Scalable Computation of Shapley Additive Explanations

Louis Le Page [1], Christina Dionysio [2], and Matthias Boehm [2]

Abstract: The growing field of explainable AI (XAI) develops methods that help better understand ML model predictions. While SHapley Additive exPlanations (SHAP) is a widely-used, model-agnostic method for explaining predictions, its use comes with a substantial computational burden, particularly for complex models and large datasets with many features. The key—and so far unaddressed—challenge lies in efficiently scaling these computations without compromising accuracy. In this paper, we present a scalable, model-agnostic SHAP sampling framework on top of Apache SystemDS. We leverage Antithetic Permutation Sampling for its efficiency and optimization potential, and we devise a carefully vectorized and parallelized implementation for local and distributed operations. Compared with the state-of-the-art Python SHAP package, our solutions yield similar accuracy but achieve substantial speedups of up to 14× for multi-threaded singlenode operations as well as up to 35× for distributed Spark operations (on a small 8 node cluster).

Keywords: Explainable AI, SHAP Values, Interpretability, Parallelization, Vectorization, Antithetic Permutation Sampling

1 Introduction

Machine learning (ML) revolutionizes numerous aspects of everyday life, and drives advancements in diverse fields such as healthcare, finance, transportation, and personalized recommendations. ML models now underpin critical systems that diagnose diseases, detect fraud, navigate vehicles, and curate content on social media [OGS20; Ra22; Ru19]. This transformation has been fueled by the ability of ML models to achieve high prediction quality, often surpassing human capabilities in specific tasks. However, the widespread adoption of ML requires dealing with the "black-box" nature of complex, high-capacity models. While these models make accurate predictions, they often provide little to no insight into how they arrive at their decisions. This lack of transparency raises concerns about trust, accountability, and fairness, especially for high-risk applications [Es21; Re20].

Explainable AI: In recent years, explainability and interpretability in AI [DK17] gained importance. One of the most promising approaches is SHAP (SHapley Additive exPlanations) [LL17]. SHAP is based on the Shapley values [Sh53], from cooperative game theory, to attribute model outputs to input features in a fair and consistent manner. By providing a clear and theoretically grounded method for feature attribution, SHAP values enable a deeper understanding of model behavior, both for individual predictions (local interpretability) and across the entire dataset (global interpretability) [Mo23].

[1] TU Berlin, l.lepage@campus.tu-berlin.de, https://orcid.org/0009-0006-9175-3551

[2] TU Berlin & BIFOLD, dionysio@tu-berlin.de, https://orcid.org/0009-0002-1295-0460; matthias.boehm@tu-berlin.de, https://orcid.org/0000-0003-1344-3663

Scaling Challenges: SHAP relies on generating synthetic instances, each representing a distinct coalition (i.e., combination) of features, for which the model must be evaluated. The number of these coalitions scales exponentially with $O(2^k)$ in the number of features k [DP94]. Accordingly, obtaining exact Shapley values is infeasible. In practice, this issue is mitigated by estimating the values on a sample of the space of possible coalitions [ŠK10]. Various methods exist for accelerating this computation, mostly focusing on the strategic sampling of feature coalitions with high impact [CGT09; Lo19; Mi22]. Some approaches are model-specific and leverage the knowledge of the model's internals to improve efficiency [Lu20]. However, broadly-applicable, model-agnostic methods remain computationally intensive and struggle to handle many features [Mo23].

Contributions: In this paper, we devise a scalable implementation of Antithetic Permutation Sampling—a well-known model-agnostic SHAP value computation approach—on top of Apache SystemDS [Bo20]. In detail, we make the following technical contributions:

- **Background of SHAP Computation:** Section 2 points out the practical challenges of SHAP value computation and introduces three levels on which these challenges can be overcome: general method choice, algorithm design, and parallelization.
- **Optimization of Antithetic Permutation Sampling:** We then analyze the optimization potential of Antithetic Permutation Sampling, and propose optimizations on all three levels in Sections 3.1-3.3. Our implementation on top of SystemDS [Bo20] generates hybrid runtime plans of local, in-memory and distributed Spark operations.
- **Experiments:** Finally, in Section 4, we report on extensive experiments, studying the efficiency and scalability of our sampling framework compared with the state-of-the-art SHAP library on a variety of models and datasets. Our optimizations substantially improve end-to-end runtime while not sacrificing accuracy.

2 Background of SHAP Value Computation

There are two general issues in SHAP value computation: the exponential complexity of the Shapley Value, and feature removal when calling a model. From an algorithmic perspective, straightforward methods such as estimation [Ca17; CGT09] and marginalization [ŠK10] can yield reasonable estimates for Shapley values. However, as a basis for devising an effective and scalable computation strategy, we first summarize the practical challenges involved and levels at which these challenges can be addressed.

Challenges: Model-agnostic methods differ substantially regarding the complexity of individual steps in the SIPA framework [Sc20]: sampling, intervention, prediction, and aggregation. Advanced techniques like stratified or antithetic sampling increase the complexity of the sampling [Mi22], while intervention involves modifying data to simulate feature presence. The prediction, especially in resource-intensive models like deep neural networks, is often the most demanding due to repeated model evaluations on modified data. Finally, aggregation can become expensive, for instance, when using methods like weighted least squares optimization in KernelSHAP [LL17]. Accordingly, we aim to spend computational efforts strategically, accepting slightly higher complexity in one step if it allows for better vectorization and parallelization. Understanding the trade-offs in the SIPA framework helps identifying optimization opportunities. Typically, sampling and intervention are of similar complexity, but the prediction can be particularly taxing if many predictions are needed. Alternatively, a more complex aggregation can yield accurate estimates with fewer samples. A scalable SHAP value computation should aim for an efficient preparation, aggregation, and optimization of the prediction process, minimizing redundancy.

Levels of Optimizations: The space for addressing challenges of SHAP value computation is vast, as shown by the extensive literature [Ca17; Ch23; LL17; Mi22; Mo23]. While many existing works focus on solving specific issues of SHAP value computation, we propose a holistic approach that combines multiple strategies. We enhance the efficiency and scalability of SHAP value computation by selecting effective techniques for each SIPA step and carefully optimizing their implementation. The three levels of optimizations are:

- **General Method:** The choice of the general type of method used to compute SHAP values forms the foundation for all other optimizations. We aim to select a method that balances accuracy with the potential for speed and efficiency improvements.
- **Optimized Preparation and Aggregation:** After selecting the general method, we need to optimize preparation and aggregation, including sampling and intervention. We aim to reduce computational overhead and improve the processing speed through reuse of intermediate results and vectorized operations, enabling effective parallelization.
- **Parallelization:** Parallelization is essential and should complement optimized execution. Splitting computations into smaller, concurrent tasks enables multi-threading or distributed computation. The granularity of parallelization—deciding task size and number—is key to performance. For example, parallelizing model evaluations for different feature coalitions can significantly reduce SHAP computation time. Effective parallelization requires balancing tasks and minimizing communication.

Data-centric Optimizations: As an orthogonal optimization opportunity, we can leverage data characteristics to reduce computational demands. For example, feature partitioning is a well-established approach that groups correlated features and computes the sum of their SHAP values with fewer model calls. This method is particularly useful when features are strongly correlated or when we aim to identify feature subsets for explaining predictions. Feature partitioning effectively treats each partition as a single feature during all phases of sampling, intervention, model probing, and aggregation. While our implementation and the SHAP Python package support partitioning, and experiments confirm its effectiveness, in this paper, we focus on holistic optimizations for the general, model-agnostic case.

Additional Related Work: Besides the background, our vectorized and parallelized SHAP value computation is also related to vectorized enumeration-based algorithms, and parallelized pipeline evaluation. First, there are multiple works that effectively vectorize complex computations with linear algebra operations. Examples include vectorized training and scoring of decision trees and random forests [BIJ23; Na20], finding the top-k worst slices (conjunctions of attributes) where an ML model underperforms [SB21], and query processing on ML systems [He22]. Second, similar parallelization strategies have been applied for data cleaning pipelines [SKB23] and AutoML [Sh19]. In contrast, we jointly optimize the algorithm, vectorization, and parallelization of SHAP value computation.

3 Analysis and Optimization Approaches

In this section, we discuss the background and optimization of Antithetic Permutation Sampling, which includes dedicated vectorization and parallelization strategies. We begin by describing design principles and algorithmic details of Antithetic Permutation Sampling in Section 3.1. In Section 3.2, we detail the vectorized implementation, focusing on transforming iterative components into vectorized operations and reusing intermediates to reduce computation per instance. Finally, Section 3.3 discusses parallelization strategies, including fine- and coarse-grained approaches and distributed computation.

3.1 Analysis of Antithetic Permutation Sampling

While there are many more complex methods for SHAP value computation, we use Antithetic Permutation Sampling (APS) combined with omitting features through replacement with features from a background dataset [ŠK14]. This sampling method is a powerful technique used in the computation of SHAP values with the major benefits of variance reduction, feature omission, reuse of predictions, and simplicity of design [Mi22; SP23]. APS builds on the `ApproShapley` algorithm [CGT09] and its application to ML models [ŠK10] in order to improve the efficiency and accuracy of Monte Carlo estimations for Shapley values. Key features are an efficient sampling process and the reuse of predictions for SHAP values of multiple features. ASP is also the default method for fully model-agnostic SHAP value computation in the SHAP Python package [Lu].

Algorithmic Benefits: One of the primary advantages of permutation sampling is its ability to reduce variance. By generating pairs of negatively correlated permutations—specifically, a forward permutation and its reverse—the method leverages the principle of negative covariance. These pairs cancel out random fluctuations, resulting in more stable and reliable estimates. This reduction in variance enhances the convergence rate compared to the naïve Monte Carlo estimator (providing more accurate SHAP values with fewer samples) [SP23]. Another major benefit of APS is the effective handling of feature omissions. When features are omitted, the model must simulate their absence in the context of SHAP values, estimating the prediction by marginalizing over a subset of possible values. APS ensures that when a feature is omitted in one permutation, its corresponding value in the reverse permutation

Algorithm 1 Structure of Antithetic Permutation Sampling

```
Require: $J, B$                    ▹ A set of Instances $J$ and a background dataset of instances $B$
Ensure: $\phi_j \forall j \in J$              ▹ SHAP values for all instances in $J$
1: $T \leftarrow n \cdot m$ random instances from background dataset $B$
2: $P \leftarrow m$ random permutations
3: $\mathbf{M} \leftarrow$ prepareIntermediateMask($P$)
4: $\mathbf{S} \leftarrow$ prepareIntermediateBackgroundSamples($T, \mathbf{M}$)
5: for $\forall j \in J$ do
6:     $X \leftarrow$ applyIntermediates($j$, $\mathbf{M}$, $\mathbf{S}$)    ▹ X is a set of prepared instances
7:     $P \leftarrow$ predict($X$)
8:     $\phi_j \leftarrow$ aggregatePreditions($P$)
9: end for
```

captures the necessary marginal effects. Furthermore, APS allows for efficient reuse of predictions as well. Since permutations add one feature after another, the prediction for the coalition for the prior feature can be reused as the base term when computing the contribution of the following feature. This reuse minimizes the number of model evaluations required, as the same prediction results can be leveraged in multiple contexts. Finally, APS is also a strikingly simple algorithm. The sampling process involves generating pairs of permutations, preparing instances, computing model predictions, and averaging the results. This straightforward approach facilities the integration into existing systems and aids debugging and optimization. At the same time, its theoretical foundation is well-understood, enabling clear explanations and justifications for its use across various applications.

Foundations: The base algorithm combines `ApproShapley` [CGT09] for general Shapley values, and Monte Carlo sampling to marginalize over the omitted features of ML models [CGT09; ŠK10; ŠK14]. We primarily adhere to the original algorithm [ŠK14] but focus on replacing iterative loops with vectorized linear algebra operations wherever possible. For the sake of scalability to many instances, our approach prepares sets of synthesized instances collectively, which is feasible because the preparation tasks are the same for all instances. Since SHAP values for each instance are typically computed independently, this unified approach streamlines the process. In order to facilitate this one-time preparation, we divide the preparation process into two parts. First, two intermediates are computed (which are consistent across all instances) for sampling and intervention. Second, these precomputed intermediates are applied to all instances.

Algorithm Design: Algorithm 1 shows the general structure of our APS implementation. We take a set of instances of interest J and a background dataset B as input, and compute SHAP values ϕ_j for all features of each instance $j \in J$. First, we sample $n \cdot m$ random instances from the background dataset, and obtain n instances for every one of the m sampled random permutations. Subsequently, in Line 3, we prepare a boolean mask for the permutations, representing the coalitions created by adding the features from each permutation and its antithetic counterpart. The procedure in Line 4 prepares the background samples for marginalization over each coalition's omitted features using the boolean mask

M. With this large intermediate prepared, we iterate over all instances of interest. The preparation then comprises applying the intermediates to each instance in Line 6, which creates the synthesized instances. These instances are used to probe the model in Line 7. Finally, the predictions are aggregated in Line 8, marginalizing the omitted features and computing the feature contributions. This method trades computational complexity for space complexity to facilitate one-time preparation. Our results indicate that reducing the sampling step and most of the intervention steps to a one-time cost amortizes quickly and the space overhead is moderate. As an alternative, one could compute the SHAP values only for one permutation or batches of background samples at a time. The straightforward aggregation step from permutation sampling remains unchanged, leaving us with three challenges: splitting up the preparation, computing the intermediates quickly, and managing the larger number of required predictions compared to methods like KernelSHAP [LL17].

3.2 Vectorized Implementation

For optimizing preparation and aggregation, the iterative components of sampling, intervention, and aggregation can be vectorized. Additionally, we separate the preparation into instance-independent and -dependent parts using two matrix operators for permutations and background data. These intermediates can then be applied to all instances at once. We will use matrices from now on because we aim to rewrite the iterative parts as vectorized, linear algebra operations. Feature values are the columns, and instances are rows in such matrices. In this section, we will explain the steps for computing SHAP values using our APS method, as outlined in Algorithm 1. First, we describe the preparation steps (Lines 1-4). Subsequently, we also describe the functions applyIntermediates(), predict() (black box model, but typically already vectorized), and aggregatePredictions().

Preparation of Intermediates: Reducing the amount of redundant computation during sampling and intervention is crucial. In the permutation sampling method, these steps consist of sampling a given number of permutations and samples from the background dataset, as well as constructing synthetic instances by picking features from the instance of interest and an instance from the background dataset. The samples can be reused for every instance. Since the samples only depend on the permutations, the coalitions (dictating which features are used from the background and which from the instances) also stay the same. Therefore, we use a masking approach, which stores the structure of the synthetic instances in a boolean mask and prepares the sampled instances simultaneously. This mask and the prepared samples result in two large but sparse matrices that can be applied via a simple, element-wise matrix multiplication and addition. This masking is inspired by the simplified binary vectors [LL17]. The preparation consists of sampling background data and permutations (a), preparing boolean masks (b), and finally, preparing the background samples with the boolean mask (c, d) as described in detail below.

a) Sampling from the Background Dataset and Permutations: The first two sampling steps are fairly straightforward. A uniform sample of instances from the background can be picked by shuffling the instances and picking the first $n \cdot m$. However, for a reasonably large number of marginalization samples n, it is common to reuse the same samples for all permutations [Lu]. Permutations are sampled by shuffling the feature indices m times and storing the resulting rows of indices in a matrix. For instance, $[3\ 1\ 2]$ is a permutation of three features, where the third feature joins the coalition first, then the first and the second.

b) Preparing Boolean Masks: The synthetic instances for each coalition of features only depend on the permutation. Instead of iterating through the possible coalitions from a permutation to create masks, our method creates the entire matrix of masks for a permutation at once. For one permutation, this mask holds the features in the columns and the coalitions in the rows and is of size $2n$-by-n, where n is the number of features in the permutation. The $0/1$ values indicate if the feature at this index is used in the coalition. The desired mask $\mathbf{M}_p^{fwd}$ for the forward iteration of permutation $\mathbf{p} = [3\ 1\ 2]$ is therefore given by:

$$\mathbf{M}_P^{fwd} = mask(\begin{bmatrix}3 & 1 & 2\end{bmatrix}) = \begin{bmatrix}0 & 0 & 0\\ 0 & 0 & 1\\ 1 & 0 & 1\\ 1 & 1 & 1\end{bmatrix} \tag{1}$$

We compute a contingency table of two carefully crafted vectors to derive this mask from the permutation. The two vectors encode the positions of ones in the boolean matrix as row-column-index pairs, derived from each permutation as the position of a feature. This position defines the row in the coalition and boolean matrix, and the feature index defines the column in the boolean matrix. The column index can be used to create a row indicator vector by taking the lower triangular part of a matrix and reshaping it into a single column, while dropping any zeros. For a permutation of length three, the process looks like this:

$$\begin{bmatrix}\mathbf{1} & 1 & 1\\ \mathbf{2} & \mathbf{2} & 2\\ \mathbf{3} & \mathbf{3} & \mathbf{3}\end{bmatrix} \rightarrow \begin{bmatrix}1 & 2 & 2 & 3 & 3 & 3\end{bmatrix} \tag{2}$$

The column indicator vector is based on the feature indices. This vector is created from a square matrix with the permutation replicated in its rows. Again, the lower triangular part is taken, flattened and the zeros are dropped, resulting in $[3\ 3\ 1\ 3\ 1\ 2]$ for the example permutation $[3\ 1\ 2]$. The contingency table of these two vectors is given by:

$$\text{table}\left(\begin{matrix}\begin{bmatrix}1 & 2 & 2 & 3 & 3 & 3\end{bmatrix},\\ \begin{bmatrix}3 & 3 & 1 & 3 & 1 & 2\end{bmatrix}\end{matrix}\right) = \begin{bmatrix}0 & 0 & 1\\ 1 & 0 & 1\\ 1 & 1 & 1\end{bmatrix} \tag{3}$$

To create the mask shown in Equation 1, we prepend $[0\ 0\ 0]$. However, we also want to generate the mask for the antithetic counterpart to permutation p. To do so, we take the complement of the matrix from Equation 3, reverse its rows, and append it below. The fact that we use the complement of the mask created for the forward iteration for the masks of the backward iteration indicates that we are generating antithetic samples of coalitions. This process results in the matrix $\mathbf{M}_p$ of size $2n \times n$ for permutation p of length n:

$$\mathbf{M}_p = \begin{matrix} \left.\begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 1 \end{bmatrix}\right\} \text{forward mask created from contingency table} \\ \left.\begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & 1 & 0 \end{bmatrix}\right\} \text{reversed complement} \end{matrix} \tag{4}$$

This matrix holds all coalitions of interest for a given permutation [3 1 2] and has the following structure: the first row only selects the first feature (3), the second row selects the first two features (3 and 1), and so on. The row of zeros on top simplifies the aggregation step. Vertically stacking $\mathbf{M}_p$ for all permutations $p \in P$ yields a single masking matrix $\mathbf{M}$.

c) Replicating and Preparing Samples for Marginalization: To marginalize out the omitted features over m samples from the background dataset for each permutation, the boolean mask must be replicated row-wise, while the matrix holding the background samples has to be replicated block-wise. Row-wise replication by m increases the size of this boolean matrix to $m2n$-by-n for n features. This row-wise replication for any matrix with s rows is done by replicating a sequence vector $(1 : s)$ over m columns, reshaping into a column vector, and one-hot-encoding. The resulting selection matrix replicates any matrix with s rows m times through matrix multiplication from the left (once for all permutations). The samples can be replicated $2n$ times in vectorized form as well. First, the matrix of size m-by-n is reshaped into a single row, then replicated using broadcasting into a matrix of size $2n$-by-mn, and then reshaped into a matrix S_p of size $2nm$-by-n. This sequence is done in parallel for all permutations. We are left with two large matrices: one for the boolean masks of a given permutation $\mathbf{M}_p$, and one containing the replicated background samples $\mathbf{S}_p$. While the boolean mask already has a 50% sparsity, we can further reduce the size of the replicated samples by applying the complement of $\mathbf{M}$, turning it into a 50% sparse matrix. The final size of $\mathbf{S}$ and $\mathbf{M}$ for n features and m samples, prepared for all p permutations is

$$dim(\mathbf{S}) = dim(\mathbf{M}) = (\underbrace{p}_{\text{\#permutations}} \cdot \underbrace{2n}_{2n \text{ coalitions}} \cdot \underbrace{m}_{\text{\#samples}}) \times \underbrace{n}_{\#features} \tag{5}$$

d) Application on each Instance: Due to the simple intermediates, all that is left to create the synthetic instances $x_{synth} \in X$ is to apply the boolean mask $\mathbf{M}$ to the instance x (with broadcasting of x) and element-wise adding the result to the masked samples $\mathbf{S}$:

$$X = \underbrace{\mathbf{S}}_{\text{prepared background samples}} + (\underbrace{\mathbf{M}}_{\text{boolean mask}} \cdot \underbrace{x}_{\text{instance}}) \tag{6}$$

This step creates $p \cdot 2nm$ synthetic instances for every instance, allowing fast preparation of thousands of instances x using $\mathbf{M}$ and $\mathbf{S}$. These instances are then scored by the model to get predictions. If the masks and samples are too large for a system, the replications can also be created for batches of synthesized instances and predictions.

Aggregation: Once the model computed predictions for the generated instances, we aggregate these results. This step consists of computing the averages of each replicated instance for marginalization over the omitted features, and the actual computation of the SHAP values ϕ_i for each feature i. Computing the means for each batch of samples is done by reshaping the predictions P so that each batch is in one row and computing the mean per row. Each batch was created from a row in $\mathbf{M}$ (same coalition), and the batches are stacked in an aligned manner in X and P. An example of this step for coalitions a and b is

$$P_{means} = \text{batchMeans}\left(\begin{bmatrix} a_1 \\ a_2 \\ a_3 \\ b_1 \\ b_2 \\ b_3 \end{bmatrix}\right) \overset{\left(\substack{\text{using reshape} \\ \text{on vector}}\right)}{=} \text{rowMeans}\left(\begin{bmatrix} a_1 & a_2 & a_3 \\ b_1 & b_2 & b_3 \end{bmatrix}\right) = \begin{bmatrix} \overline{a} \\ \overline{b} \end{bmatrix} \tag{7}$$

With this vector P_{means} of marginalized predictions for each coalition, the feature contributions to the coalition can be computed by a matrix difference of the vector with itself, shifted by one row. To allow a single vectorized operation, the row of zeros in $\mathbf{M}_p$ (see Equation 4) is moved to the top. For example, we obtain the mapping $g : \mathbb{P}_p \to \mathbb{C}_p$ from predictions $p_i \in \mathbb{P}_p$ in the rows, to coalitions $c \in \mathbb{C}_p$ of permutation $p = [3\ 1\ 2]$ as:

$$g : \mathbb{P}_p \to \mathbb{C}_p \Rightarrow \begin{bmatrix} p_0 \\ p_1 \\ p_2 \\ p_3 \\ p_4 \\ p_5 \end{bmatrix} \begin{matrix} \mapsto & \{\varnothing\} \\ \mapsto & \{3\} \\ \mapsto & \{3,1\} \\ \mapsto & \{3,1,2\} \\ \mapsto & \{2\} \\ \mapsto & \{2,1\} \end{matrix} \tag{8}$$

From this mapping, we derive that the contribution of feature i in the forward pass (p_0 to p_3) and in the backward pass (p_4 to p_5) is given by $p_i - p_{i-1}$. We compute this difference in a shifted difference as $\phi_p = P_p[1 : n] - P_p[0 : n-1]$, leaving us with two rows that need fixing: the rows for the first and last element of the backward pass. The first element uses the full coalition for its difference. However, it should reuse the empty coalition from the first row. The contribution of the last element uses the last element of P_p and reuses the element of the full coalition. We calculate them separately to facilitate the fast calculation of all the other contributions. This procedure is done once for all permutations by reshaping the marginalized predictions into a matrix so that the predictions for one permutation are in one column. After averaging the forward and backward pass, the resulting matrix is reshaped back into a column vector. This vector now holds the SHAP values for all permutations stacked vertically. From this vector, the SHAP values for each feature are

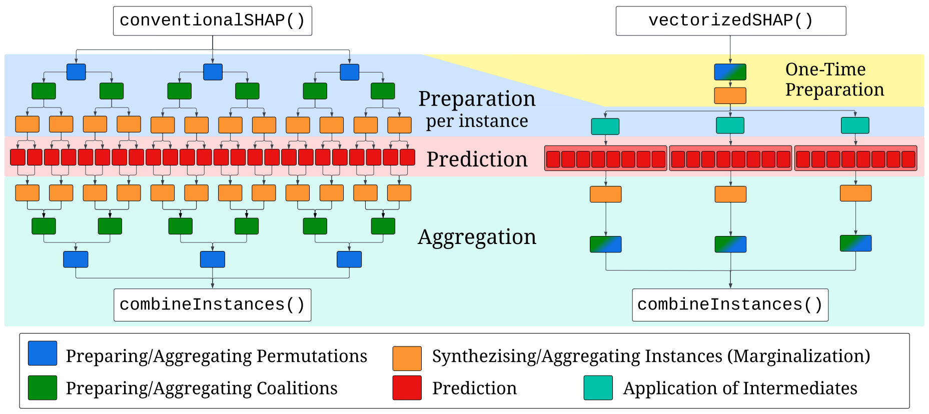
conventionalSHAP()
vectorizedSHAP()
One-Time Preparation
Preparation per instance
Prediction
Aggregation
combineInstances()
combineInstances()
Preparing/Aggregating Permutations
Synthezising/Aggregating Instances (Marginalization)
Preparing/Aggregating Coalitions
Prediction
Application of Intermediates

Fig. 1: Comparison of Operator Graphs. On the left, the conventional method iteratively prepares and aggregates permutations, coalitions, and synthetic instances for each instance of interest. On the right, the vectorized implementation prepares all permutations, coalitions, and background samples for marginalization only once and applies this to every instance of interest.

aggregated according to the order of features in the permutations and averaged. Finally, we obtain a single vector ϕ with the estimated SHAP value for each feature i at index i.

Reducing Redundancy: By preparing a large matrix operator once and applying it to each instance, we reduce the amount of redundant preparation tasks. Figure 1 compares the conventional computation (on the left) with our vectorized computation (on the right). On the left, the steps consist of preparing permutations (blue), deriving the antithetic coalitions for each permutation (green), and synthesizing instances for each coalition by creating multiple instances using background samples (orange). On the right, this nested structure is reduced into a one-time preparation and an application step (turquoise) per instance. The prediction in the red area is still of the same size. However, the method allows larger batches and thus, improved parallelization. During aggregation, averaging predictions for marginalization (orange), computing contributions per coalition (green), and aggregating results for the permutations (blue) are reduced into two consecutive operations per instance. The vectorized design exhibits fewer operations for preparation and aggregation, fewer synchronization barriers, and ultimately lower overhead. Using more permutations or background samples only increases the size of intermediates but not the number of operations.

3.3 Parallelization

Parallelization is crucial in optimizing the computation of SHAP values, mainly due to the computational intensity of the prediction step. Through vectorization, the tasks involved in preparation, prediction, and aggregation can be parallelized. At the same time, this

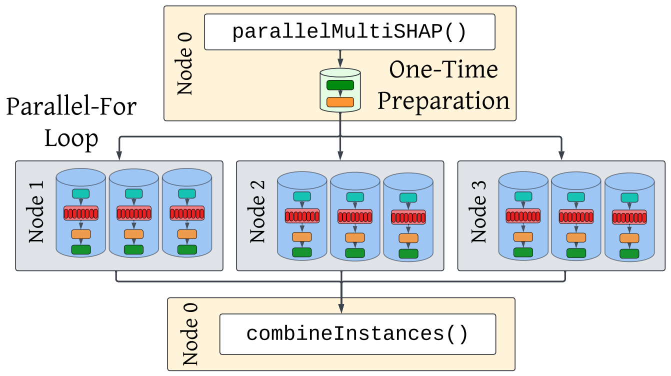

Fig. 2: Parallel Execution of Computations for each Instance (once the intermediates are prepared, the operations for each instance can be executed in a task-parallel manner on a cluster of nodes).

high-level pipeline can run in parallel on many instances. Parallelization may introduce additional overhead though for coordinating parallel tasks and communicating intermediates. Effective parallelization strategies mitigate the computational demands of SHAP value computation, making it feasible to apply this method on large datasets and complex models.

Granularity: A key aspect is the parallelization granularity in terms of task sizes. Fine-grained parallelism involves breaking down tasks into very small subtasks, potentially leading to high overhead in task management and communication. Coarse-grained parallelism, on the other hand, uses larger subtasks, which can be more efficient but might not fully utilize available resources and face load balance issues. Finding the right balance is essential. For SHAP values, parallelization can be applied at various stages and three levels of granularity:

- **Algorithm Level:** The entire computation process can be parallelized by distributing the computation for individual instances over many nodes in a data-parallel manner.
- **High-level Operators:** Specific subroutines within the SHAP computation, such as the prediction steps for multiple coalitions, can be parallelized. This approach allows for efficient execution of independent sub-tasks within the overall algorithm.
- **Operations:** Furthermore, we can parallelize individual operations, such as matrix multiplications and other linear algebra operations in local or distributed environments.

Since each instance only depends on the background dataset and not on any other instance, we could partition the instances and let each worker compute the SHAP values for one partition. However, this approach would introduce unnecessary overhead since the preparation step would have to be executed again for each instance, or at least for each partition.

Coarse-grained Parallelization: We propose combining as many of the subroutines as possible into independent pipelines. These pipelines can be executed in a parallel for-loop [Bo14], employing coarse-grained task parallelism without synchronization at operation granularity. As described in Section 3.2, the redundancy reduction is achieved by preparing boolean masks and background samples before computing SHAP values for each instance of interest. These are also prepared in a parallel for-loop over the permutations. Additionally, this coarse-grained data-parallel execution implicitly parallelizes model calls since pipelines run in parallel. The parallel pipelines can be executed in a multi-threaded fashion on a single node or on a cluster of nodes. Figure 2 depicts the coarse-grained parallelization. After the intermediates are prepared, each worker gets the intermediates and a subset of the instances. Using the intermediates, a pipeline (blue) is executed for each instance. The results are finally aggregated. If the distribution introduces too much overhead, all pipelines can also be executed on a single node using multi-threading. We utilize SystemDS's `parfor` construct [Bo14] and Spark backend [Bo16]. SystemDS compiles the high-level algorithm into efficient execution plans, handles the necessary data exchange between backends, and also partitions the instances of interest and assigns batches to workers.

Fine-grained Parallelization: In a more fine-grained manner, vectorized operations already allow for multi-threaded or distributed execution. Vectorization involves converting operations into a format where multiple data points are processed together, which internally uses different means of parallelization but with synchronization barriers after every operation. This transformation allows the system to perform optimizations such as reordering and fusing operations to maximize efficiency. Vectorized operations simplify parallelization, better exploit the hardware capabilities, and avoid unnecessary allocations. As we established in Section 3.2, the steps within each parallel pipeline are vectorized, allowing for fine-grained parallelization on the operations level. Using SystemDS' scripting language DML, we use high-level abstractions for vectorized tasks, but rely on the SystemDS optimizer to identify optimization opportunities in linear algebra computations, and to compile efficient execution plans [Bo20]. Each pipeline calls the model once with a single batch of all its generated instances simultaneously. Despite the black-box nature of model predictions (called through second-order `eval`), SystemDS can still collapse these unnecessary abstractions during compilation and optimize the parallelization of model predictions as well.

4 Experiments

In this section, we study the accuracy and runtime of our optimized SHAP value computation method, including a comparison with the state-of-the-art SHAP Python package.

4.1 Experimental Setting

Hardware Setup: All experiments were conducted on a 1+8 node cluster with each node comprising an AMD EPYC 7443P processor at 2.8 GHz, featuring 24 physical cores, 48 virtual cores, and 256 GB of DDR4 memory. The operating system was Ubuntu 20.04. To ensure a fair comparison, both our implementation and the baseline SHAP Python package

Tab. 1: Datasets and the ML Models used (one-hot encoding includes binning, where applicable).

Characteristics	**Adult**	**US Census (1990)**
Instances	32,561	2,458,286
Features	14	68
Features (One-Hot Encoded)	107	371
ML models	multiLogReg, FNN	linear SVM

were run on a single server, as the baseline does not support distributed operations. For the scalability experiments, we then additionally use the entire cluster.

Implementation: The proposed algorithm is implemented as a DML script (with R-like syntax) on top of Apache SystemDS [Bo20] and available as a builtin function[3]. The code of the experiments is also available in the corresponding reproducibility repository[4].

Datasets & Models: To evaluate the performance of our SHAP value computation, we trained and explained three ML models: a multinomial logistic regression, a linear support vector machine (SVM), and a feed-forward neural network (FNN). All models were applied to classification tasks for which we selected two benchmark datasets from the UCI ML Repository. Table 1 shows the data characteristics and models used for each. The Adult dataset consists of 14 categorical and numerical features to predict whether an individual's income exceeds $50k [BK96]. The US Census dataset contains 68 categorical features and aims to identify clusters within the data [MTH]. Both datasets are preprocessed by binning numerical features into ten equi-width bins and one-hot encoding all features. The Adult dataset comes with predefined binary labels. For the US Census dataset, we use k-means clustering to label the data into four clusters and then apply multi-class classification. Logistic regression was used on the Adult dataset due to its simplicity, which allowed us to compute exact SHAP values for comparison based on the coefficients. The SVM and FNN were trained on the UC Census dataset, where the FNN uses a sequence of dense, dropout, and dense layers with ReLu and Sigmoid activation. These models also allow a seamless conversion between SystemDS and Python, facilitating a direct comparison with the SHAP Python package. To evaluate the influence of model complexity when analyzing weak scaling, the FNN was additionally trained with one to eight hidden layers.

Baseline SHAP Python Package: Computing SHAP values for ML models is almost synonymous with using the SHAP Python package, introduced by Lundberg and Lee in 2017 along with the SHAP framework [LL17]. This package is still maintained and new features are regularly added. With nearly 6.7 million downloads per month[5], it is likely the most widely adopted collection of methods for computing SHAP values. The package aims to detect the model type and select the best-suited explainer automatically, but defaults to the

[3] SystemDS Repository: https://github.com/apache/systemds
[4] Reproducibility Repository: https://github.com/damslab/reproducibility
[5] As presented on the repository as of June 13th, 2024: https://github.com/shap/shap

Tab. 2: MSE for Adult and Census datasets from the parallel method and the SHAP Python package baseline. MSE was calculated over 100 computations of 50 instances at 100 samples and 10 permutations. Ground truth *large* is from Python package with 1000 samples and 500 permutations.

Model & Dataset	**Our Method exact / large**	**Python exact / large**
LogReg on Adult	4.5e-05 / 4.3e-06	4.1e-05 / 4.4e-06
SVM on US Census	N/A / 2.5e-05	N/A / 1.9e-05

`PermutationExplainer` (which performs APS) if no model-specific explainer is available. For our experiments, we force the use of the `PermutationExplainer` as well as the number of permutations and background samples to ensure a similar workload.

4.2 Accuracy Comparison

Our overall goal was a method that scales better to many instances without sacrificing accuracy. We compute SHAP values for the LogReg and the SVM models using our parallel method and the `PermutationExplainer` from the SHAP Python package. We select LogReg and SVM because these models are largely identical across implementations and produce the same predictions for the same instances. Additionally, we selected a single instance for comparing the computed SHAP values by both methods in detail.

Overall Accuracy: Table 2 shows the overall accuracy in terms of mean squared error (MSE) of SHAP values computed using both implementations on the LogReg and SVM models. These values were averaged over 100 computations for the same 50 instances and same 100 background samples. As ground truth, we used the SHAP values from the SHAP package but with 1,000 samples and 500 permutations, assuming these values have converged close to the actual values. For reference, computing SHAP values with this many permutations and samples took six hours for the SVM on the US Census dataset, highlighting the importance of achieving low errors with substantially reduced runtime. In contrast, for LogReg, we use the model-specific `LinearExplainer` from the SHAP package to compute an exact ground truth from the coefficients, scaled to the probability space. Both implementations exhibit a similar MSE for all models and datasets. The magnitude of 10^{-5} is small compared to the absolute size of SHAP values (up to 0.6 for Adult, 0.7 for Census). The reference method shows a slightly lower MSE than our parallel method though.

SHAP Value Quality: Having established that both methods achieve similar overall MSE, we now conduct a qualitative analysis. We compare our SHAP values to those computed by the `PermutationExplainer` for a randomly selected example instance of the Adult dataset. Figure 3 shows the SHAP values with the feature indices on the x-axis. The blue bars represent the values computed by our parallel implementation in SystemDS, the orange bars indicate the values computed by the SHAP Python package, and the green bars show the

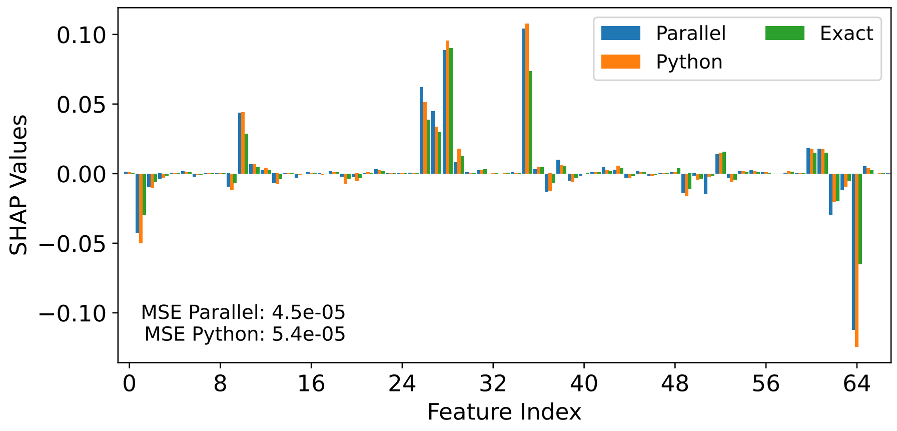

Fig. 3: Accuracy of SHAP values for all features of Adult. Features with indices above 65 have insignificant SHAP values and were omitted for clarity.

rescaled exact SHAP values derived from the coefficients. Both implementations compute similar SHAP values for each feature. Interestingly, both APS methods are more similar to each other than to the exact values. This discrepancy may be due to scaling errors when converting the exact values from log-odds to the probability space used by the other methods. The MSE was 4.5e-05 for our method and 5.4e-05 for the Python method.

Discussion: Our new parallel method exhibits similar accuracy as the SHAP package with errors in the order of 10^{-5}. These minor errors could be attributed to numerical precision issues from the large sums involved in the marginalization and averaging steps. Our parallel method accumulates larger quantities before aggregation but relies on Kahan addition for numerically stable aggregations. These results demonstrate the accuracy of our method. An exemplary study of the SHAP values also showed qualitatively similar results.

4.3 Local Runtime Comparison

We now evaluate the end-to-end runtime for different optimization levels and compare these variants of our method with the SHAP `PermutationExplainer`. All SHAP values of these local runtime experiments were computed using three permutations and 100 samples from the background dataset. Further increasing the number of samples or permutations does not significantly reduce the MSE. We report the mean of 3 repetitions.

Levels of Optimization: We evaluate different variants of our method, each representing a different level of optimization. Every one of these methods runs in parallel on multiple instances of interest. In our single-node setup, parallelization and vectorized execution can reduce runtime by efficiently utilizing all available cores.

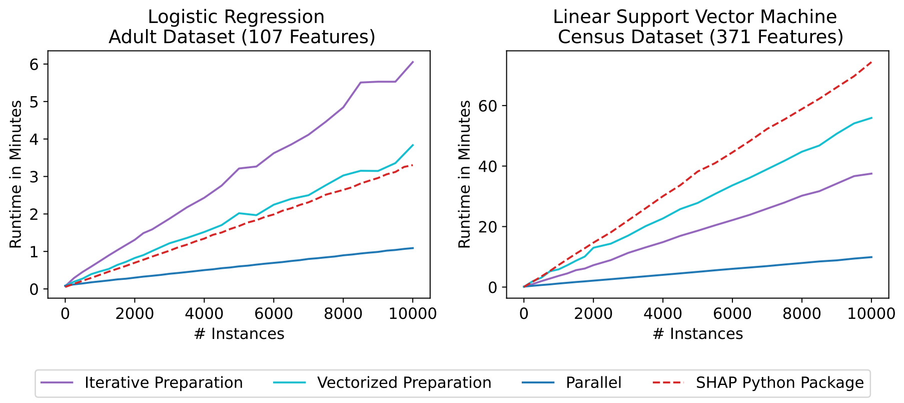

Fig. 4: Runtime for increasing numbers of instances. Left: LogReg on the Adult dataset with 107 one-hot-encoded features; Right: SVM on the Census dataset with 371 features.

- **Iterative:** Computes SHAP values the conventional way by iterating through permutations, their features, and coalitions for each instance individually.
- **Vectorized:** Computes SHAP values using vectorization but always prepares a new set of permutations and masks for each instance.
- **Parallel:** Our full method prepares intermediates for all permutations in parallel, computes SHAP values using vectorization, and reuses prepared intermediates.

Overall Runtime: Figure 4 shows the local runtime for the different variants in SystemDS as well as the baseline. The left plot shows the runtime for LogReg on Adult, whereas the right plot shows the runtime for SVM on Census. Both models use a simple matrix-vector multiplication on the features for prediction. This simplicity—and similarity in SystemDS and Python—allows us to effectively isolate the influence of the SHAP value computation method. On the x-axes, we vary the number of instances, and the y-axes show the corresponding runtimes. The red dashed lines indicate the runtime of the Python implementation. All runtimes linearly increase with more instances, but with different slopes. In both scenarios, our parallel method consistently outperforms the `PermutationExplainer`. For the Adult dataset, the iterative method performs worst, followed by the method that prepares and aggregates using vectorized operations only but without reuse of intermediates. The Python baseline is only slightly faster. The method based on vectorization and parallel reuse of the intermediates is the fastest. For US Census—which has about three times the number of features—even the variants with the fewest optimizations outperform the baseline. Although the final parallel method (which reuses intermediates) is again the fastest, the results also demonstrate that the vectorized method is slower than the iterative method for this dataset. This effect may be due to the overhead introduced by rewriting operations as vectorized operations (creation of large vectors, reshape operations), and potentially large allocations and evictions. However, in the final parallel method, this creation only needs to

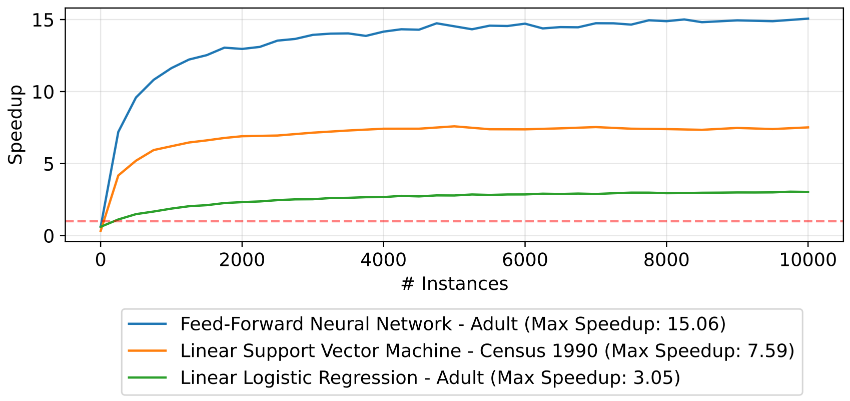

Fig. 6: Speedup of Parallel Permutation Sampling, compared to the SHAP package.

be executed once, the result is reused for every instance, and thus, easily amortized. The combination of techniques in our parallel variant shows robust performance across datasets and more than an order of magnitude runtime improvements compared to SHAP.

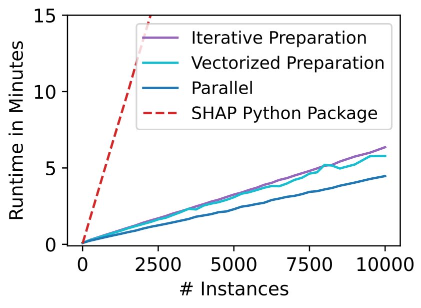

Fig. 5: Runtime for an FNN on the Adult dataset with 107 features.

Model Impact: To determine the impact of the model, we additionally trained a more complex FNN for classification on Adult. This model's prediction step is more compute-intensive. Even though there are fast, model-specific methods for computing SHAP values for linear models, neural networks can only be explained by model-agnostic methods like APS. Figure 5 shows the runtime for different numbers of instances. Similar to our previous results, the runtime increases linearly. In contrast to LogReg though, we observe an overall increase in runtime, likely due to the more complex prediction. Our variants all outperform the SHAP package by a large margin, and our parallel method with reuse in blue is still the fastest. The iterative and vectorized methods—each performing independent preparation for each instance—exhibit similar results as before. However, both methods have runtimes that are closer to the parallel method with reuse, indicating that as prediction complexity increases, the proportion of runtime attributed to predictions grows, thereby diminishing the relative impact of optimizations in the preparation and aggregation steps.

Speedup: We further compute the speedup of our parallel method over the `PermutationExplainer` for different numbers of instances. Figure 6 shows these speedups, where the red dashed line represents a speedup of one, marking the threshold of performance improvements. With the fastest prediction, LogReg shows the smallest speedup, converging

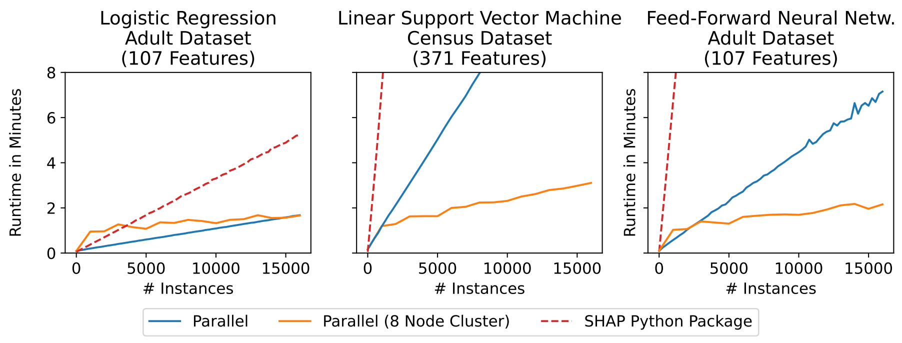

Fig. 7: Distributed Runtime Comparison for Different Numbers of Instances, Models, and Datasets.

to 3x at around 8,000 instances. The highest speedup is achieved for the FNN, with a speedup above 14x beyond 3,000 instances. The SHAP package is faster for very small numbers of instances (speedup below the red line). However, even for LogReg with the lowest speedup, our method surpasses this threshold at around 250 instances. Models with more features or complex prediction steps exceed this threshold even sooner.

4.4 Distributed Runtime Comparison

Having established the performance benefit of effective parallelization in a single-node environment, we now explore the scalability in distributed environments. Our design—of utilizing independent pipelines per instance through SystemDS's parfor-loop—allows the workload to be distributed across multiple executors in a scale-out Spark cluster as well. We use the same models and datasets as before but run the parfor-loop in remote Spark mode.

Runtime: Figure 7 shows the runtime for computing SHAP values across different models and datasets: LogReg on Adult (left), SVM on Census (center), and FNN on Adult (right). We compare the runtime of our method on a single node and an eight-node cluster, as well as the SHAP package (which runs on a single node though). For LogReg, there is a break-even point for using distributed operations at around 15,000 instances. Before this point, the reduced overhead of the single-node configuration (e.g., no Spark context creation) results in faster runtime. The SHAP package is also faster than distributed operations for up to 2,700 instances. For SVM on Census, distributed operations already pay off for few instances, outperforming the single-node methods substantially. Compared to SHAP, which takes over 20 minutes to compute 2,500 instances, the distributed method completes 15,000 instances in less than 3 minutes. We observe a similar pattern for FNN on Adult, which has a more complex prediction step. As with LogReg, single-node execution is faster for few instances but is surpassed at around 2,500 instances. Beyond this point, distributed execution is always faster when computing SHAP values for all 107 features.

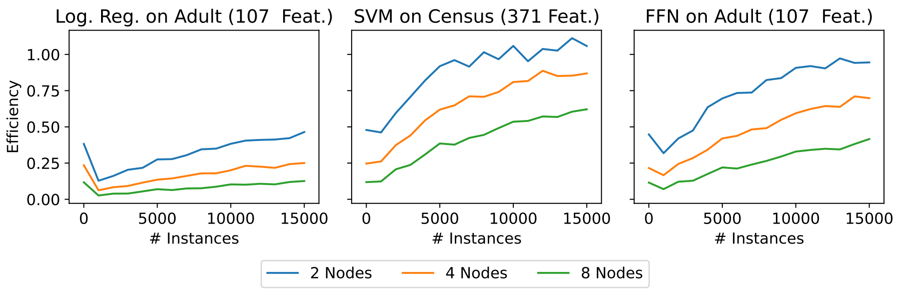

Fig. 8: Distributed Scaling Efficiency of Parallelized Permutation Sampling.

Scalability: To quantify the scalability of our method, we use scaling efficiency as a metric how well additional nodes are utilized. We define this efficiency as $\eta = \frac{S_n}{n}$, where S_n is the speedup achieved using n nodes compared to using the same algorithm on a single node. Figure 8 shows the efficiency with increasing number of instances and again for LogReg on Adult, SVM on Census, and FNN on Adult. Across all models, scaling efficiency increases with the number of instances and understandably decreases with the number of nodes because it becomes harder to fully utilize them. LogReg shows generally low efficiency for all configurations due to the short runtime for single-node execution. Distributed computation only becomes faster at around 15,000 instances. The peaks in efficiency at one instance are due to SystemDS falling back to single-node execution in this case, yielding an efficiency of $\frac{1}{n}$ for n nodes in the configuration. The middle plot shows higher efficiency for the larger SVM on Census. The two-node cluster even surpasses an efficiency of one, indicating very good scalability. On the right, FNN on Adult—with a more complex prediction step—shows also good efficiency which increases faster than LogReg with the number of instances. At 10,000 instances, we achieve a scaling efficiency close to one. Furthermore, weak scaling evaluates how well a method handles increasing problem sizes proportionally to the number of nodes, maintaining a constant problem size per node. We independently scale the number of instances for the SVM, the number of features for SHAP computation on the same SVM, and the number of hidden layers in the FNN on Adult, while keeping other parameters (30k instances and all features) constant. As shown in Figure 9, single node runtime increases linearly with the size of the problem, while the runtime remains nearly constant when scaling the number

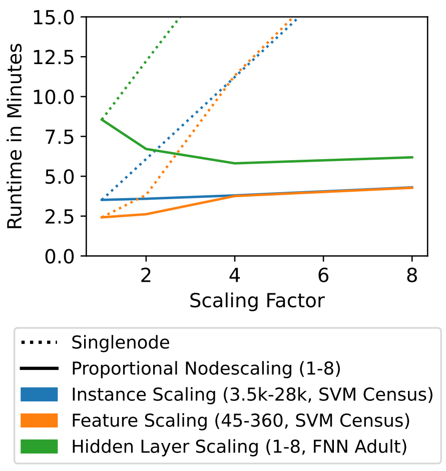

Fig. 9: Weak Scaling of Distributed Computation.

of nodes. For the FNN, runtime even drops slightly. This result shows that models with complex prediction steps benefit significantly from parallelization.

Discussion: The results demonstrate substantial improvements of our parallel method in end-to-end runtime for all use cases and good scalability for large and complex models. Moreover, we also observe substantial performance improvements compared to the SHAP Python package across all models. While the benefits of specific optimization components may vary depending on model characteristics, combining optimizations at all levels results in a method that robustly outperforms the current state-of-the-art. Models with more features or more complex prediction steps, such as SVM on Census or FNN on Adult, benefit more than simple models from our scalable implementation. In terms of scaling efficiency, there is room for improvements of the underlying SystemDS components (which we used without dedicated modifications) as well as potential for scaling to larger clusters (because the efficiency did not converge), larger data sizes, and large number of instances. Distributed operations brought the runtimes down to below 3 minutes in all our experiments, which means that one-time costs of Spark context creation ($\approx$ 20 seconds), communication, and scheduling consume a substantial portion of the overall runtime. The near-constant runtime under weak scaling demonstrates strong scalability across various problem dimensions, such as the number of instances, features, or model complexity. From these results, we conclude that our design substantially reduces end-to-end runtime on a single node, and larger and more complex models benefit from good scalability in distributed environments.

5 Conclusions

We devised a scalable, model-agnostic method for computing SHAP values on top of Apache SystemDS. To this end, we use Antithetic Permutation Sampling as a solid foundation due to its sampling efficiency and optimization potential. Key features of our approach include reusing intermediates, reducing redundancy, as well as careful vectorization and parallelization. When applying these features together, we achieve similar accuracy to the SHAP Python package with speedups of up to 14$\times$ and 35$\times$ for local and distributed operations, respectively. In conclusion, our scalable SHAP sampling enables the efficient computation of SHAP explanations, especially for complex models and large datasets. More broadly, this work is also an example of mapping complex, enumeration-based algorithms to linear algebra and then utilizing ML systems to generate efficient local plans or scalable distributed plans when needed. Interesting future work includes a broader set of sampling algorithms, and pushing the envelope of scalability via a spectrum of dedicated parallelization strategies (task-parallel, data-parallel, hybrid strategies).

6 Acknowledgments

We gratefully acknowledge funding from the German Federal Ministry of Education and Research (under research grant BIFOLD24B).

References

[BIJ23] Boehm, M.; Interlandi, M.; Jermaine, C.: Optimizing Tensor Computations: From Applications to Compilation and Runtime Techniques. In: SIGMOD. Pp. 53–59, 2023, DOI: 10.1145/3555041.3589407.

[BK96] Becker, B.; Kohavi, R.: Adult, UCI Machine Learning Repository, 1996.

[Bo14] Boehm, M.; Tatikonda, S.; Reinwald, B.; Sen, P.; Tian, Y.; Burdick, D.; Vaithyanathan, S.: Hybrid Parallelization Strategies for Large-Scale Machine Learning in SystemML. PVLDB 7 (7), pp. 553–564, 2014, DOI: 10.14778/2732286.2732292.

[Bo16] Boehm, M.; Dusenberry, M. W.; Eriksson, D.; Evfimievski, A. V.; Manshadi, F. M.; Pansare, N.; Reinwald, B.; Reiss, F. R.; Sen, P.; Surve, A. C.; Tatikonda, S.: SystemML: declarative machine learning on spark. PVLDB 9 (13), pp. 1425–1436, 2016, DOI: 10.14778/3007263.3007279.

[Bo20] Boehm, M.; Antonov, I.; Baunsgaard, S.; Dokter, M.; Ginthör, R.; Innerebner, K.; Klezin, F.; Lindstaedt, S. N.; Phani, A.; Rath, B.; Reinwald, B.; Siddiqui, S.; Wrede, S. B.: SystemDS: A Declarative Machine Learning System for the End-to-End Data Science Lifecycle. In: CIDR. 2020, URL: http://cidrdb.org/cidr2020/papers/p22-boehm-cidr20.pdf.

[Ca17] Castro, J.; Gómez, D.; Molina, E.; Tejada, J.: Improving polynomial estimation of the Shapley value by stratified random sampling with optimum allocation. Computers & Operations Research 82, pp. 180–188, 2017, DOI: 10.1016/j.cor.2017.01.019.

[CGT09] Castro, J.; Gómez, D.; Tejada, J.: Polynomial calculation of the Shapley value based on sampling. Computers & Operations Research, ISOLDE X 36 (5), pp. 1726–1730, 2009, DOI: 10.1016/j.cor.2008.04.004.

[Ch23] Chen, H.; Covert, I. C.; Lundberg, S. M.; Lee, S.-I.: Algorithms to estimate Shapley value feature attributions. en, Nature Machine Intelligence 5 (6), pp. 590–601, 2023, DOI: 10.1038/s42256-023-00657-x.

[DK17] Doshi-Velez, F.; Kim, B.: Towards A Rigorous Science of Interpretable Machine Learning. arXiv: Machine Learning, 2017, DOI: 10.48550/arXiv.1702.08608.

[DP94] Deng, X.; Papadimitriou, C. H.: On the Complexity of Cooperative Solution Concepts. Mathematics of Operations Research 19 (2), p. 257, 1994, DOI: 10.1287/moor.19.2.257.

[Es21] von Eschenbach, W. J.: Transparency and the Black Box Problem: Why We Do Not Trust AI. en, Philosophy & Technology 34 (4), 2021, DOI: 10.1007/s13347-021-00477-0.

[He22] He, D.; Nakandala, S. C.; Banda, D.; Sen, R.; Saur, K.; Park, K.; Curino, C.; Camacho-Rodríguez, J.; Karanasos, K.; Interlandi, M.: Query Processing on Tensor Computation Runtimes. PVLDB 15 (11), pp. 2811–2825, 2022, DOI: 10.14778/3551793.3551833.

[LL17] Lundberg, S. M.; Lee, S.-I.: A unified approach to interpreting model predictions. In: NeurIPS. Red Hook, NY, USA, pp. 4768–4777, 2017, DOI: 10.48550/arXiv.1705.07874.

[Lo19] Lomelí, M.; Rowland, M.; Gretton, A.; Ghahramani, Z.: Antithetic and Monte Carlo kernel estimators for partial rankings. en, Statistics and Computing 29 (5), pp. 1127–1147, 2019, DOI: 10.1007/s11222-019-09859-z.

[Lu] Lundberg, S.: shap Python Package: A game theoretic approach to explain the output of any machine learning model. Code Repository, URL: https://github.com/shap/shap, visited on: 05/20/2024.

[Lu20] Lundberg, S. M.; Erion, G.; Chen, H.; DeGrave, A.; Prutkin, J. M.; Nair, B.; Katz, R.; Himmelfarb, J.; Bansal, N.; Lee, S.-I.: From local explanations to global understanding with explainable AI for trees. en, Nature Machine Intelligence 2 (1), pp. 56–67, 2020, DOI: 10.1038/s42256-019-0138-9.

[Mi22] Mitchell, R.; Cooper, J.; Frank, E.; Holmes, G.: Sampling permutations for Shapley value estimation. The Journal of Machine Learning Research 23 (1), 43:2082–43:2127, 2022, ISSN: 1532-4435, DOI: 10.48550/arXiv.2104.12199.

[Mo23] Molnar, C.: Interpreting Machine Learning Models With SHAP. Germany, Munich, 2023, ISBN: 9798857734445.

[MTH] Meek, C.; Thiesson, B.; Heckerman, D.: US Census Data (1990), UCI Machine Learning Repository, DOI: 10.24432/C5VP42.

[Na20] Nakandala, S.; Saur, K.; Yu, G.; Karanasos, K.; Curino, C.; Weimer, M.; Interlandi, M.: A Tensor Compiler for Unified Machine Learning Prediction Serving. In: OSDI. Pp. 899–917, 2020, URL: https://www.usenix.org/conference/osdi20/presentation/nakandala.

[OGS20] Ozbayoglu, A. M.; Gudelek, M. U.; Sezer, O. B.: Deep learning for financial applications: A survey. Applied Soft Computing 93, p. 106384, 2020, DOI: 10.1016/j.asoc.2020.106384.

[Ra22] Rajpurkar, P.; Chen, E.; Banerjee, O.; Topol, E. J.: AI in health and medicine. en, Nature Medicine 28 (1), pp. 31–38, 2022, DOI: 10.1038/s41591-021-01614-0.

[Re20] Reddy, S.; Allan, S.; Coghlan, S.; Cooper, P.: A governance model for the application of AI in health care. en, Journal of the American Medical Informatics Association 27 (3), pp. 491–497, 2020, ISSN: 1527-974X, DOI: 10.1093/jamia/ocz192.

[Ru19] Rudin, C.: Stop explaining black box machine learning models for high stakes decisions and use interpretable models instead. en, Nature Machine Intelligence 1 (5), pp. 206–215, 2019, ISSN: 2522-5839, DOI: 10.1038/s42256-019-0048-x.

[SB21] Sagadeeva, S.; Boehm, M.: SliceLine: Fast, Linear-Algebra-based Slice Finding for ML Model Debugging. In: SIGMOD. Pp. 2290–2299, 2021, DOI: 10.1145/3448016.3457323.

[Sc20] Scholbeck, C.; Molnar, C.; Heumann, C.; Bischl, B.; Casalicchio, G.: Sampling, Intervention, Prediction, Aggregation: A Generalized Framework for Model-Agnostic Interpretations. In. Pp. 205–216, 2020, DOI: 10.1007/978-3-030-43823-4_18.

[Sh19] Shang, Z.; Zgraggen, E.; Buratti, B.; Kossmann, F.; Eichmann, P.; Chung, Y.; Binnig, C.; Upfal, E.; Kraska, T.: Democratizing Data Science through Interactive Curation of ML Pipelines. In: SIGMOD. Pp. 1171–1188, 2019, DOI: 10.1145/3299869.3319863.

[Sh53] Shapley, L. S.: 17. A Value for n-Person Games. In: Contributions to the Theory of Games (AM-28), Volume II. Princeton University Press, pp. 307–318, 1953, DOI: 10.1515/9781400881970-018.

[ŠK10] Štrumbelj, E.; Kononenko, I.: An Efficient Explanation of Individual Classifications using Game Theory. Journal of Machine Learning Research 11 (1), pp. 1–18, 2010, URL: http://jmlr.org/papers/v11/strumbelj10a.html.

[ŠK14] Štrumbelj, E.; Kononenko, I.: Explaining prediction models and individual predictions with feature contributions. en, Knowledge and Information Systems 41 (3), pp. 647–665, 2014, DOI: 10.1007/s10115-013-0679-x.

[SKB23] Siddiqi, S.; Kern, R.; Boehm, M.: SAGA: A Scalable Framework for Optimizing Data Cleaning Pipelines for Machine Learning Applications. Proc. ACM Manag. Data 1 (3), 218:1–218:26, 2023, DOI: 10.1145/3617338.

[SP23] Staudacher, J.; Pollmann, T.: Assessing Antithetic Sampling for Approximating Shapley, Banzhaf, and Owen Values. en, AppliedMath 3 (4), Number: 4 Publisher: Multidisciplinary Digital Publishing Institute, pp. 957–988, 2023, DOI: 10.3390/appliedmath3040049.