Yue
Nanyang Technological University
URECA Research Programme

December 2025

Abstract

Shapley Additive Explanations (SHAP) provide an axiomatically grounded framework for feature attribution, but exact SHAP computation is intractable in general due to its exponential dependence on the number of features. Recent work has shown that this intractability can be overcome when models admit tractable circuit representations. In this project, we study exact SHAP computation for Boolean classifiers represented as Sentential Decision Diagrams (SDDs), a structured subclass of deterministic and decomposable negation normal forms (d-DNNFs). Building on the dynamic programming algorithm of Arenas et al. (2023), we implement an exact SHAP computation procedure that exploits the determinism, decomposability, and structured decomposability of SDDs. Our results confirm that exact, axiomatically justified feature attribution is practically feasible for Boolean models that admit tractable circuit representations.

1 Introduction

Feature attribution is a core problem in Explainable AI (XAI), aiming to determine which features are responsible for a model's output. SHAP is particularly important due to its strong theoretical basis in cooperative game theory, model-agnostic applicability, and desirable axiomatic properties. However, we must note that the SHAP score is exponentially expensive to compute, and thus there is a need to search for classes of models where computing SHAP becomes tractable.

2 Background on Tractable Boolean Circuits

Tractable circuits are computational representations, either Boolean or Arithmetic representations, which support efficient reasoning tasks that are normally intractable due to being NP-hard or worse. They operate by traversing the circuit in a feed-forward fashion much like neural networks (Darwiche, 2021). Tractable circuits have structural properties (e.g decomposability, determinism, smoothness etc.) , which ensures that many inference queries can be evaluated in linear time in the circuit size (Darwiche, 2021).

2.1 Boolean Circuits

Boolean circuits operate over logical values (true or false) using three types of gates: and-gates, or-gates, and inverters ('not'). They are mostly used for symbolic reasoning, such as determining satisfiability, entailment or logical consistency (Darwiche, 2021).

2.2 Key Structural Properties Enabling Tractability

These properties ensure that tractable circuits can be evaluated in a single bottom-up traversal, similar to a neural network forward pass. Different combinations of the properties enable different classes of inference. Additionally, these properties also explain why SHAP scores become tractable on d-DNNFs, since SHAP reduces to a sequence of weighted model counting queries, which d-DNNFs support efficiently.

2.2.1 Decomposability

An AND-gate is decomposable if its inputs depend on disjoint sets of variables. This ensures independence between subcircuits and enables linear-time satisfiability checking and model counting. Negation Normal Form (NNF) circuits that are decomposable are known as DNNF circuits.

2.2.2 Determinism

An OR-gate is deterministic if at most one of its inputs evaluates to true under any assignment. Determinism prevents double-counting and is essential for tractable weighted model counting and MPE computation. NNF circuits that are both decomposable and deterministic are known as d-DNNF circuits, and are exponentially less succinct than DNNF circuits.

2.2.3 Smoothness

An OR-gate is smooth if all of its inputs mention the same set of variables. This property ensures that counts and probabilities are aggregated over comparable spaces.

2.2.4 Decision Property

An OR-gate satisfies the decision property if it branches on a single Boolean variable, analogous to an if–else construct. This property characterises Decision-DNNFs and is inherent to ordered binary decision diagrams (OBDDs).

2.2.5 Structured Decomposability and Partitioned Determinism

These are stronger versions of decomposability and determinism, defined relative to a vtree, a full binary tree whose leaves are in one-to-one correspondence with the circuit variables. Together they define Sentential Decision Diagrams (SDDs), which allows us to solve expressive reasoning tasks such as MajMajSAT in linear time (Darwiche, 2021).

2.3 From Boolean to Arithmetic Circuits

Arithmetic circuits can be viewed as Boolean circuits, where or-gates are replaced with sum nodes, and and-gates are replaced with product nodes. This turns Boolean model counting into weighted model counting (WMC), which directly corresponds to probabilistic inference tasks such as computing marginals and computing most probable explanations (MPE) (Darwiche, 2021).

Arithmetic operations are applicable to gradient-based optimisation commonly used in modern machine learning, such as backpropagation and stochastic gradient descent (SGD). However, Boolean circuits can only output true or false values, which cannot be differentiated.

2.4 Compilation and Learning of Tractable Circuits

Tractable circuits can be obtained in two main ways: either through knowledge compilation or learning. The choice between the two ways depends on whether we know the underlying model structure beforehand. Compilation is suitable when we know the explicit model structure, while learning is used when only data is available, and we must infer the model structure ourselves.

2.4.1 Knowledge Compilation

Logical or probabilistic models (e.g. CNF formulas, Bayesian networks) can be compiled into tractable circuit representations such as d-DNNFs or probabilistic Sentential Decision Diagrams (PSDDs). Once compiled, expensive reasoning tasks can be performed in linear time.

2.4.2 Learning

Other circuit classes, such as Sum-Product Networks (SPNs), are learned directly from data. These typically enforce decomposability and smoothness, enabling tractable marginal inference. However, without determinism, MPE computation is generally intractable unless additional constraints are learned.

3 SHAP Computation on Tractable Boolean Circuits

SHapley Additive exPlanations (SHAP) is one of the most widely used feature attribution methods, grounded in cooperative game theory and defined as the average marginal contribution of a feature across all possible coalitions of features. However, computing the exact SHAP values of features can be computationally hard, making it hard to use (Arenas et al., 2023). However, SHAP values become tractable when the underlying model is represented using tractable Boolean circuits, particularly d-DNNFs, which possess deterministic and decomposable properties.

3.1 Computation Hardness of SHAP

Shapley values formalise the contribution of a feature i in a feature set X by averaging its marginal contributions over all possible coalitions of features. For a fixed instance x and corresponding

feature set X, the SHAP value of feature i is

$$\phi_i = \sum_{S \subseteq X \setminus \{i\}} \sigma(S)\, \Delta_i(S), \tag{1}$$

where S ranges over all subsets of X not containing i, $\sigma(S)$ is the Shapley kernel weight, and $\Delta_i(S)$ is the marginal contribution of feature i to coalition S.

The Shapley kernel assigns a weight to each coalition size:

$$\sigma(S) = \frac{|S|!\,\big(|X| - |S| - 1\big)!}{|X|!}, \tag{2}$$

and the marginal contribution of feature i to coalition S is given by

$$\Delta_i(S) = v(S \cup \{i\}) - v(S), \tag{3}$$

where $v(\cdot)$ denotes the value function associated with the model and the underlying distribution. We can already identify two sources of combinatorial blowup.

1. Exponential number of coalitions: For each feature i, the sum in the definition of ϕ_i ranges over all subsets $S \subseteq X \setminus \{i\}$. If $|X| = n$, there are 2^{n-1} such subsets. A naive computation of ϕ_i therefore requires exponentially many value-function evaluations $v(S)$ and $v(S \cup \{i\})$.

2. Hardness of the value function: In the expectation-based SHAP setting considered in recent work, the value function is defined as

$$v(S) = \mathbb{E}\big[\, f(X) \,\big|\, X_S = x_S \big], \tag{4}$$

where f is the classifier, X is a random vector distributed according to some background distribution P, and x_S denotes the restriction of the instance x to the coordinates in S. Intuitively, $v(S)$ measures the expected model output when the features in S are fixed to their observed values, and the remaining features are integrated out. For general Boolean classifiers, this corresponds to a #P-hard weighted model counting problem (Arenas et al., 2023).

3. Main hardness result: Arenas et al. (2023) formalise this intuition and show that computing SHAP values is #P-hard for Boolean classifiers with expectation-based value functions, even under fairly simple distributions (e.g., the uniform distribution). In particular:

- computing a single value $v(S)$ is already #P-hard in general, as it encodes a weighted model counting problem;
- computing all marginal contributions $\Delta_i(S)$ requires evaluating $v(S)$ and $v(S \cup \{i\})$ for exponentially many coalitions S; and

- aggregating these contributions into ϕ_i therefore inherits both the exponential blowup in the number of coalitions and the intrinsic #P-hardness of each value-function evaluation.

These results imply that, without additional structural assumptions on the model or the underlying representation, exact SHAP computation is intractable in the worst case, which motivates the search for tractable subclasses such as circuits in d-DNNF or SDD form.

3.2 Expectation-Based SHAP and Value Functions on Circuits

In the expectation-based formulation of SHAP, the contribution of a feature subset is defined through a value function $v(S)$, which assigns a score to every coalition $S \subseteq X$ of features.

For a Boolean classifier $f : \{0,1\}^n \to \{0,1\}$ and an instance $x^\star$, the value function is defined as

$$v(S) = \Pr(f(X) = f(x^\star) \mid X_S = x^\star_S)\,, \tag{5}$$

where X is a random input drawn from the background distribution, and $X_S = x^\star_S$ denotes conditioning on the features in S being fixed to their observed values. Intuitively, $v(S)$ measures how well fixing only the features in S preserves the original model prediction. A value close to 1 indicates that the prediction is largely determined by S, whereas a value close to 0 indicates that the omitted features significantly influence the outcome.

This expectation-based formulation is particularly suitable for models represented as Boolean circuits, such as d-DNNFs and SDDs. These representations support tractable probabilistic reasoning and enable exact computation of conditional probabilities of the form (5).

Connection to Weighted Model Counting

Arenas et al. show that the conditional probability in (5) can be expressed in terms of weighted model counting (WMC). Specifically,

$$v(S) = \frac{\mathrm{WMC}\left(f(X) = f(x^\star),\ X_S = x^\star_S\right)}{\mathrm{WMC}\left(X_S = x^\star_S\right)}, \tag{6}$$

where $\mathrm{WMC}(\cdot)$ denotes the weighted sum of satisfying assignments under the background distribution. The numerator computes the total probability mass of all assignments consistent with the evidence $X_S = x^\star_S$ that also preserve the prediction $f(x) = f(x^\star)$, while the denominator normalises by the probability mass of the evidence itself.

The reduction in (6) is made possible by the structural properties of tractable circuits:

- *Determinism* ensures that the satisfying assignments of disjoint branches do not overlap, allowing probabilities to be added without double-counting.

- *Decomposability* guarantees that child subcircuits operate on disjoint variable sets, enabling independent aggregation of probability mass.

- *Smoothness* ensures that all subcircuits refer to the same set of variables, which is necessary for correct marginalisation.

These properties make weighted model counting linear-time in the size of a d-DNNF or SDD. Consequently, computing the value function $v(S)$ becomes a single upward pass over the circuit, in contrast with the #P-hard complexity of model counting on unrestricted Boolean formulas. This connection enables tractable SHAP computation on structured circuit representations.

3.3 Algorithm Adopted

The SHAP computation procedure for deterministic and decomposable circuits follows the algorithmic framework of Arenas et al. (2023), which exploits the structural properties of d-DNNFs to compute SHAP scores in linear time to the size of the circuit. The circuit representation decomposes the computation into local operations that can be aggregated efficiently. The algorithm proceeds in three main phases:

1. Circuit Normalisation and Variable Extraction

Given an input deterministic and decomposable Boolean circuit C, the algorithm first transforms C into an equivalent *smooth* circuit D in which every disjunction- and conjunction-gate has fan-in equal to 2. Smoothness ensures that all children of a gate reference the same variable set, which is crucial for the correct arithmetic of the γ and δ values.

For every gate g in D, the algorithm computes the set $\mathrm{var}(g)$ of variables that appear in the subcircuit rooted at g.

2. Bottom-Up Computation of γ_g^ℓ and δ_g^ℓ

For each gate g and each index $\ell \in \{0, \ldots, |\mathrm{var}(g) \setminus \{x\}|\}$, the algorithm computes two arrays:

$$\gamma_g^\ell, \qquad \delta_g^\ell,$$

which encode, respectively, the weighted model counts associated with g under completions of size ℓ, and the corresponding expected-value terms derived from the value function.

These are computed by bottom-up induction on the circuit structure:

Constant gates: If g is a constant gate with label $a \in \{0, 1\}$,

$$\gamma_g^0 \leftarrow a, \qquad \delta_g^0 \leftarrow a.$$

Variable gate for the target feature x: If g is a variable gate with $\mathrm{var}(g) = \{x\}$,

$$\gamma_g^0 \leftarrow 1, \qquad \delta_g^0 \leftarrow 0.$$

Variable gate for a non-target variable $y \neq x$:

$$\gamma_g^0 \leftarrow p(y), \qquad \delta_g^0 \leftarrow e(y),$$

where $p(y)$ is the probability of y under the background distribution and $e(y)$ is the corresponding expected-value term.

Negation gates: If g is a $\neg$-gate with input g', then for all ℓ,

$$\gamma_g^\ell \leftarrow (|\mathrm{var}(g) \setminus \{x\}| - \ell) - \gamma_{g'}^\ell, \qquad \delta_g^\ell \leftarrow (|\mathrm{var}(g) \setminus \{x\}| - \ell) - \delta_{g'}^\ell.$$

OR gates: If g is a $\vee$-gate with children g_1, g_2, then for all ℓ,

$$\gamma_g^\ell \leftarrow \gamma_{g_1}^\ell + \gamma_{g_2}^\ell, \qquad \delta_g^\ell \leftarrow \delta_{g_1}^\ell + \delta_{g_2}^\ell.$$

Determinism guarantees that the model counts from both children can be added without double-counting.

AND gates: If g is an $\wedge$-gate with children g_1, g_2, decomposability ensures that their variable sets are disjoint. Thus,

$$\gamma_g^\ell \leftarrow \sum_{\ell_1+\ell_2=\ell} \gamma_{g_1}^{\ell_1} \cdot \gamma_{g_2}^{\ell_2}, \qquad \delta_g^\ell \leftarrow \sum_{\ell_1+\ell_2=\ell} \delta_{g_1}^{\ell_1} \cdot \delta_{g_2}^{\ell_2}.$$

This convolution-style recurrence arises directly from the independence induced by decomposability.

3. SHAP Aggregation

After computing γ_g^ℓ and δ_g^ℓ for all gates, the SHAP value for feature x is computed by examining only the output gate g_{out}.

The Shapley value is given by

$$\mathrm{SHAP}(C, e, x) = \sum_{k=0}^{|X|-1} \frac{k!\,(|X| - k - 1)!}{|X|!} \left[(e(x) - p(x)) \left(\gamma_{g_{\mathrm{out}}}^k - \delta_{g_{\mathrm{out}}}^k \right) \right].$$

This expression is the Shapley kernel expansion over coalition sizes k, but the algorithm never enumerates coalitions. Each γ^ℓ and δ^ℓ encodes the contributions of all coalitions of size ℓ, extracted from the circuit structure.

Linear-Time Computation: Each gate of the circuit is processed exactly once in the bottom-up computation. All updates involve constant-time operations or bounded-size convolutions proportional to the number of variables in the gate, which is at most the circuit size. As a result, the

algorithm runs in time

$$O(|D|),$$

where $|D|$ is the number of gates in the smooth d-DNNF.

The combination of decomposability, determinism, and smoothness ensures that these computations correctly aggregate all SHAP contributions without explicitly iterating over subsets of variables. This is the key insight that makes exact SHAP computation tractable on d-DNNFs and SDDs.

3.4 Complexity

Arenas et al. (2023) established both positive and negative results regarding the computational complexity of SHAP when models are represented as Boolean circuits. On the positive side, they show that when the classifier is compiled into a deterministic and decomposable circuit, such as a d-DNNF, and the input distribution is fully factorised, exact SHAP values can be computed in polynomial time with respect to the size of the circuit.

However, in the absence of determinism or decomposability, SHAP computation becomes #P-hard. Furthermore, for general models that are not represented in tractable circuit form, approximating SHAP values is also computationally hard. As such, tractable circuits such as d-DNNFs and SDDs are essentially the only representations for feasible computation of SHAP scores.

3.5 Implications for SDD SHAP Computation

Since SDDs are a structured subclass of d-DNNFs, satisfying determinism, decomposability, and structured decomposability with respect to a vtree (Darwiche, 2011), all tractability results for SHAP on d-DNNFs transfer directly to SDDs. As such, this justifies the use of SDDs as the circuit representation for the SHAP implementation.

4 Computation of SHAP Scores on SDDs (SDDs as Subsets of d-DNNFs)

By representing Boolean circuits as SDDs, folowing Algorithm 2 of Arenas et al. (2023) and adapting it to the SDD structure, we are able to compute SHAP scores on SDDs.

4.1 Problem Formulation

We consider Boolean classifiers represented as tractable circuits and adopt the expectation-based SHAP formulation of Arenas et al. (2023). Let

$$C : \{0,1\}^n \to \{0,1\}$$

be a Boolean classifier represented as an SDD, let

$$X = \{x_1, \ldots, x_n\}$$

denote the set of input features, and let

$$e = (e_1, \ldots, e_n) \in \{0, 1\}^n$$

be the entity (instance) whose prediction we wish to explain. We assume a fully factorised background distribution

$$P(X) = \prod_{i=1}^{n} P(x_i),$$

with marginal probabilities

$$p_i = P(x_i = 1).$$

The SHAP value of feature x_i is defined as the Shapley value of player i under a value function based on conditional expectations. Formally,

$$\mathrm{SHAP}_i(C, e, P) = \sum_{S \subseteq X \setminus \{x_i\}} w(|S|, n) \Big(\mathbb{E}\big[C \mid X_S = e_S,\, x_i = e_i\big] - \mathbb{E}\big[C \mid X_S = e_S\big] \Big), \tag{7}$$

where X_S denotes the set of variables in coalition S, e_S is the restriction of the entity e to S, and the Shapley weight for coalitions of size k is

$$w(k, n) = \frac{k!\,(n - k - 1)!}{n!}. \tag{8}$$

The quantity

$$\mathbb{E}\big[C \mid X_S = e_S,\, x_i = e_i\big]$$

is the expected output of the classifier when the features in S and x_i are fixed to their entity values, while

$$\mathbb{E}\big[C \mid X_S = e_S\big]$$

is the corresponding expectation when only S is fixed and x_i remains random. Their difference therefore measures the marginal contribution of x_i to coalition S, and the Shapley kernel aggregates these marginal effects across all possible coalitions in a fair and axiomatically justified manner.

4.2 Algorithm Overview

For each target variable x_i, the SHAP formula requires conditional expectations of the form

$$\mathbb{E}\big[C \mid X_S = e_S,\, x_i = 1\big] \quad \text{and} \quad \mathbb{E}\big[C \mid X_S = e_S,\, x_i = 0\big],$$

for many coalitions $S \subseteq X \setminus \{x_i\}$.

4.2.1 Auxiliary arrays per gate

Fix a target variable x_i. For each SDD gate g and each coalition size ℓ, we maintain two auxiliary quantities

$$\gamma_g[\ell], \qquad \delta_g[\ell].$$

These summarise all conditional expectations involving coalitions of size ℓ, without explicitly enumerating such coalitions.

- $\gamma_g[\ell]$ is the expected value of gate g when x_i is forced to 1, exactly ℓ variables in $\text{var}(g) \setminus \{x_i\}$ are fixed to their entity values, and the remaining variables follow the background product distribution.
- $\delta_g[\ell]$ is defined exactly the same way, but with $x_i = 0$.

The difference $\gamma_g[\ell] - \delta_g[\ell]$ encodes the local marginal effect of setting x_i to 1 instead of 0, under coalitions of size ℓ at gate g. These two arrays allow SHAP to be computed by aggregating over coalition sizes $\ell = 0, 1, \ldots, n-1$, rather than over all 2^{n-1} subsets.

4.2.2 Base cases (terminal SDD nodes)

Before handling internal SDD nodes, we initialise γ and δ at terminals. These formulas follow directly from the semantics of conditional expectation under product distributions.

1. Constant terminals $\top$ and $\bot$: Constant gates do not depend on any variable, so only $\ell = 0$ is relevant and we set

$$\gamma_g[0] = \delta_g[0] = \begin{cases} 1, & \text{if } g = \top, \\ 0, & \text{if } g = \bot. \end{cases} \tag{9}$$

2. Terminal for the target variable x_i: If g is a literal for the target variable, i.e. $g = x_i$ or $g = \neg x_i$, then $\text{var}(g) = \{x_i\}$, so only $\ell = 0$ applies. We set

$$\gamma_{x_i}[0] = 1, \qquad \delta_{x_i}[0] = 0, \tag{10}$$

$$\gamma_{\neg x_i}[0] = 0, \qquad \delta_{\neg x_i}[0] = 1. \tag{11}$$

These assignments follow directly from Boolean semantics: forcing $x_i = 1$ makes x_i true and $\neg x_i$ false, and vice versa for $x_i = 0$.

3. Terminals for non-target variables $y \neq x_i$: Let y be a non-target variable with marginal probability $p_y = P(y = 1)$ and entity value $e_y \in \{0, 1\}$. When y is not in the coalition (i.e. $\ell = 0$ for this gate), it either remains random (in γ) or is fixed to the entity (in δ). Thus we set

$$\gamma_y[0] = p_y, \qquad \gamma_{\neg y}[0] = 1 - p_y, \tag{12}$$

$$\delta_y[0] = e_y, \qquad \delta_{\neg y}[0] = 1 - e_y. \tag{13}$$

4.2.3 AND-aggregation in SDD decision elements

An SDD decision node consists of elements (p_i, s_i), each representing the conjunction $p_i \wedge s_i$. Because SDDs satisfy decomposability, their variable sets are disjoint:

$$\mathrm{var}(p_i) \cap \mathrm{var}(s_i) = \emptyset. \tag{14}$$

This is the same decomposability property used in the d-DNNF algorithm. It implies that expectations factor multiplicatively. For each element (p_i, s_i) we define an intermediate gate $h_i = p_i \wedge s_i$ and compute, for every coalition size ℓ,

$$\gamma_{h_i}[\ell] = \sum_{\ell_p + \ell_s = \ell} \gamma_{p_i}[\ell_p]\, \gamma_{s_i}[\ell_s], \tag{15}$$

$$\delta_{h_i}[\ell] = \sum_{\ell_p + \ell_s = \ell} \delta_{p_i}[\ell_p]\, \delta_{s_i}[\ell_s]. \tag{16}$$

This convolution rule is identical to the AND-aggregation used for d-DNNFs in Arenas et al. (2023). Decomposability guarantees that any coalition of ℓ fixed variables splits uniquely between the prime and sub parts.

4.2.4 OR-aggregation in SDD decision nodes (virtual smoothing)

An SDD decision node g represents a disjunction of its elements:

$$g = \bigvee_{i=1}^{m} (p_i \wedge s_i), \tag{17}$$

and determinism ensures that different elements are mutually exclusive:

$$(p_i \wedge s_i) \wedge (p_j \wedge s_j) = \bot \qquad \text{for } i \neq j. \tag{18}$$

Thus, expectations across elements sum, but each child $(p_i \wedge s_i)$ may mention only a subset of the variables present in the parent gate. To combine the arrays correctly, we use the *virtual smoothing* technique of Arenas et al. (2023).

Let

$$V_g = \mathrm{var}(g) \setminus \{x_i\}$$

be the coalition variables available to the parent gate,

$$V_c^{(i)} = \mathrm{var}(p_i \wedge s_i)$$

the variables present in child i, and

$$V_{\text{miss}}^{(i)} = V_g \setminus V_c^{(i)}$$

the variables that appear in the parent but not in that child. To compute $\gamma_g[\ell]$, we consider all ways in which a coalition of size ℓ can distribute its fixed variables between the child and the missing variables. The OR-aggregation becomes

$$\gamma_g[\ell] = \sum_{i=1}^{m} \sum_{\ell_c=0}^{\min(\ell,|V_c^{(i)}|)} \binom{|V_{\text{miss}}^{(i)}|}{\ell - \ell_c} \gamma_{h_i}[\ell_c], \tag{19}$$

where $h_i = p_i \wedge s_i$ is the element gate. The binomial coefficient counts how many ways $\ell - \ell_c$ fixed variables may lie among the missing variables. The same reasoning applies to $\delta_g[\ell]$:

$$\delta_g[\ell] = \sum_{i=1}^{m} \sum_{\ell_c=0}^{\min(\ell,|V_c^{(i)}|)} \binom{|V_{\text{miss}}^{(i)}|}{\ell - \ell_c} \delta_{h_i}[\ell_c]. \tag{20}$$

This is exactly the virtual smoothing procedure used for OR-gates in d-DNNFs, specialised to SDD decision nodes.

4.2.5 Summary of algorithm and complexity

The components above form a complete dynamic program for SHAP on SDDs. The arrays $\gamma_g[\ell]$ and $\delta_g[\ell]$ act as compressed summaries of exponentially many conditional expectations. Terminal formulas follow directly from expectation semantics under the product distribution, AND-aggregation is enabled by decomposability, and OR-aggregation is enabled by determinism plus smoothing over missing variables.

Let $|C|$ denote the number of gates in the SDD and d the maximum fan-in of any gate. The cost of computing SHAP for a single variable x_i is $O(|C| \cdot d^2)$, since we perform one bottom-up pass over the circuit and, at each decision node, combining child arrays across coalition sizes induces a quadratic dependence on the gate degree. Computing SHAP scores for all n variables thus runs in time

$$O(n \cdot |C| \cdot d^2),$$

matching the complexity bounds reported for d-DNNFs in Arenas et al. (2023), up to structural constants specific to SDDs.

4.3 SHAP Score Aggregation at the Root

Once the arrays $\gamma_g[\ell]$ and $\delta_g[\ell]$ have been computed for all gates, the SHAP score of a feature is obtained by aggregating the information stored at the output gate of the circuit. Following the dynamic-programming formulation of Arenas et al. (2023), the SHAP value of feature x_i can be expressed purely in terms of the arrays $\gamma_{g_{\text{out}}}[k]$ and $\delta_{g_{\text{out}}}[k]$ associated with the output gate g_{out},

aggregated over coalition sizes.

For a given coalition size k, the quantity $\gamma_{g_{\text{out}}}[k]$ denotes the expected value of the circuit output when the target feature x_i is fixed to 1, exactly k other features are fixed to their entity values, and the remaining features are sampled from the product distribution. The corresponding value $\delta_{g_{\text{out}}}[k]$ is the analogous expectation when x_i is fixed to 0. Their difference,

$$\gamma_{g_{\text{out}}}[k] - \delta_{g_{\text{out}}}[k],$$

therefore captures the marginal impact of feature x_i on the expected output under coalitions of size k.

By grouping the Shapley sum over all coalitions $S \subseteq X \setminus \{x_i\}$ according to their size $k = |S|$, and using the standard Shapley weights

$$w(k, n) = \frac{k!\,(n-k-1)!}{n!}, \tag{21}$$

we obtain the compact aggregation formula

$$\text{SHAP}_i = \sum_{k=0}^{n-1} w(k, n)\,(e_i - p_i)\,\big(\gamma_{g_{\text{out}}}[k] - \delta_{g_{\text{out}}}[k]\big), \tag{22}$$

where e_i is the entity's value for feature i and $p_i = P(x_i = 1)$ is its marginal probability under the background distribution. The factor $(e_i - p_i)$ quantifies how different the entity's feature value is relative to the prior, while the bracketed term encodes the feature's marginal effect on the expected output aggregated over all coalition sizes. Equation (22) is a direct algebraic consequence of the original SHAP-score definition combined with the circuit-based dynamic program of Arenas et al. (2023), and can be evaluated in time polynomial in the size of the underlying SDD.

4.4 Implementation Architecture

The implementation follows a layered architecture that cleanly separates circuit construction, input handling, SHAP computation, and visualisation. This design decouples the mathematical core of the algorithm from auxiliary concerns such as file parsing and debugging support, enabling incremental development and facilitating future extensions.

4.4.1 System Overview

At a high level, the workflow proceeds as a pipeline. A Boolean formula is first compiled into an SDD, marginal probabilities and entity values are then validated and loaded, the SHAP algorithm is executed on the compiled circuit, and the resulting feature attributions are returned together with diagnostic quantities for verification. The project structure reflects this separation of concerns:

```
shap-sdd-project/
+-- src/
```

```
|   +-- main.py
|   +-- shap/compute_shap.py
|   +-- sdd/sdd_utils.py
|   +-- sdd/sdd_visualizer.py
|   +-- utils/helpers.py
+-- tests/
+-- output/
```

This modular layout allows individual components to be tested and reasoned about independently.

4.4.2 Core SHAP Computation Module

The core SHAP computation is implemented in `src/shap/compute_shap.py`, which realises Algorithm 2 of Arenas et al. (2023) for SDDs. Given a compiled SDD, a product distribution over variables, and an entity e, the module computes exact SHAP scores via a dynamic program over the circuit.

The computation begins by deriving a topological ordering of the SDD nodes to ensure that all child nodes are processed before their parents. The algorithm then iterates over target variables x_i, computing the γ and δ arrays bottom-up for every gate, including the two-phase aggregation required at SDD decision nodes (AND aggregation via decomposability, followed by OR aggregation with virtual smoothing).

Once the arrays at the output gate g_{out} are available, the SHAP score of feature x_i is obtained by Shapley-weighted aggregation:

$$\phi_i = \sum_{k=0}^{n-1} \frac{k!\,(n-k-1)!}{n!}\,(e_i - p_i)\,\big(\gamma_{g_{\text{out}}}[k] - \delta_{g_{\text{out}}}[k]\big). \tag{23}$$

A wrapper function exposes this functionality through a user-facing interface. It handles input validation, mapping between external variable names and internal identifiers, and explicit handling of edge cases, ensuring that the core algorithm remains independent of input formats and error-handling logic.

The overall time complexity of computing SHAP scores for all n variables is

$$O(n \cdot |C| \cdot d^2),$$

where $|C|$ is the number of gates in the SDD and d is the maximum fan-in of any decision node.

4.4.3 SDD Construction Module

The SDD construction module, implemented in `src/sdd/sdd_utils.py`, is responsible for compiling Boolean formulas in CNF into SDD representations using the `PySDD` library. This compilation

step guarantees the structural properties (determinism, decomposability, and structured decomposability) that underpin tractable SHAP computation.

A right-linear vtree is used throughout the implementation. A vtree is said to be right-linear if each left-child is a leaf (Darwiche, 2011). This choice yields deterministic and reproducible circuit structures.

4.4.4 Command-Line Interface: main.py

An optional command-line interface provides end-to-end execution of the SHAP pipeline. Through this interface, users specify a CNF file defining the Boolean classifier, a JSON file containing marginal feature probabilities, and a JSON file describing the entity of interest. The system then compiles the formula into an SDD, executes the SHAP computation, and outputs both the feature attributions and auxiliary diagnostic information. This interface supports reproducible experiments and facilitates integration with automated benchmarking and testing workflows, while remaining decoupled from the core algorithmic implementation.

4.4.5 Visualisation Support

To support debugging and structural inspection, the implementation includes a visualisation module that exports SDDs in Graphviz DOT format. These visualisations allow inspection of circuit structure, verification of determinism and decomposability, and qualitative analysis of how SHAP contributions propagate through the circuit. The visualisation functionality is integrated into the testing pipeline, enabling diagrams to be generated automatically for selected test cases without manual intervention.

4.4.6 Testing Infrastructure

The system is supported by a comprehensive testing infrastructure comprising 22 unit and integration tests. Tests are organised by module and cover SHAP computation correctness, input validation, SDD construction, and visualisation output. Key theoretical properties of SHAP (such as efficiency and symmetry) are explicitly validated using small Boolean formulas with analytically predictable behaviour. All tests are executed automatically using `pytest`, ensuring that changes to individual components do not introduce regressions elsewhere in the system.

4.4.7 Good Software Practices

The modular design provides several practical advantages. From a maintainability perspective, each module has a clearly defined responsibility, enabling isolated debugging and targeted modifications. From a testability standpoint, components can be evaluated independently and mock inputs can be substituted during integration testing. The architecture also supports extensibility: new vtree strategies, alternative SHAP aggregation schemes, or additional validation rules can be introduced

without modifying the core dynamic-programming algorithm. Finally, reproducibility is ensured through deterministic vtree construction, structured logging, and version-controlled test data.

4.4.8 Performance Benchmarking

Performance benchmarking was conducted to evaluate the practical cost of computing exact SHAP scores on SDDs. All experiments were run on a standard development environment (Python 3.12 on an Apple M1 system). To assess scalability across increasing structural complexity, three representative Boolean formulas were used:

- **Small:** $(A \vee B)$,
- **Medium:** $(A \vee B) \wedge (C \vee D)$,
- **Large:** a constraint-based formula with a 3-SAT–like structure.

Each configuration was evaluated over 20 independent runs to account for variability in compilation and execution time.

Across all benchmarks, SDD compilation was the dominant cost, accounting for between approximately 78% and 93% of total runtime. For the smallest symmetric formula (2 variables, SDD size = 5), compilation consumed about 93.4% of execution time, while SHAP computation contributed only 1.6%. As circuit size increased, the relative cost of SHAP computation grew, but compilation remained the primary bottleneck. For the medium (4 variables, SDD size = 12) and larger constraint-based formulas (6 variables, SDD size = 28), SHAP computation accounted for roughly 18–19% of total runtime, with compilation still contributing close to 79%.

In absolute terms, SHAP computation was consistently fast, with average runtimes of 0.13 ms, 2.15 ms, and 3.32 ms for the small, medium, and large benchmarks, respectively. These runtimes scale smoothly with both the number of variables and the circuit size, and are consistent with the theoretical time complexity

$$O(n \cdot |C| \cdot d^2),$$

where n is the number of variables, $|C|$ the number of SDD nodes, and d the maximum fan-in of any decision node. In contrast, compilation time exhibited higher variance across runs, reflecting the sensitivity of SDD construction to formula structure and vtree choice.

Overall, these results confirm that once a tractable SDD representation is available, the additional computational cost of exact SHAP computation is modest. The primary performance bottleneck lies in compiling the Boolean formula into an SDD, rather than in the SHAP algorithm itself. This empirical finding supports the theoretical motivation of the approach: SHAP scores can be computed efficiently in practice provided the underlying model admits a tractable circuit representation.

5 Conclusion

In this report, we studied the problem of exact SHAP computation for Boolean classifiers through the lens of tractable circuit representations. While SHAP values are known to be computationally intractable in general, recent theoretical results show that SHAP values are tractable when models are represented as deterministic and decomposable circuits. As such, we focused on Sentential Decision Diagrams (SDDs), a structured subclass of d-DNNFs that offers strong tractability and practical support through existing compilation tools.

We formalised the expectation-based SHAP framework for Boolean circuits and explained how SHAP computation reduces to a sequence of weighted model counting queries. Leveraging the structural properties of SDDs (determinism, decomposability, and structured decomposability) we adapted the dynamic programming algorithm of Arenas et al. (2023) to SDDs. Central to this approach is the use of the auxiliary γ and δ arrays, which compactly encode exponentially many conditional expectations and enable SHAP values to be computed without enumerating feature coalitions.

Beyond the theoretical formulation, we presented a complete implementation architecture that decouples circuit construction, SHAP computation, validation, and visualisation. It computes exact SHAP scores for Boolean classifiers compiled into SDDs and provides diagnostic checks to verify key Shapley properties such as efficiency. Empirical benchmarking demonstrated that, once an SDD representation is available, the additional cost of exact SHAP computation is modest and scales smoothly with circuit size, in line with the theoretical complexity bounds. In practice, the dominant computational bottleneck lies in SDD compilation rather than in the SHAP algorithm itself.

6 Acknowledgements

I would like to express my sincere gratitude to Dr. Timothy Van Bremen and Dr. Huang Xuanxiang for their guidance, insightful discussions, and constructive feedback throughout the course of this project. Their expertise and mentorship were invaluable in shaping both the theoretical direction and the practical implementation of this work.

This research was conducted as part of the Undergraduate Research Experience on Campus (URECA) programme at Nanyang Technological University, whose support and structure made this project possible.

References

[1] Arenas, M., Barceló, P., Bertossi, L., & Monet, M. (2023). On the complexity of SHAP-score-based explanations: Tractability via knowledge compilation and non-approximability results. Journal of Machine Learning Research, 24(63), 1-58.

[2] Darwiche, A. (2021). Tractable Boolean and arithmetic circuits. In Neuro-Symbolic Artificial Intelligence: The State of the Art (pp. 146-172). IOS Press.

[3] Darwiche, A. (2011, July). SDD: A new canonical representation of propositional knowledge bases. In IJCAI Proceedings-International Joint Conference on Artificial Intelligence (Vol. 22, No. 1, p. 819-826).

Software and Tools

References

[1] Van den Broeck, G., et al. (2015). *PySDD: A Python library for Sentential Decision Diagrams.* Software repository. `https://github.com/ML-KULeuven/PySDD`