Journal of Statistical Software

MMMMMM YYYY, Volume VV, Issue II. *doi: 10.18637/jss.v000.i00*

shapr: Explaining Machine Learning Models with Conditional Shapley Values in R and Python

Martin Jullum
Norwegian Computing Center, Norway

Lars Henry Berge Olsen
University of Oslo, Norway
Norwegian Computing Center, Norway

Jon Lachmann
Indicio Technologies, Sweden

Annabelle Redelmeier
Norwegian Computing Center, Norway

Abstract

This paper introduces the **shapr** R package, a versatile tool for generating Shapley value-based prediction explanations for machine learning and statistical regression models. Moreover, the **shaprpy** Python library brings the core capabilities of **shapr** to the Python ecosystem. Shapley values originate from cooperative game theory in the 1950s, but have over the past few years become a widely used method for quantifying how a model's features/covariates contribute to specific prediction outcomes. The **shapr** package emphasizes *conditional* Shapley value estimates, providing a comprehensive range of approaches for accurately capturing feature dependencies – a crucial aspect for correct model explanation, typically lacking in similar software. In addition to regular tabular data, the **shapr** R package includes specialized functionality for explaining time series forecasts. The package offers a minimal set of user functions with sensible default values for most use cases while providing extensive flexibility for advanced users to fine-tune computations. Additional features include parallelized computations, iterative estimation with convergence detection, and rich visualization tools. **shapr** also extends its functionality to compute *causal* and *asymmetric* Shapley values when causal information is available. Overall, the **shapr** and **shaprpy** packages aim to enhance the interpretability of predictive models within a powerful and user-friendly framework.

Keywords: Explainable artificial intelligence, XAI, interpretable machine learning, IML, prediction explanation, Shapley values, feature dependence.

1. Introduction

Understanding how complex predictive models produce their outcomes is crucial for their practical application, particularly in high-stakes contexts where trust, transparency, and accountability are essential. The inherent trade-off between model complexity and interpretability often leaves simpler, more interpretable models behind, favoring advanced statistical regression and machine learning models such as generalized additive models (with higher-order interactions), support vector machines, (tree-based) boosting and bagging models, neural networks, and others. As a result, the growing demand for understanding how these high-performance models operate has led to a surge of research in the fields of eXplainable AI (XAI) and Interpretable Machine Learning (IML). During the past few years, the Shapley value framework has established itself as the most prominent framework in this domain.

The Shapley value (Shapley 1953) originates from cooperative game theory, where it is used to distribute the payoff of a cooperative game to the players based on their contribution. In the context of XAI/IML, the framework is extensively used as a *model-agnostic local* explanation framework to explain a prediction $f(\boldsymbol{x}^*)$ from a predictive model $f(\cdot)$. *Model-agnostic* means it can explain any predictive model f, and local means it explains the prediction of a single, specific set of feature (covariate) values $\boldsymbol{x}^*$. The Shapley value of feature j is given by the formula

$$\phi_j = \sum_{\mathcal{S}\subseteq\mathcal{M}\setminus\{j\}} \frac{|\mathcal{S}|!(M-|\mathcal{S}|-1)!}{M!}\left(v(\mathcal{S}\cup\{j\})-v(\mathcal{S})\right), \tag{1}$$

where $\mathcal{M}=\{1,2,\ldots,M\}$ is the set of the M features, and $v(\mathcal{S})$ is the so-called characteristic/value/contribution function, which is some function representing the prediction with only the features in subset/coalition $\mathcal{S}$ present in the model[1]. That is, the Shapley value measures how much a feature contributes to the prediction, averaged over all possible combinations of whether the other feature values are known or not. In general, the Shapley values $\boldsymbol{\phi}=(\phi_0,\phi_1,\ldots,\phi_M)$ satisfies a series of beneficial properties such as summing to $v(\mathcal{S})-\phi_0$, where $\phi_0=v(\emptyset)$, and ϕ_j can be roughly interpreted as the increase or decrease in the prediction caused by the knowledge of $x_j=x_j^*$, see e.g., Shapley (1953); Aas, Jullum, and Løland (2021) for details.

Using the Shapley value framework to explain model predictions was first proposed by Štrumbelj and Kononenko (2010). However, the method did not gain widespread recognition until the publication of Lundberg and Lee (2017), who defined

$$v(\mathcal{S})=\mathbb{E}_{\boldsymbol{x}_{\bar{\mathcal{S}}}}\left[f(\boldsymbol{x})|\boldsymbol{x}_{\mathcal{S}}=\boldsymbol{x}_{\mathcal{S}}^*\right]=\mathbb{E}_{\boldsymbol{x}_{\bar{\mathcal{S}}}}\left[f(\boldsymbol{x}_{\bar{\mathcal{S}}},\boldsymbol{x}_{\mathcal{S}})|\boldsymbol{x}_{\mathcal{S}}=\boldsymbol{x}_{\mathcal{S}}^*\right]=\int f(\boldsymbol{x}_{\bar{\mathcal{S}}},\boldsymbol{x}_{\mathcal{S}}^*)p(\boldsymbol{x}_{\bar{\mathcal{S}}}|\boldsymbol{x}_{\mathcal{S}}=\boldsymbol{x}_{\mathcal{S}}^*)\,\mathrm{d}\boldsymbol{x}_{\bar{\mathcal{S}}}. \tag{2}$$

Here $\boldsymbol{x}_{\mathcal{S}}=\{x_j : j\in\mathcal{S}\}$ denotes the features in subset/coalition $\mathcal{S}$, $\boldsymbol{x}_{\bar{\mathcal{S}}}=\{x_j : j\in\bar{\mathcal{S}}\}$ denotes the features not in $\mathcal{S}$ (i.e., $\bar{\mathcal{S}}=\mathcal{M}\setminus\mathcal{S}$), $\mathbb{E}_{\boldsymbol{x}_{\bar{\mathcal{S}}}}$ denotes the expectation over $\boldsymbol{x}_{\bar{\mathcal{S}}}$, and $p(\boldsymbol{x}_{\bar{\mathcal{S}}}|\boldsymbol{x}_{\mathcal{S}}=\boldsymbol{x}_{\mathcal{S}}^*)$ is the conditional density of $\boldsymbol{x}_{\bar{\mathcal{S}}}$ given $\boldsymbol{x}_{\mathcal{S}}=\boldsymbol{x}_{\mathcal{S}}^*$.

Although Lundberg and Lee (2017) defined $v(\mathcal{S})$ as a conditional expectation, their suggested

[1] Although $v(\mathcal{S})$ also depends on the specific observation $\boldsymbol{x}^*$ being explained, we omit this dependence for notational convenience.

estimation method actually estimates the following contribution function:

$$v_{\text{marg}}(\mathcal{S}) = \mathbb{E}_{\boldsymbol{x}_{\bar{\mathcal{S}}}}\left[f(\boldsymbol{x}_{\bar{\mathcal{S}}}, \boldsymbol{x}^*_{\mathcal{S}})\right] = \int f(\boldsymbol{x}_{\bar{\mathcal{S}}}, \boldsymbol{x}^*_{\mathcal{S}}) p(\boldsymbol{x}_{\bar{\mathcal{S}}}) \, \mathrm{d}\boldsymbol{x}_{\bar{\mathcal{S}}}. \tag{3}$$

That is, the contribution function in (3) implicitly ignores the dependence between the features in $\boldsymbol{x}_{\mathcal{S}}$ and $\boldsymbol{x}_{\bar{\mathcal{S}}}$. This has significant consequences for the properties of the obtained Shapley values and is known to provide misleading conclusions in the presence of highly dependent features. This arises in part because (3) involves model evaluations at implausible feature combinations not seen during training, where the model's output holds little practical relevance. Aas *et al.* (2021) were the first to identify this issue and provide methods for properly providing Shapley values with the conditional expectation in (2). Despite the drawbacks, the benefits of significantly simpler computation and the availability of easy-to-use software (**shap**, Lundberg and Lee (2024), in Python), have led to (3) still being very much in use (Chen, Janizek, Lundberg, and Lee 2020). Shapley values computed with (3) are now often referred to as *marginal* Shapley values, while those computed using (2) are referred to as *conditional* Shapley values.

As mentioned above, explaining predictions with Shapley values helps quantify how valuable the observation of each feature is to a specific prediction. *Conditional* Shapley values do this while respecting the distribution of the features, leading to explanations that are more relevant and realistic in practical use. As a result, using conditional Shapley values to explain predictions enables practitioners to extract relevant knowledge about the local behavior of black-box models.

The growing popularity of the Shapley value framework for prediction explanation has led to the emergence of numerous software packages in recent years. The most popular software is the **shap** (Lundberg and Lee 2024) library in Python, which implements a range of methods for computing/estimating Shapley values for different models and data types, including the model-agnostic KernelSHAP (Lundberg and Lee 2017) and permutation-based formulation of Štrumbelj and Kononenko (2010, 2014) (hereafter referred to as PermSHAP), and model-specific methods (TreeSHAP (Lundberg, Erion, Chen, DeGrave, Prutkin, Nair, Katz, Himmelfarb, Bansal, and Lee 2020) for tree-based models, and DeepLIFT for approximated Shapley values (Shrikumar, Greenside, and Kundaje 2017) for neural networks). Building on the `PyTorch` framework, the **captum** Python library (Kokhlikyan, Miglani, Martin, Wang, Alsallakh, Reynolds, Melnikov, Kliushkina, Araya, Yan, and Reblitz-Richardson 2020) implements several explanation methods, including KernelSHAP and PermSHAP, as well as model-specific estimators. The recent **shapiq** (Muschalik, Baniecki, Fumagalli, Kolpaczki, Hammer, and Hüllermeier 2024) Python library implements a wide range of methods for computing Shapley values for different types of games, including KernelSHAP and PermSHAP.

The *DrWhy* universe is a collection of R packages for creating explanations and visual explorations of predictive models. It contains the **DALEX** (Biecek 2018) and **modelStudio** (Baniecki and Biecek 2019) libraries, which work as high-level explanation tools where Shapley value-based prediction explanations are one of several ingredients. The actual Shapley value computations happen through sister packages: **shapper** (Maksymiuk, Gosiewska, and Biecek 2020) is just a wrapper for the KernelSHAP implementation in the **shap** Python library (Lundberg and Lee 2024), **treeshap** (Komisarczyk, Kozminski, Maksymiuk, and Biecek 2024) implements variants of the model-specific TreeSHAP algorithm (Lundberg *et al.* 2020), **kernelshap** (Mayer and Watson 2024) allows computing Shapley values through either KernelSHAP or PermSHAP, **fastshap** (Greenwell 2024) also offers a fast implementation of the

latter, while **iBreakDown** (Gosiewska and Biecek 2019) uses so-called "Break Down" Tables (Biecek and Burzykowski 2021, Ch. 6) to approximate the Shapley values. The associated **shapviz** (Mayer 2024) package provides Shapley value visualizations. Outside of the *DrWhy* universe, the **iml** package (Molnar, Bischl, and Casalicchio 2018) provides several interpretation/explanation methods. The package includes a method for deriving Shapley values for individual predictions based on PermSHAP.

What is common for almost all of the above software packages is that they exclusively estimate, compute, or approximate *marginal* Shapley values. While the so-called path-dependent variant of the TreeSHAP algorithm (Lundberg *et al.* 2020) aims at estimating *conditional* Shapley values, it is often (severely) biased in practice, see e.g., Aas *et al.* (2021, Sec. 4) and Chen, Covert, Lundberg, and Lee (2023, Sec. 5.2). The only library which touches upon *conditional* Shapley values is the **shapiq** package, which, according to Muschalik *et al.* (2024, Appendix C), offers a simple tree-based regression method for estimating the $v(\mathcal{S})$ in (2).

The **shapr** R package introduced in this paper implements an extended version of the KernelSHAP method (Olsen and Jullum 2025) for approximating Shapley values, heavily focused on *conditional* Shapley values. The core idea of the package is to be completely model-agnostic and offer a wide range of approaches for estimating $v(\mathcal{S})$ in (2) (see Section 2.3), allowing accurately estimated *conditional* Shapley values to be computed for different types of features, dependencies, and distributions. Evaluation metrics for comparing different approaches are also readily available within the package. Combined with parallelized computations, convergence detection, progress updates, and extensive plotting functionality, the **shapr** package offers an efficient and user-friendly solution for estimating *conditional* Shapley values. These accurate estimates are essential for an increased understanding of how features *actually* contribute to model predictions in practice. To increase the accessibility of the methodology, the **shapr** R package also comes with an accompanying Python wrapper called **shaprpy**. The wrapper makes it possible to explain models available in Python with the same estimation approaches and interface as the R package.

The present paper is based on **shapr** version 1.0.6 and **shaprpy** version 0.4.0. Note that Sellereite and Jullum (2019) briefly describes version 0.1.1 of **shapr**, which had significantly less functionality and used a different syntax compared to the current version[2].

The rest of the paper is organized as follows: Section 2 provides descriptions of the main methodology implemented in **shapr**. In particular, it briefly describes all approaches used to estimate the $v(\mathcal{S})$ in (2). Section 3 introduces the **shapr** package and its main functionality, and provides basic usage examples for estimating conditional Shapley value explanations. In Section 4, we introduce asymmetric and causal Shapley values, and show how such types of Shapley values can be computed with **shapr**, when causal information is available. The associated **shaprpy** Python wrapper is introduced in Section 5. Section 6 describes functionality for explaining time series models with multiple forecasting horizons. In Section 7, we provide a summary and discuss potential future work.

2. Methodological background

Computing conditional Shapley values for prediction explanations involves two key steps:

[2]Version 1.0.0 of **shapr** represented a complete rewrite of the package, adding the majority of the functionality and flexibility in the current version, and enabling the creation of the **shaprpy** Python wrapper.

Obtaining accurate estimates of $v(\mathcal{S})$ in (2), and computing the Shapley values based on these estimates. Below, we briefly introduce the methodology implemented in the **shapr** package to address both steps.

2.1. KernelSHAP

The Shapley value formula in (1) can be computationally intensive in high-dimensional settings, as its complexity grows exponentially with the number of features. Charnes, Golany, Keane, and Rousseau (1988); Lundberg and Lee (2017) showed that the Shapley value formula in (1) may also be conveniently expressed as the solution of the following weighted least squares (WLS) problem:

$$\underset{\boldsymbol{\phi}\in\mathbb{R}^{M+1}}{\arg\min} \sum_{\mathcal{S}\subseteq\mathcal{M}} k(M,|\mathcal{S}|)\Big(\phi_0 + \sum_{j\in\mathcal{S}}\phi_j - v(\mathcal{S})\Big)^2, \tag{4}$$

where

$$k(M,|\mathcal{S}|) = \frac{M-1}{\binom{M}{|\mathcal{S}|}|\mathcal{S}|(M-|\mathcal{S}|)}, \tag{5}$$

for $|\mathcal{S}| = 0, 1, 2, \ldots, M$, are the *Shapley kernel weights* (Charnes *et al.* 1988; Lundberg and Lee 2017). In practice, the infinite Shapley kernel weights $k(M,0) = k(M,M) = \infty$ can be set to a large constant like $C = 10^6$ (Aas *et al.* 2021). The matrix solution of (4) is

$$\boldsymbol{\phi} = (\boldsymbol{Z}^T\boldsymbol{W}\boldsymbol{Z})^{-1}\boldsymbol{Z}^T\boldsymbol{W}\boldsymbol{v} = \boldsymbol{R}\boldsymbol{v}. \tag{6}$$

Here $\boldsymbol{Z}$ is a $2^M \times (M+1)$ matrix with 1s in the first column (to obtain ϕ_0) and the binary representations[3] of the coalitions $\mathcal{S} \subseteq \mathcal{M}$ in the remaining columns. $\boldsymbol{W} = \text{diag}(C, \boldsymbol{w}, C)$ is a $2^M \times 2^M$ diagonal matrix containing the Shapley kernel weights $k(M,|\mathcal{S}|)$ on the diagonal. The $\boldsymbol{w}$ vector contains the $2^M - 2$ finite Shapley kernel weights from (5). Finally, $\boldsymbol{v}$ is a column vector of size 2^M containing the contribution function values $v(\mathcal{S})$. The $\mathcal{S}$ in $\boldsymbol{W}$ and $\boldsymbol{v}$ corresponds to the coalition of the corresponding row in $\boldsymbol{Z}$. The $\boldsymbol{R}$ matrix is independent of the explicands. When explaining N_{explain} predictions (explicands), we can replace $\boldsymbol{v}$ with a $2^M \times N_{\text{explain}}$ matrix $\boldsymbol{V}$, where column i contains the contribution functions for the ith explicand.

Computing either (1) or (4) is infeasible in higher dimensions as the number of coalitions 2^M grows exponentially with the number of features M. A common solution is to approximate the Shapley values by solving (4) using a randomly sampled set $\mathcal{D}$ of coalitions $\mathcal{S} \subseteq \mathcal{M}$ (with replacement), instead of all the 2^M coalitions (Lundberg and Lee 2017). The coalitions are sampled from a distribution proportional to the Shapley kernel weights in (5), while the empty and grand coalitions are always included and exempt from the sampling. The corresponding Shapley value approximation is

$$\boldsymbol{\phi}_{\mathcal{D}} = (\boldsymbol{Z}_{\mathcal{D}}^T\boldsymbol{W}_{\mathcal{D}}\boldsymbol{Z}_{\mathcal{D}})^{-1}\boldsymbol{Z}_{\mathcal{D}}^T\boldsymbol{W}_{\mathcal{D}}\boldsymbol{v}_{\mathcal{D}} = \boldsymbol{R}_{\mathcal{D}}\boldsymbol{v}_{\mathcal{D}} \tag{7}$$

where only the $N_{\text{coal}} = |\mathcal{D}|$ unique coalitions in $\mathcal{D}$ are used. If a coalition $\mathcal{S}$ is sampled K times, then the corresponding weight in $\boldsymbol{W}_{\mathcal{D}} = \text{diag}(C, \boldsymbol{w}_{\mathcal{D}}, C)$, is set to its sampling frequency,

[3]For example, the binary representation of $\mathcal{S} = \{1,3\}$ in an $M = 4$-dimensional setting is $[1,0,1,0]$.

i.e., $w_{\mathcal{S}} = K$. In practice, the $w_{\mathcal{S}}$ values are often normalized for numerical stability. We refer to both (6) and the approximate solution in (7) as KernelSHAP (Lundberg and Lee 2017). Williamson and Feng (2020) shows that the KernelSHAP approximation framework is consistent and asymptotically unbiased, while Covert and Lee (2021) demonstrates that it is empirically unbiased for even a modest number of coalitions.

Antithetic or *paired* sampling is a simple variance reduction technique for Monte Carlo sampling (Kroese, Taimre, and Botev 2013, Ch. 9). In the KernelSHAP sampling setting, this has been utilized by always including $v(\bar{\mathcal{S}})$ whenever $v(\mathcal{S})$ is included in the computation, to significantly reduce the variance of the Shapley value estimates from (7) (Covert and Lee 2021; Mitchell, Cooper, Frank, and Holmes 2022). Pairing coalitions stabilizes the sampling frequencies by ensuring that $\mathcal{S}$ and $\bar{\mathcal{S}}$ consistently receive the same weight $w_{\mathcal{S}}$ in (7). However, due to randomness in the sampling procedure, the observed coalition weights may deviate from their nominal sampling probabilities – for example, coalitions of the same size can easily end up with different weights.

To address this and further reduce the variance of the KernelSHAP estimator, Olsen and Jullum (2025) suggest to use reweighing strategies. The authors propose a simple correction to the frequency based WLS weights $w_{\mathcal{S}}$ used in (7). The corrected weights are calculated by normalizing the Shapley kernel weights in (5) and conditioning on the coalition being sampled at least once, i.e.,

$$\tilde{p}_{\mathcal{S}} = \frac{p_{\mathcal{S}}}{\Pr(\mathcal{S} \text{ sampled at least once})} = \frac{p_{\mathcal{S}}}{1 - \Pr(\mathcal{S} \text{ not sampled})} = \frac{p_{\mathcal{S}}}{1 - (1 - p_{\mathcal{S}})^L}, \tag{8}$$

where $p_{\mathcal{S}} = k(M, |\mathcal{S}|) / \left(\sum_{q=1}^{M-1} k(M, q) \binom{M}{q} \right)$ is the normalized version of the Shapley kernel weights and L is the total number of sampled coalitions in $\mathcal{D}$ (including duplicates). Thus, for a sampled coalition $\mathcal{S}$ the corrected weight $w_{\mathcal{S}}$ used in (7) is

$$w_{\mathcal{S}} \propto \frac{p_{\mathcal{S}}}{1 - (1 - p_{\mathcal{S}})^L}. \tag{9}$$

Moreover, the authors describe a *semi-deterministic sampling* scheme implemented in the **shap** Python library (Lundberg and Lee 2024). The procedure essentially includes all coalitions of size k when a fully random sample of size N_{coal} would be expected to cover them, and samples the remaining coalitions randomly with probabilities proportional to their Shapley kernel weights in (5).

Extensive numerical experiments by Olsen and Jullum (2025) demonstrate that combining *paired sampling* with the corrected Shapley kernel weights, both with and without the semi-deterministic sampling component, are superior as they match the accuracy of the standard KernelSHAP approximator using only 5% to 50% of the coalitions.

2.2. Estimation paradigms

The model-agnostic approaches used to estimate the conditional contribution function in (2) can be categorized into two main paradigms: The *Monte Carlo* and *regression* paradigms.

In the Monte Carlo paradigm, Monte Carlo integration is used to estimate $v(\mathcal{S})$:

$$v(\mathcal{S}) = \mathbb{E}_{\boldsymbol{x}_{\bar{\mathcal{S}}}} \left[f(\boldsymbol{x}_{\bar{\mathcal{S}}}, \boldsymbol{x}_{\mathcal{S}}) | \boldsymbol{x}_{\mathcal{S}} = \boldsymbol{x}_{\mathcal{S}}^* \right] \approx \frac{1}{K} \sum_{k=1}^{K} f(\boldsymbol{x}_{\bar{\mathcal{S}}}^{(k)}, \boldsymbol{x}_{\mathcal{S}}^*) = \hat{v}(\mathcal{S}), \tag{10}$$

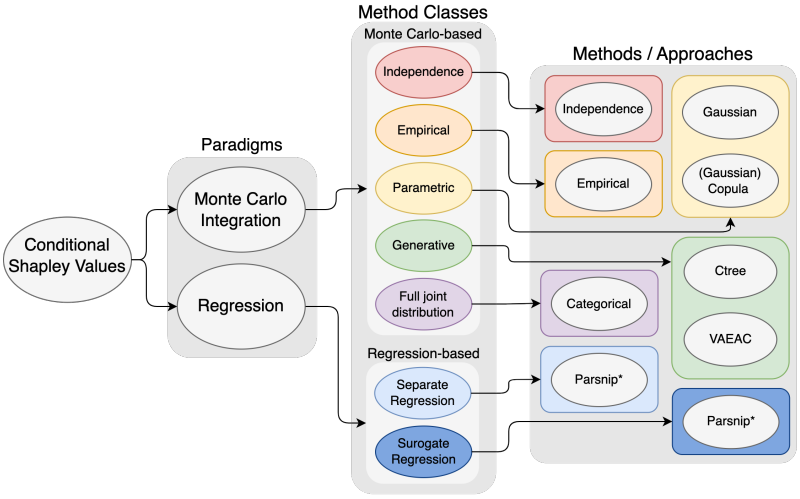


Figure 1: Schematic overview of the available approaches in **shapr** for computing conditional Shapley value explanations. *Parsnip** indicates that the regression-based approaches can use any regression methods available in the **parsnip** package (Kuhn and Vaughan 2024), including user-specified methods. The **shapr** package also implements the `timeseries` approach, which is specifically designed for time series data and not applicable to regular tabular datasets. See Jullum *et al.* (2021, Sec. 3.4) for details.

where $\boldsymbol{x}_{\bar{\mathcal{S}}}^{(k)} \sim p(\boldsymbol{x}_{\bar{\mathcal{S}}}|\boldsymbol{x}_{\mathcal{S}} = \boldsymbol{x}_{\mathcal{S}}^*)$, for $k = 1, 2, \ldots, K$, and K is the number of Monte Carlo samples. The estimated $v(\mathcal{S})$ can then be inserted into (1), (6), or (7) to approximate the Shapley values. To obtain accurate conditional Shapley values, we need to generate Monte Carlo samples that follow the conditional distribution of the data. This distribution is generally not known and needs to be estimated based on the training data. Approaches for estimating those are discussed in Section 2.3.

The regression paradigm uses the fact that the conditional expectation is the minimizer of the mean squared error loss function:

$$v(\mathcal{S}) = \mathbb{E}_{\boldsymbol{x}_{\bar{\mathcal{S}}}}\left[f(\boldsymbol{x}_{\bar{\mathcal{S}}}, \boldsymbol{x}_{\mathcal{S}})|\boldsymbol{x}_{\mathcal{S}} = \boldsymbol{x}_{\mathcal{S}}^*\right] = \arg\min_c \mathbb{E}_{\boldsymbol{x}_{\bar{\mathcal{S}}}}\left[(f(\boldsymbol{x}_{\bar{\mathcal{S}}}, \boldsymbol{x}_{\mathcal{S}}) - c)^2|\boldsymbol{x}_{\mathcal{S}} = \boldsymbol{x}_{\mathcal{S}}^*\right]. \tag{11}$$

Thus, any regression model $g_{\mathcal{S}}(\boldsymbol{x}_{\mathcal{S}})$, which is fitted with the mean squared error loss function as the objective function will approximate (11), obtaining an alternative estimator $\hat{v}(\mathcal{S})$ of $v(\mathcal{S})$ (Frye, de Mijolla, Begley, Cowton, Stanley, and Feige 2021; Williamson and Feng 2020; Olsen, Glad, Jullum, and Aas 2024). The accuracy of the approximation will depend on the form of the predictive model $f(\boldsymbol{x})$, the flexibility of the regression model $g_{\mathcal{S}}(\boldsymbol{x}_{\mathcal{S}})$, and the optimization routine. We can either train a separate regression model $g_{\mathcal{S}}(\boldsymbol{x}_{\mathcal{S}})$ for each $\mathcal{S}$ or we can train a single regression model $g(\tilde{\boldsymbol{x}}_{\mathcal{S}})$ which approximates the contribution function $v(\mathcal{S})$ for all $\mathcal{S}$ simultaneously. Here $\tilde{\boldsymbol{x}}_{\mathcal{S}}$ is an augmented version of $\boldsymbol{x}_{\mathcal{S}}$ with fixed-length representation (see *Regression Surrogate* in Section 2.3).

2.3. Estimation approaches

Below we give brief introductions to a wide range of approaches for estimating $v(\mathcal{S})$ with both paradigms described in Section 2.2, all of which are implemented in the **shapr** package. Unless

"Regression" is explicitly mentioned in the approach name, the approaches described below follow the Monte Carlo paradigm. Figure 1 provides a schematic overview of the approaches.

In addition to the model-agnostic approaches described below, **shapr** also includes the `timeseries` approach. This is specifically designed for time series data and not suitable for standard tabular datasets. For more details of that special case, we refer to Jullum *et al.* (2021, Sec. 3.4).

Independence

Lundberg and Lee (2017) avoided estimating the complex conditional distributions by implicitly assuming feature independence. While this often leads to incorrect conditional Shapley value explanations for real-world data (Aas *et al.* 2021; Frye *et al.* 2021) (and is therefore not recommended) the method is included for reference. In the `independence` approach, the conditional distribution $p(\boldsymbol{x}_{\bar{\mathcal{S}}}|\boldsymbol{x}_{\mathcal{S}})$ simplifies to $p(\boldsymbol{x}_{\bar{\mathcal{S}}})$, and the corresponding Shapley values are the marginal Shapley values obtained by using the $v_{\text{marg}}(\mathcal{S})$ in (3). The Monte Carlo samples $\boldsymbol{x}_{\bar{\mathcal{S}}}^{(k)} \sim p(\boldsymbol{x}_{\bar{\mathcal{S}}})$ in (10) are generated by randomly sampling observations from the training data; thus, no modeling is needed. For this reason, this is usually a very fast approach.

Empirical

A natural way to account for feature dependence when sampling from the training data is to sample observations similar to $\boldsymbol{x}_{\mathcal{S}}^*$, rather than sampling entirely at random as done in the `independence` approach. This is the idea behind the `empirical` method described in Aas *et al.* (2021).

A scaled version of the Mahalanobis distance $D_{\mathcal{S}}$ is used to calculate a distance $D_{\mathcal{S}}(\boldsymbol{x}^*, \boldsymbol{x}^{[i]})$ between the explicand $\boldsymbol{x}^*$ and every training instance $\boldsymbol{x}^{[i]}$, for the features in $\mathcal{S}$. Then, a Gaussian distribution kernel is used to convert the distance into a weight $w_{\mathcal{S}}(\boldsymbol{x}^*, \boldsymbol{x}^{[i]})$. All the weights are sorted in increasing order with $\boldsymbol{x}^{\{k\}}$ having the kth largest value. Finally, (2) is approximated by a weighted version of (10):

$$\hat{v}(\mathcal{S}) = \sum_{k=1}^{K^*} \frac{w_{\mathcal{S}}(\boldsymbol{x}^*, \boldsymbol{x}^{\{k\}}) f(\boldsymbol{x}_{\bar{\mathcal{S}}}^{\{k\}}, \boldsymbol{x}_{\mathcal{S}}^*)}{\sum_{k=1}^{K^*} w_{\mathcal{S}}(\boldsymbol{x}^*, \boldsymbol{x}^{\{k\}})}. \tag{12}$$

The K^* here is typically set such that, for instance, $\eta = 95\%$ of the total weight across the training set is accounted for: $K^* = \min_{L \in \mathbb{N}} \{ \sum_{k=1}^{L} w_{\mathcal{S}}(\boldsymbol{x}^*, \boldsymbol{x}^{\{k\}}) / \sum_{i=1}^{N_{\text{train}}} w_{\mathcal{S}}(\boldsymbol{x}^*, \boldsymbol{x}^{[i]}) > \eta \}$.

Gaussian

For well-behaved unimodal data, a natural approach is to assume that $\boldsymbol{x}$ stems from a multivariate Gaussian distribution with mean $\boldsymbol{\mu}$ and covariance matrix $\boldsymbol{\Sigma}$. In the present context, this method was first suggested by Aas *et al.* (2021).

A classic and beneficial property of this assumption is that the conditional distributions are also multivariate Gaussian. That is, if $p(\boldsymbol{x}) = p(\boldsymbol{x}_{\mathcal{S}}, \boldsymbol{x}_{\bar{\mathcal{S}}}) = \mathcal{N}_M(\boldsymbol{\mu}, \boldsymbol{\Sigma})$, where $\boldsymbol{\mu} = [\boldsymbol{\mu}_{\mathcal{S}}, \boldsymbol{\mu}_{\bar{\mathcal{S}}}]^T$ and $\boldsymbol{\Sigma} = \begin{bmatrix} \boldsymbol{\Sigma}_{\mathcal{S}\mathcal{S}} & \boldsymbol{\Sigma}_{\mathcal{S}\bar{\mathcal{S}}} \\ \boldsymbol{\Sigma}_{\bar{\mathcal{S}}\mathcal{S}} & \boldsymbol{\Sigma}_{\bar{\mathcal{S}}\bar{\mathcal{S}}} \end{bmatrix}$, then, $p(\boldsymbol{x}_{\bar{\mathcal{S}}}|\boldsymbol{x}_{\mathcal{S}} = \boldsymbol{x}_{\mathcal{S}}^*) = \mathcal{N}_{|\bar{\mathcal{S}}|}(\boldsymbol{\mu}_{\bar{\mathcal{S}}|\mathcal{S}}, \boldsymbol{\Sigma}_{\bar{\mathcal{S}}|\mathcal{S}})$, where $\boldsymbol{\mu}_{\bar{\mathcal{S}}|\mathcal{S}} = \boldsymbol{\mu}_{\bar{\mathcal{S}}} + \boldsymbol{\Sigma}_{\bar{\mathcal{S}}\mathcal{S}}\boldsymbol{\Sigma}_{\mathcal{S}\mathcal{S}}^{-1}(\boldsymbol{x}_{\mathcal{S}}^* - \boldsymbol{\mu}_{\mathcal{S}})$ and $\boldsymbol{\Sigma}_{\bar{\mathcal{S}}|\mathcal{S}} = \boldsymbol{\Sigma}_{\bar{\mathcal{S}}\bar{\mathcal{S}}} - \boldsymbol{\Sigma}_{\bar{\mathcal{S}}\mathcal{S}}\boldsymbol{\Sigma}_{\mathcal{S}\mathcal{S}}^{-1}\boldsymbol{\Sigma}_{\mathcal{S}\bar{\mathcal{S}}}$. The parameters $\boldsymbol{\mu}$ and $\boldsymbol{\Sigma}$ are easily estimated using, respectively, the sample mean and covariance matrix of the training

data. Finally, (10) is used to estimate $v(\mathcal{S})$ with samples $\boldsymbol{x}_{\bar{\mathcal{S}}}^{(k)}$ from $p(\boldsymbol{x}_{\bar{\mathcal{S}}}|\boldsymbol{x}_{\mathcal{S}} = \boldsymbol{x}_{\mathcal{S}}^*)$, for $k = 1, 2, \ldots, K$.

Gaussian copula

A generalization of the `gaussian` approach, also proposed by Aas *et al.* (2021), is to use a Gaussian copula. That is, to represent the marginals of the features by their empirical distributions and then model the dependence structure by a Gaussian distribution. This `copula` approach generates the K conditional Monte Carlo samples $\boldsymbol{x}_{\bar{\mathcal{S}}}^{(k)} \sim p(\boldsymbol{x}_{\bar{\mathcal{S}}}|\boldsymbol{x}_{\mathcal{S}} = \boldsymbol{x}_{\mathcal{S}}^*)$ with the following routine:

1. Convert each marginal x_j of the feature distribution $\boldsymbol{x}$ to a Gaussian distributed variable v_j by $v_j = \Phi^{-1}(\hat{F}(x_j))$, where $\hat{F}(x_j)$ is the empirical distribution function of x_j.
2. Assume that $\boldsymbol{v}$ is distributed according to a multivariate Gaussian, and sample from the conditional distribution $p(\boldsymbol{v}_{\bar{\mathcal{S}}}|\boldsymbol{v}_{\mathcal{S}} = \boldsymbol{v}_{\mathcal{S}}^*)$ using the method described for the `gaussian` approach.
3. Convert the marginals v_j in the conditional distribution to the original distribution using $\hat{x}_j = \hat{F}_j^{-1}(\Phi(v_j))$.

Ctree

Redelmeier, Jullum, and Aas (2020) compute conditional Shapley values by modeling the dependence structure between the features with conditional inference trees (*ctree*) to then sample from these, i.e., using a generative method. A *ctree* is a type of recursive partitioning algorithm that builds trees recursively by doing binary splits on features until a stopping criterion is satisfied (Hothorn, Hornik, and Zeileis 2006). The process is sequential, where the splitting feature is chosen first using statistical significance tests, and then the splitting point is chosen using a splitting criterion. The *ctree* algorithm is independent of the dimension of the response, which in our case is $\boldsymbol{x}_{\bar{\mathcal{S}}}$. The input features are $\boldsymbol{x}_{\mathcal{S}}$, which varies in dimension based on the coalition $\mathcal{S}$. That is, for each coalition $\mathcal{S}$, a *ctree* with $\boldsymbol{x}_{\mathcal{S}}$ as the features and $\boldsymbol{x}_{\bar{\mathcal{S}}}$ as the response is fitted to the training data. For a given $\boldsymbol{x}_{\mathcal{S}}^*$, the `ctree` approach finds the corresponding leaf node and samples K observations with replacement from the $\boldsymbol{x}_{\bar{\mathcal{S}}}$ part of the training observations in the same node to generate the conditional Monte Carlo samples $\boldsymbol{x}_{\bar{\mathcal{S}}}^{(k)} \sim p(\boldsymbol{x}_{\bar{\mathcal{S}}}|\boldsymbol{x}_{\mathcal{S}} = \boldsymbol{x}_{\mathcal{S}}^*)$. Whenever K is larger than the number of samples in the leaf node, the `ctree` approach uses all unique samples in the leaf node. It computes the $v(\mathcal{S})$ as a weighted mean similar to (12) instead of by (10).

VAEAC

The `vaeac` approach is another generative method for estimating $v(\mathcal{S})$. Olsen, Glad, Jullum, and Aas (2022) use a type of variational autoencoder called *vaeac* (Ivanov, Figurnov, and Vetrov 2019) to generate the conditional Monte Carlo samples. Briefly stated, the original variational autoencoder (Kingma and Welling 2014, 2019; Rezende, Mohamed, and Wierstra 2014) gives a probabilistic representation of the true unknown distribution $p(\boldsymbol{x})$. The *vaeac* model extends this methodology to all conditional distributions $p(\boldsymbol{x}_{\bar{\mathcal{S}}}|\boldsymbol{x}_{\mathcal{S}} = \boldsymbol{x}_{\mathcal{S}}^*)$ simultaneously. That is, a single *vaeac* model can generate Monte Carlo samples $\boldsymbol{x}_{\bar{\mathcal{S}}}^{(k)} \sim p(\boldsymbol{x}_{\bar{\mathcal{S}}}|\boldsymbol{x}_{\mathcal{S}} = \boldsymbol{x}_{\mathcal{S}}^*)$

for all coalitions $\mathcal{S} \subseteq \mathcal{M}$. It is advantageous to only have to fit a single model for all coalitions, as the number of coalitions increases exponentially with the number of features. In contrast, `ctree` trains $2^M - 2$ different models, eventually becoming computationally intractable for large M. The `vaeac` approach trains a *vaeac* model by maximizing a variational lower bound, which conceptually corresponds to artificially masking features, and then trying to reproduce them using a probabilistic representation. In deployment, it considers the unconditional features $\boldsymbol{x}_{\bar{\mathcal{S}}}$ as masked features to be imputed.

Categorical

When the features are all categorical, we can estimate the conditional expectations using basic statistical theory. That is, all marginal and joint probabilities are estimated by simply counting the frequencies of each feature, feature pair, feature triplets, etc. Then all conditional probabilities can be computed by dividing the full joint distributions by the necessary lower-order joint distributions. Since this provides a complete tabular description of all conditional distributions $p(\boldsymbol{x}_{\bar{\mathcal{S}}}|\boldsymbol{x}_{\mathcal{S}})$, we can easily *compute* the $v(\mathcal{S})$ through

$$v(\mathcal{S}) = \mathbb{E}_{\boldsymbol{x}_{\bar{\mathcal{S}}}}[f(\boldsymbol{x})|\boldsymbol{x}_{\mathcal{S}} = \boldsymbol{x}^*_{\mathcal{S}}] = \sum_{\boldsymbol{z} \in \mathcal{Z}} f(\boldsymbol{x}^*_{\mathcal{S}}, \boldsymbol{z}) p(\boldsymbol{x}_{\bar{\mathcal{S}}} = z|\boldsymbol{x}_{\mathcal{S}} = \boldsymbol{x}^*_{\mathcal{S}}), \tag{13}$$

where $\mathcal{Z}$ is the finite sample space of $\boldsymbol{x}_{\bar{\mathcal{S}}}|\boldsymbol{x}_{\mathcal{S}}$. Even though this is an exact computation of the expectation, given the estimated marginal and joint distributions, we group the method together with the Monte Carlo approaches since the computation of the expectations takes the same form. For computational reasons, the approach is most relevant when the cardinality (number of factor levels) of the features is not too large.

Regression separate

Turning to the regression paradigm, the `regression_separate` approach estimates $v(\mathcal{S}) = \mathbb{E}_{\boldsymbol{x}_{\bar{\mathcal{S}}}}[f(\boldsymbol{x})|\boldsymbol{x}_{\mathcal{S}} = \boldsymbol{x}^*_{\mathcal{S}}]$ separately for each coalition $\mathcal{S}$ using regression (Olsen *et al.* 2024). Let $\boldsymbol{x}^{[i]}$ denote the ith M-dimensional input and $y^{[i]}$ the associated response of the training data. For each $\mathcal{S}$, define the coalition dataset

$$\mathcal{X}_{\mathcal{S}} = \{\boldsymbol{x}^{[i]}_{\mathcal{S}}, f(\underbrace{\boldsymbol{x}^{[i]}_{\bar{\mathcal{S}}}, \boldsymbol{x}^{[i]}_{\mathcal{S}}}_{\boldsymbol{x}^{[i]}})\}_{i=1}^{N_{\text{train}}} = \{\boldsymbol{x}^{[i]}_{\mathcal{S}}, \underbrace{f(\boldsymbol{x}^{[i]})}_{z^{[i]}}\}_{i=1}^{N_{\text{train}}} = \{\boldsymbol{x}^{[i]}_{\mathcal{S}}, z^{[i]}\}_{i=1}^{N_{\text{train}}}.$$

For each $\mathcal{X}_{\mathcal{S}}$, we train a regression model $g_{\mathcal{S}}(\boldsymbol{x}_{\mathcal{S}})$ with the mean squared error loss function, which has the property of aiming at estimating precisely $\mathbb{E}_{\boldsymbol{x}_{\bar{\mathcal{S}}}}[z|\boldsymbol{x}_{\mathcal{S}}] = \mathbb{E}_{\boldsymbol{x}_{\bar{\mathcal{S}}}}[f(\boldsymbol{x}_{\bar{\mathcal{S}}}, \boldsymbol{x}_{\mathcal{S}})|\boldsymbol{x}_{\mathcal{S}}]$. To get estimates of $v(\mathcal{S})$ for a specific explicand $\boldsymbol{x}^*$, the estimated regression model $g_{\mathcal{S}}(\boldsymbol{x}_{\mathcal{S}})$ is simply evaluated at $\boldsymbol{x}^*_{\mathcal{S}}$. See, e.g., Olsen *et al.* (2024, Sec. 3.5) for more details.

Regression surrogate

A drawback of the `separate_regression` approach is that training a separate regression model $g_{\mathcal{S}}(\boldsymbol{x}_{\mathcal{S}})$ for each coalition $\mathcal{S}$ becomes infeasible as the number of features M increases. Additionally, the separate training of each $g_{\mathcal{S}}$ from scratch prevents them from borrowing strength from each other, even though the regression problems may be similar for similar coalitions. The `surrogate_regression` approach addresses these limitations by training a single regression model g on an augmented dataset that handles all coalitions simultaneously.

The surrogate model g must account for the varying size of $\boldsymbol{x}_\mathcal{S}$ for each coalition $\mathcal{S}$. Olsen *et al.* (2024, Sec. 3.6) solve this problem by creating a fixed-length representation $\tilde{\boldsymbol{x}}_\mathcal{S}$ of $\boldsymbol{x}_\mathcal{S}$ for all coalitions $\mathcal{S}$. The $\tilde{\boldsymbol{x}}_\mathcal{S}$ representation is of length $2M$, with the two halves being the fixed-length representations of $\boldsymbol{x}_\mathcal{S}$ and $\mathcal{S}$, respectively. The latter is included to enable the regression model g to distinguish between different coalitions. The training dataset is augmented by systematically applying these fixed-length representations to each training observation. The surrogate regression model g is then fitted to this augmented dataset using the mean squared loss function, as done for the `separate_regression` approach. This enables g to estimate the contribution function $v(\mathcal{S})$ for all coalitions $\mathcal{S} \subseteq \mathcal{M}$ simultaneously. See, e.g., Olsen *et al.* (2024, Sec. 3.6) for more details.

2.4. Iterative KernelSHAP and convergence detection

Since it is computationally intensive to estimate many $v(\mathcal{S})$ with most of the methods described in Section 2.3, we aim to use only as many as needed to achieve sufficiently accurate Shapley value estimates. A simple solution to this is to iteratively add more $v(\mathcal{S})$ values, and stop once the desired accuracy level is reached. This can be done by initially sampling a set of coalitions and then using bootstrapping (Goldwasser and Hooker 2024, Sec. 4.2) to estimate the variance of the approximated Shapley values. A convergence criterion is used to determine if the variances of the Shapley values are sufficiently small. If the variances are too large, we may estimate the number of required samples to reach convergence and then increase the number of coalitions accordingly. The process is repeated until the variances are below the convergence threshold.

Covert and Lee (2021) outline such an algorithm with a convergence criterion for computing the Shapley values of a single explicand $\boldsymbol{x}^*$. We suggest modifying their criterion to work for multiple observations with a simple median across the explicands, leaving us with

$$\underset{i}{\text{median}}\left(\frac{\max_j \text{sd}(\phi_{ij})}{\max_j \phi_{ij} - \min_j \phi_{ij}}\right) < t, \tag{14}$$

where ϕ_{ij} denotes the estimated Shapley value of feature j for explicand i, and $\text{sd}(\phi_{ij})$ is its (bootstrap) estimated standard deviation. The convergence threshold value t may be set to a small number (Covert and Lee (2021) suggest 0.01).

2.5. MSE_v evaluation criterion

To evaluate and rank Shapley values from different approaches in Section 2.3, Frye *et al.* (2021) proposed to use the mean squared error across N_{explain} observations $\boldsymbol{x}^{[1]}, \ldots, \boldsymbol{x}^{[N_{\text{explain}}]}$:

$$\text{MSE}_v(\text{method } \mathtt{q}) = \frac{1}{N_{\text{coal}}} \sum_{\mathcal{S}} \frac{1}{N_{\text{explain}}} \sum_{i=1}^{N_{\text{explain}}} \left(f(\boldsymbol{x}^{[i]}) - \hat{v}_\mathtt{q}(\mathcal{S}, \boldsymbol{x}^{[i]})\right)^2, \tag{15}$$

where N_{coal} is the number of used coalitions and $\hat{v}_\mathtt{q}(\mathcal{S}, \boldsymbol{x}^{[i]})$ denotes the estimated contribution function using method $\mathtt{q}$, evaluated at $\boldsymbol{x}^{[i]}$.

The motivation behind the MSE_v criterion is that $\mathbb{E}_\mathcal{S}\mathbb{E}_{\boldsymbol{x}}(v_\mathtt{true}(\mathcal{S}, \boldsymbol{x}) - \hat{v}_\mathtt{q}(\mathcal{S}, \boldsymbol{x}))^2$ can be decomposed as

$$\mathbb{E}_\mathcal{S}\mathbb{E}_{\boldsymbol{x}}(v_\mathtt{true}(\mathcal{S}, \boldsymbol{x}) - \hat{v}_\mathtt{q}(\mathcal{S}, \boldsymbol{x}))^2 = \mathbb{E}_\mathcal{S}\mathbb{E}_{\boldsymbol{x}}(f(\boldsymbol{x}) - \hat{v}_\mathtt{q}(\mathcal{S}, \boldsymbol{x}))^2 - \mathbb{E}_\mathcal{S}\mathbb{E}_{\boldsymbol{x}}(f(\boldsymbol{x}) - v_\mathtt{true}(\mathcal{S}, \boldsymbol{x}))^2,$$

see Covert, Lundberg, and Lee (2020, Appendix A). The first term on the right-hand side can be estimated by (15), while the second term is a fixed (unknown) constant not influenced by the approach `q`. Thus, a low value of MSE_v in (15) indicates that the estimated contribution function $\hat{v}_{\texttt{q}}$ is closer to the true counterpart $v_{\texttt{true}}$ than a high value.

The MSE_v evaluation criterion is helpful as it does not need knowledge about the true explanations. Thus, we can apply it to real-world datasets where the true Shapley values are unknown. However, the criterion has two drawbacks. First, we can only use it to rank the methods and not assess their closeness to the optimum since the minimum value of the MSE_v criterion is unknown. Second, the criterion evaluates the contribution functions rather than the Shapley values. Thus, the estimates for $v(\mathcal{S})$ can undershoot and overshoot for different coalitions, but these errors might cancel each other out in the Shapley value formula (1).

Nevertheless, in a comprehensive simulation study, Olsen *et al.* (2024, Figure 11) observe a relatively linear relationship between the MSE_v evaluation scores and the mean absolute error (MAE) between the estimated and true Shapley values. That is, a method that achieves a low MSE_v score also tends to obtain a low MAE score, and vice versa. We therefore consider (15) to be a decent evaluation metric for Shapley value estimates.

3. The shapr R-package

The philosophy behind the **shapr** package is to provide a minimal set of user functions, which have good default settings for most use cases and are easy to use. Furthermore, we aim to offer extensive flexibility for advanced users to define both *how* computations should be performed, the desired level of accuracy, and the type of Shapley values that should be computed.

The **shapr** package is available from the Comprehensive R Archive Network (CRAN) at CRAN.R-project.org/package=shapr or from the package's GitHub repository at github.com/NorskRegnesentral/shapr. Documentation and the full suite of four vignettes showcasing the extensive functionality of the package are also easily accessible through **shapr**'s **pkgdown** (Wickham, Hesselberth, Salmon, Roy, and Brüggemann 2024) online documentation at norskregnesentral.github.io/shapr/. **shapr** can be installed directly from CRAN by:

```
R> install.packages("shapr")
```

Below we describe the basic usage, functionality, and flexibility of the **shapr** R package for explaining predictions with conditional Shapley values, and showcase its usage with a few practical code examples.

3.1. Basic usage

The `explain()` function is the main function of the **shapr** R package. The function is used to set up and execute all computations for producing Shapley value-based explanations for a set of predictions from a model, and takes the following form:

```
explain <- function(model,
                    x_explain,
                    x_train,
                    approach,
```

```
    phi0,
    iterative = NULL,
    max_n_coalitions = NULL,
    group = NULL,
    n_MC_samples = 1e3,
    seed = NULL,
    verbose = "basic",
    predict_model = NULL,
    get_model_specs = NULL,
    prev_shapr_object = NULL,
    asymmetric = FALSE,
    causal_ordering = NULL,
    confounding = NULL,
    extra_computation_args = list(),
    iterative_args = list(),
    output_args = list(),
    ...)
```

The many input arguments reflect the package's flexibility. Most of the arguments are related to *how* the estimation and computation of the Shapley values should be carried out and have good default values applicable for standard use cases. Only the following five arguments need to be set by the user:

- `model`: The model object whose predictions we want to explain.
- `x_explain`: The dataset whose predictions we want to explain.
- `x_train`: The dataset used to estimate the $v(\mathcal{S})$.
- `approach`: The approach used to estimate the $v(\mathcal{S})$. Allowed values are: `"gaussian"`, `"copula"`, `"empirical"`, `"ctree"`, `"vaeac"`, `"categorical"`, `"timeseries"`, `"independence"`, `"regression_separate"`, and `"regression_surrogate"`, corresponding to the approaches described in Section 2.3.
- `phi0`: The value to use for $\phi_0 = \mathbb{E}[f(\boldsymbol{x})]$. For (approximately) unbiased models, $f(\cdot)$, we recommend setting this to the mean of the response used to fit the model.

The output of `explain()` is an object of class `(shapr, list)` and contains the following elements:

- `shapley_values_est`: Table with the estimated Shapley values.
- `shapley_values_sd`: Table with standard deviations of the estimated Shapley values (with respect to the sampling uncertainty in KernelSHAP described in Section 2.1).
- `pred_explain`: The prediction outcome for the observations we are explaining.
- `MSEv`: The MSE_v evaluation criterion for the approach as described in Section 2.5.
- `iterative_results`: Details on the iterative estimation, its convergence, and the intermediate Shapley value estimates.

- `saving_path`: The path where the iterative results are saved.
- `internal`: Various additional information about the performed computations.
- `timing`: The time spent executing the various parts of the function call.

The `shapr` class provides three methods (attribution functions):

- `print.shapr()`: Displays a table of the computed Shapley values by default. The optional `what` argument can be used to show other components, such as standard deviations, MSE estimates, and timing summaries.
- `plot.shapr()`: Creates various visualizations of the estimated Shapley values using **ggplot2** (Wickham 2016). The function returns a `(gg, ggplot)` object, enabling full use of **ggplot2**'s customization features. By default, bar plots are produced for each explicand. Alternative visualizations, such as `"waterfall"`, `"scatter"`, and `"beeswarm"` plots, can be specified via the `plot_type` argument.
- `summary.shapr()`: Outputs a formatted summary of the Shapley value computation and invisibly returns a named list containing the summary components.

In addition, `get_results()` provides convenient access to all information related to the Shapley value computation. By default, it returns the same output as `summary.shapr()`, but it can also be called directly to extract specific components via the `what` argument.

3.2. Package functionality

The **shapr** package provides a wide range of functionality and flexibility related to the *type* of Shapley values computed, the way they are computed (computationally), and the feedback given to the user during the potentially long-running computations. Below, we describe the most important functionality and how it can be controlled and specified by the user of the package.

Supported models

shapr provides native support for explaining predictions from models fit using `stats::lm`, `stats::glm`, `ranger::ranger` (Wright and Ziegler 2017), `mgcv::gam` (Wood 2025), and `xgboost::xgboost`/`xgboost::xgb.train` (Chen, He, Benesty, Khotilovich, Tang, Cho, Chen, Mitchell, Cano, Zhou, Li, Xie, Lin, Geng, Li, and Yuan 2024), in addition to the hundreds of models available in the popular **parsnip** package (Kuhn and Vaughan 2024) when fit through the associated **workflows** (Vaughan and Couch 2025) package. The latter is recommended when using models that require one-hot encoding of categorical features, such as, e.g., **xgboost** (Chen *et al.* 2024), as **workflows** handles this transformation automatically, making the process easier and less error-prone for the user. We refer the reader to the **pkgdown** website of **workflows** for details on how to fit models with the **workflows** package. For all of these models, the user needs only to pass the model object to `explain()`. The **shapr** package will then check consistency between the `model` object and the data in `x_train` and `x_explain`, and generate the necessary predictions from the model when needed. Moreover, any other predictive model

that outputs a single numeric value can be explained using `explain()` by supplying a custom prediction function through the `predict_model` argument. The `predict_model` function should take `model` and a `data.frame/data.table` as arguments and return the associated predictions as a numeric vector. To enable feature and model checking, an additional function can be supplied via the `get_model_specs` argument. The required structure of this function is provided in Appendix A.1.

Group-wise Shapley values

In certain cases, particularly when the number of features is large, it may be more effective and informative to explain predictions using groups of features rather than individual ones. By defining coalitions of feature *groups* rather than single features, the computational complexity of the Shapley value computations may also be significantly reduced. Group-wise Shapley values can be computed with `explain()` by supplying a (named) list of feature vector names to `group`. For further intuition and real-world examples of group-wise Shapley values, see Jullum *et al.* (2021); Au, Herbinger, Stachl, Bischl, and Casalicchio (2022).

Combining approaches

In some use cases, it may be beneficial to use different approaches for estimating $v(\mathcal{S})$ for different coalitions $\mathcal{S}$. As suggested by Aas *et al.* (2021, Sec. 3.4), Izbicki and Lee (2017) among others, simpler models may be more appropriate for estimating conditional distributions on the form $p(\boldsymbol{x}_{\text{few}}|\boldsymbol{x}_{\text{many}})$ than $p(\boldsymbol{x}_{\text{many}}|\boldsymbol{x}_{\text{few}})$, where $\boldsymbol{x}_{\text{many}}$ is of (much) higher dimension than $\boldsymbol{x}_{\text{few}}$. For instance, a Gaussian distribution may sometimes be a fast and reasonable approach to estimate $p(\boldsymbol{x}_{\text{few}}|\boldsymbol{x}_{\text{many}})$ (i.e., when $|\mathcal{S}|$ is large), while being a poor approximation approach to estimate $p(\boldsymbol{x}_{\text{many}}|\boldsymbol{x}_{\text{few}})$ (i.e., when $|\mathcal{S}|$ is small). In the latter case, the more flexible *ctree* or *vaeac* approaches may be more appropriate. Combining approaches can be done in `explain()` by setting `approach` equal to a character vector of length $M - 1$, where M is the number of features. In that case, the j-th element specifies the approach to use for $v(\mathcal{S})$ when $|\mathcal{S}| = j$.

Causal and asymmetric Shapley values

Asymmetric (Frye, Rowat, and Feige 2020) and Causal (Heskes, Sijben, Bucur, and Claassen 2020) Shapley values are modified Shapley value frameworks that aim at incorporating causal knowledge into the explanations. The frameworks require knowledge or assumptions about a (partial) causal ordering of the features or feature-groups and whether confounders exist for the features/groups in the model. All of this must be specified through the arguments `asymmetric`, `causal_ordering`, and `confounding`. They are all disabled by default. Section 4 describes the methodologies and how to compute such Shapley values with the `explain()` function.

Flexible estimation with the regression paradigm

As described in Sections 2.2–2.3, the `separate_regression` approach trains a new regression model to estimate the $v(\mathcal{S})$ values for each coalition $\mathcal{S}$, while the `surrogate_regression` approach trains a single regression model for all coalitions $\mathcal{S}$. For these approaches, **shapr** allows using any regression model in the popular **parsnip** package. These regression models can also preprocess the data and be tuned to optimize their performance by leveraging other packages in the **tidymodels** framework (Kuhn and Wickham 2020), such as **recipes**, **tune**,

and **workflows** through the ellipsis arguments `regression.model`, `regression.tune_values`, `regression.vfold_cv_para`, `regression.recipe_func` and `regression.surrogate_n_comb` of `explain()`. The default regression model for both regression-based approaches (`approach = "regression_separate"` and `approach = "regression_surrogate"`) is a standard linear regression model. For more details, see the regression vignette "Shapley value explanations using the regression paradigm" available through `vignette("regression","shapr")` or the online documentation. We also provide an example of the `regression_surrogate` approach in Section 3.4.

Shapley values for forecasting models

The `explain()` function explains single-outcome predictions. However, in some cases, particularly in time series forecasting settings, we are interested in explaining multiple outcomes from the same observations. While in principle, this can be achieved by calling `explain()` multiple times with custom `predict_model`s as described above, it is not computationally efficient because the computationally intensive part (modeling of the conditional distributions) is repeated unnecessarily. Therefore, **shapr** offers a dedicated function, `explain_forecast()`, designed for the specific use-case of explaining forecasts from time series models at one or more future time steps. The usage and functionality of this function is described in Section 6.

Improved KernelSHAP

In Section 2.1, we discussed how paired sampling, variance-reducing reweighting, and semi-deterministic sampling of coalitions significantly improve the efficiency of KernelSHAP. The **shapr** package uses paired sampling (unless `asymmetric = TRUE`) and the reweighting method in (8) by default. Although not recommended, these can be disabled by setting `paired_shap_sampling = FALSE` and `kernelSHAP_reweighting = "none"` in the list passed to the `extra_computation_args` argument of `explain()`. Semi-deterministic sampling is disabled by default but can be enabled with `semi_deterministic_sampling = TRUE` in the same list. A few other reweighting strategies are also available. For a complete overview of all available options, as well as other computational defaults and settings that can be passed via `extra_computation_args`, see the help file of `get_extra_comp_args_default`.

Direct and iterative estimation

The estimation procedure for computing Shapley values is controlled by the `iterative` argument in `explain()`. The iterative estimation procedure with convergence detection described in Section 2.4 is mainly useful for reducing the runtime of the Shapley values computation when there are many features/feature-groups. When there are fewer Shapley values to compute, the computational gain of stopping early is often negated by the cost of computing the bootstrapped standard deviations of the Shapley values. Therefore, in **shapr**, the default behavior (`iterative = NULL`) is to use the direct estimation procedure when there are five or fewer features (or feature-groups), and the iterative procedure otherwise. In the former case `explain()` also uses all 2^M coalitions by default. The number of coalitions to use can be restricted through the argument `max_n_coalitions`. For the direct estimation procedure, this is the actual number of unique coalitions to use. The iterative procedure stops when the number of coalitions reaches this number. Specifics related to the iterative procedure and convergence criterion are set through the `iterative_args` argument, where the user can

set the convergence criterion parameter t in (14), the number of "burn-in" coalitions used in the first iteration, the maximum number of iterations, etc. For details about defaults and arguments, see the help file of `get_iterative_args_default`. In Section 3.4, we provide examples with both the direct and iterative estimation procedure.

Continued estimation

While the iterative estimation procedure with convergence detection aims to balance runtime and accuracy, there may be cases where further reduction of the estimating uncertainty is desired. **shapr** allows continuation of iterative or non-iterative Shapley value computations via the `prev_shapr_object` argument in `explain()`, which accepts either a `(shapr, list)` object or a file path to saved intermediate results from a previous call to `explain()`. For an object `ex` of class `(shapr, list)`, the path is accessible via `ex$saving_path`, which defaults to a file in `tempdir()`. This feature is useful for resuming interrupted computations or improving the accuracy by continuing iterative estimation. To include additional coalitions, adjustments to `max_n_coalitions` or the `iterative_args` parameters described above may be necessary. We provide an example of this functionality in Section 3.4.

Parallelized batch computations

Estimation of the conditional expectations in $v(\mathcal{S}) = \mathbb{E}_{\boldsymbol{x}_{\bar{\mathcal{S}}}}[f(\boldsymbol{x})|\boldsymbol{x}_{\mathcal{S}}]$ is the most computationally intensive part of the Shapley value computation. When relying on Monte Carlo-based estimation approaches, these computations may also involve processing a large amount of data and may, therefore, be somewhat memory-intensive if handled all at once. To reduce memory usage, **shapr** supports batch-wise estimation of $v(\mathcal{S})$, discarding intermediate objects and data between batches. In the iterative estimation procedure, batch computations are performed within each iteration. To reduce runtime, **shapr** also supports parallelization over these batches (within every iteration). The parallelized computations are handled by the **future** framework (Bengtsson 2021), enabling flexible parallel computations on all major operating systems, in addition to computing clusters, completely controlled by the user outside of **shapr**. We give an example of how to specify the parallelization setup in Section 3.4. Batch computations do increase the runtime by a small amount, and it is therefore advised not to use too many batches if memory consumption is not an issue. Users can control the batch size and number of them by specifying `min_n_batches`, `max_batch_size` as named list elements passed to the `extra_computation_args` argument of `explain()`. If not specified, both default to 10, typically providing a decent trade-off between computation speed and memory consumption. While parallelization speeds up computations, it also increases memory consumption and incurs some overhead from duplicating data across processes, so it is often beneficial to limit the number of parallel sessions.

Verbosity and progress reports

Since conditional Shapley value computations often take time, it can be useful for the user to get feedback on the current stage of the computation process. In **shapr**, the verbosity of the output is controlled by the `verbose` argument, taking one or more strings as input. It can provide information about the computation to be performed (`"basic"`, default), the current stage of the estimation procedure (`"progress"`), how close to convergence the iterative procedure is (`"convergence"`), intermediate Shapley values estimates, (`"shapley"`), and details

about the $v(\mathcal{S})$ estimation process (`"vS_details"`). The output is neatly formatted using the **cli** package (Csárdi 2025).

In addition, progress bars for the $v(\mathcal{S})$ computations are available through the **progressr** package (Bengtsson 2024), which are specified separately from the **shapr** package. **progressr** integrates seamlessly with the **future** parallelization framework (Bengtsson 2021) and is, to our knowledge, the only tool that provides progress updates for parallel tasks in R. These progress bars can be used alongside or independently of the `verbose` argument.

Comparing approaches

Following a comparative study of different approaches for estimating $v(\mathcal{S})$, Olsen *et al.* (2024) provide some rough practical guidelines for how to choose the most appropriate approach. As the performance may vary quite a bit depending on the actual feature distribution, we generally recommend comparing the performance of multiple approaches using the MSE_v evaluation criterion described in Section 2.5. In **shapr**, the MSE_v scores are automatically computed and returned by `explain()`. The metric can be plotted using the `plot_MSEv_eval_crit()` function, which takes a list of outputs from `explain()` as the main argument.

3.3. Implementation details

Algorithm 1 provides an overview of the structure of `shapr::explain()`. As shown, the function is divided into several smaller components, each responsible for a specific part of the computation. These components operate on a shared list named `internal`, which serves as both input and output, and stores parameters, settings, and intermediate results throughout the process. This modular design also facilitates a relatively simple **shaprpy** Python implementation (see Section 5).

Since computing (conditional) Shapley values is computationally demanding, we emphasize efficiency in our implementation. We rely on the fast and memory-efficient **data.table** package (Barrett, Dowle, Srinivasan, Gorecki, Chirico, Hocking, and Schwendinger 2024) for nearly all internal data operations. To further enhance performance, we write computationally demanding code in C++, using the **Rcpp** and **RcppArmadillo** packages (Eddelbuettel, Francois, Allaire, Ushey, Kou, Russell, Ucar, Bates, and Chambers 2024a; Eddelbuettel, Francois, Bates, Ni, and Sanderson 2024b). Finally, the computations in the `compute_vS` function can be parallelized, with full user control via the **future** framework (Bengtsson 2021).

Moreover, we employ various methodological techniques to accelerate the computation of $v(\mathcal{S})$. For example, in the `gaussian` approach, we efficiently reuse an initial set of samples when generating K Monte Carlo samples from $p(\boldsymbol{x}_{\bar{\mathcal{S}}}|\boldsymbol{x}_{\mathcal{S}} = \boldsymbol{x}^*_{\mathcal{S}})$ for the potentially many explicands $\boldsymbol{x}^*$ and subsets $\mathcal{S}$ (within the same batch): First, we generate $p \times K$ standard normally distributed variables and split them into vectors of size p. Then, for each $\mathcal{S}$, we multiply these vectors by the Cholesky decomposition of their associated conditional covariance matrix in a computationally efficient, vectorized manner. Finally, we add the conditional means, which differ for each explicand $\boldsymbol{x}^*$. Internal benchmarks have shown that our implementation of this procedure in C++ provides speedups of several orders of magnitude compared to the standard approach of separate sampling solely in R. This same technique is applied to the internal Gaussian samples in the `copula` approach.

Algorithm 1 Rough structure of `shapr::explain()`

converged ← FALSE
`internal` ← `list()` ▷ Creates the `internal` list
`internal` ← `setup()` ▷ Checks arguments and sets (default) parameters
Call `test_predict_model(internal)` ▷ Tests the prediction model
Some additional setup for special cases
Call `cli_startup(internal)` ▷ Prints basic verbosity
while converged = FALSE **do**
  `internal` ← `shapley_setup(internal)` ▷ Samples coalitions
  `internal` ← `setup_approach(internal)` ▷ Prepares the approach(es)
  `vS_list` ← `compute_vS(internal)` ▷ Estimates $v(\mathcal{S})$ (in parallel)
  `internal` ← `compute_estimates(vS_list, internal)` ▷ Computes ϕ_j, sd(ϕ_j)
  `internal` ← `check_convergence(internal)` ▷ Checks if convergence is reached
  Call `save_results()` ▷ Saves the intermediate results
  `internal` ← `prepare_next_iteration(internal)` ▷ Prepares next iteration
  Call `print_iter(internal)` ▷ Prints iterative estimates
  `converged` ← from `internal` ▷ Set to TRUE if converged
end while
`output` ← `finalize_explanation(internal)` ▷ Extracts the computations to return
`output` ← `compute_time(output)` ▷ Computes and gathers the task specific timing
Return `output`

3.4. Examples

Below, we provide some basic use cases of the **shapr** package. More extensive examples are available through the package's "General usage" vignette available through `vignette("general_usage","shapr")` or the online documentation.

We demonstrate **shapr** on the bike sharing dataset from the UCI Machine Learning Repository. We follow Heskes *et al.* (2020) who use this dataset and model the daily bike rental counts based on $M = 7$ features, where some are hand-crafted from other features in the original dataset. The model is a simple **xgboost** model with default hyperparameters and 100 trees. The features are the number of days since Jan 2011 (`trend`), two cyclical variables representing the season (`cosyear`, `sinyear`), temperature (`temp`), temperature feel (`atemp`), wind speed (`windspeed`), and humidity (`hum`). 80% of the 731 observations are used to train the model, while the remaining 20% of the data are held out for testing (and explanation in the present case). The script used to prepare the data is available in the **shapr** GitHub repository.

We first load the required packages and read in the processed data and fitted model:

```
R> library(xgboost)
R> library(data.table)
R> library(shapr)
R> x_explain <- fread(file.path("data_and_models", "x_explain.csv"))
R> x_train <- fread(file.path("data_and_models", "x_train.csv"))
R> y_train <- unlist(fread(file.path("data_and_models", "y_train.csv")))
R> model <- readRDS(file.path("data_and_models", "model.rds"))
```

We utilize the **future** and **progressr** packages to enable parallelized computations (using 4

parallel sessions) with progress updates for the $v(\mathcal{S})$ estimation as follows:

```
R> library(future)
R> library(progressr)
R> future::plan(multisession, workers = 4)
R> progressr::handlers(global = TRUE)
```

We then show how to explain the 146 explicands in `x_explain` with the `independence` and `ctree` approaches, using a maximum of 40 coalitions:

```
R> # 40 indep
R> exp_40_indep <- explain(model = model,
+                          x_explain = x_explain,
+                          x_train = x_train,
+                          max_n_coalitions = 40,
+                          approach = "independence",
+                          phi0 = mean(y_train),
+                          verbose = NULL,
+                          seed = 1)

R> # 40 ctree
R> exp_40_ctree <- explain(model = model,
+                          x_explain = x_explain,
+                          x_train = x_train,
+                          max_n_coalitions = 40,
+                          approach = "ctree",
+                          phi0 = mean(y_train),
+                          verbose = NULL,
+                          ctree.sample = FALSE,
+                          seed = 1)
```

We compare the two approaches based on their resulting MSE_v:

```
R> print(exp_40_indep, what = "MSEv")
       MSEv MSEv_sd
      <num>   <num>
 1: 1730485  113303
R> print(exp_40_ctree, what = "MSEv")
       MSEv MSEv_sd
      <num>   <num>
 1: 1308239   93916
```

The `ctree` method is clearly the best. We thus print its estimated Shapley values.

```
R> print(exp_40_ctree)
      explain_id  none trend cosyear sinyear  temp atemp windspeed      hum
           <int> <num> <num>   <num>   <num> <num> <num>     <num>    <num>
   1:          1  4537 -2529    -617    6.03  -274  -242     159.2 -293.094
   2:          2  4537 -1284    -457  -40.97  -511 -1008     170.6   67.519
   3:          3  4537 -1128    -496 -163.80  -576  -945      43.5    0.485
   4:          4  4537 -1472    -323 -126.43  -566  -947     417.5 -540.775
   5:          5  4537 -1363    -325 -207.16  -912 -1268     439.0  128.800
  ---
 142:        142  4537   619    -260  249.48  -429  -495     208.9  287.379
 143:        143  4537   998    -456  108.38  -144  -234     443.9   85.584
 144:        144  4537  1263    -428   62.70   370   306     458.3 -164.642
 145:        145  4537   335    -367   90.14 -1327  -812     239.8  125.416
 146:        146  4537 -1048    -715   97.56  -763  -958     807.2 -709.912
```

By visually inspecting this subset of explanations, it seems that `trend` and `temp`/`atemp` are highly influential features. Calling `summary(exp_40_ctree)` outputs a structured summary of the Shapley value computation to the console, as illustrated in Figure 2.

```
── Summary of Shapley value explanation ──────────────────────────────────────
• Computed with `shapr::explain()` in 8.1 seconds, started YYYY-mm-DD HH:MM:SS
• Model class: <xgb.Booster>
• v(S) estimation class: Monte Carlo integration
• Approach: ctree
• Procedure: Iterative
• Number of Monte Carlo integration samples: 1000
• Number of feature-wise Shapley values: 7
• Number of observations to explain: 146
• Number of coalitions used: 40 (of total 128)
• Computations (temporary) saved at: /tmp/RtmpjaHaq2/shapr_obj_775b270a42d3.rds

── Convergence info
✓ Iterative Shapley value estimation stopped at 40 coalitions after 3 iterations, due to:
Maximum number of coalitions (40) reached!

── Estimated Shapley values (sd in parentheses)
     explain_id       none             trend           cosyear          sinyear
          <int>     <char>            <char>            <char>           <char>
  1:          1 4536.6 (0) -2529.41 ( 79.70)  -616.74 ( 98.93)    6.03 (106.75)
  2:          2 4536.6 (0) -1284.26 (118.35)  -457.32 (148.46)  -40.97 (158.01)
  3:          3 4536.6 (0) -1127.96 (137.32)  -495.85 (161.36) -163.80 (176.38)
  4:          4 4536.6 (0) -1472.44 (101.77)  -322.88 (134.74) -126.43 (136.08)
  5:          5 4536.6 (0) -1363.28 (105.86)  -325.38 (137.72) -207.16 (128.66)
 ---
142:        142 4536.6 (0)   618.85 ( 49.01)  -259.76 ( 55.34)  249.48 ( 54.32)
143:        143 4536.6 (0)   997.69 ( 59.67)  -456.20 ( 92.07)  108.38 ( 85.90)
144:        144 4536.6 (0)  1263.50 ( 66.91)  -428.00 ( 95.02)   62.70 (107.13)
145:        145 4536.6 (0)   335.20 ( 52.46)  -366.65 ( 88.32)   90.14 ( 77.38)
146:        146 4536.6 (0) -1047.57 (126.52)  -715.36 (112.29)   97.56 (148.37)
                 temp             atemp         windspeed              hum
               <char>            <char>            <char>           <char>
  1:  -274.38 (128.88)  -242.30 (117.60)   159.24 ( 95.45)  -293.09 ( 77.63)
  2:  -511.18 (431.30) -1008.01 (290.03)   170.62 (272.14)    67.52 (149.47)
  3:  -576.46 (493.87)  -945.36 (339.62)    43.53 (326.09)     0.49 (186.44)
  4:  -566.33 (396.04)  -947.30 (246.10)   417.46 (222.04)  -540.77 (147.21)
  5:  -911.89 (406.55) -1267.58 (263.39)   439.00 (214.80)   128.80 (147.61)
 ---
142:  -428.74 ( 59.19)  -494.87 ( 70.30)   208.94 ( 67.09)   287.38 ( 59.33)
143:  -144.17 (208.53)  -234.48 (132.51)   443.87 (121.71)    85.58 ( 71.95)
144:   369.88 (232.50)   305.90 (160.55)   458.32 (155.32)  -164.64 (111.92)
145: -1326.52 (111.07)  -812.39 (105.68)   239.78 ( 73.36)   125.42 ( 65.05)
146:  -762.61 (221.63)  -957.60 (153.70)   807.22 (223.91)  -709.91 (161.48)

── Estimated MSEv
Estimated MSE of v(S) = 1308239 (with sd = 93916)
```

Figure 2: The output displayed in the console when calling `summary(exp_40_ctree)`.

Next, we show how to resume estimation with the `ctree` approach from the first 40 coalitions, and also enable verbose output to monitor convergence. The output is shown in Figure 3.

```
── Starting `shapr::explain()` at YYYY-mm-DD HH:MM:SS ──────────────────────────
ℹ Feature classes extracted from the model contain `NA`.
  Assuming feature classes from the data are correct.
ℹ `max_n_coalitions` is `NULL` or larger than `2^n_features = 128`, and is therefore set to `2^n_features = 128`.

── Explanation overview ──

• Model class: <xgb.Booster>
• v(S) estimation class: Monte Carlo integration
• Approach: ctree
• Procedure: Iterative
• Number of Monte Carlo integration samples: 1000
• Number of feature-wise Shapley values: 7
• Number of observations to explain: 146
• Computations (temporary) saved at: /tmp/RtmpjaHaq2/shapr_obj_775b7e7197cb.rds

── Iterative computation started ──

── Iteration 4 ─────────────────────────────────────────────────────────────────
ℹ Using 66 of 128 coalitions, 26 new.

── Convergence info
ℹ Not converged after 66 coalitions:
Current convergence measure: 0.045 [needs 0.02]
Estimated remaining coalitions: 62
(Conservatively) adding about 40% of that (24 coalitions) in the next iteration.

── Iteration 5 ─────────────────────────────────────────────────────────────────
ℹ Using 90 of 128 coalitions, 24 new.

── Convergence info
✓ Iterative Shapley value estimation stopped at 90 coalitions after 5 iterations, due to:
Standard deviation convergence threshold (0.02) reached: 0.019!
```

Figure 3: The output to the console when creating `exp_iter_ctree` with `explain()` using `verbose = c("basic", "convergence")`.

```
R> exp_iter_ctree <- explain(model = model,
+                            x_explain = x_explain,
+                            x_train = x_train,
+                            approach = "ctree",
+                            phi0 = mean(y_train),
+                            prev_shapr_object = exp_40_ctree,
+                            ctree.sample = FALSE,
+                            verbose = c("basic", "convergence"),
+                            seed = 1)
```

To further investigate the results, we create a scatter plot of two of the features (`atemp` and `windspeed`). The plot is displayed in Figure 4.

```
R> library(ggplot2)
R> plot(exp_iter_ctree,
+       plot_type = "scatter",
+       scatter_features = c("atemp", "windspeed"))
```

Finally, we showcase how group-wise Shapley value explanations can be provided, using the `regression_separate` approach with both a default and tuned **xgboost** (Chen *et al.* 2024) model. The features are divided into three groups based on their type (temperature, time and weather).

```
R> group <- list(temp = c("temp", "atemp"),
+                time = c("trend", "cosyear", "sinyear"),
+                weather = c("hum","windspeed"))
```

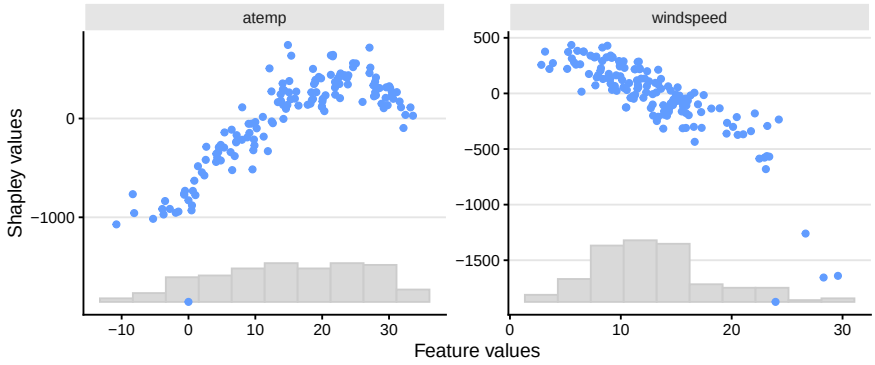


Figure 4: Scatter plot of the Shapley values for the two features `atemp` and `windspeed`, created by `plot.shapr`. The Shapley values are largest for `atemp` around 20 and `windspeed` less than 10, meaning that in these situations, the temperature and wind speed increases the predicted number of bike rentals the most. For very low `atemp` and high `windspeed`, the predicted number of bike rentals is greatly reduced, which makes intuitive sense.

```
R> exp_g_reg <- explain(model = model,
+                       x_explain = x_explain,
+                       x_train = x_train,
+                       phi0 = mean(y_train),
+                       group = group,
+                       approach = "regression_separate",
+                       regression.model = parsnip::boost_tree(
+                         engine = "xgboost",
+                         mode = "regression"
+                       ),
+                       verbose = NULL,
+                       seed = 1)

R> tree_vals <- c(10, 15, 25, 50, 100, 500)
R> exp_g_reg_tuned <- explain(model = model,
+                              x_explain = x_explain,
+                              x_train = x_train,
+                              phi0 = mean(y_train),
+                              group = group,
+                              approach = "regression_separate",
+                              regression.model = parsnip::boost_tree(
+                                trees = hardhat::tune(),
+                                engine = "xgboost",
+                                mode = "regression"
+                              ),
+                              regression.tune_values = expand.grid(
+                                trees = tree_vals
+                              ),
```

```
+                                regression.vfold_cv_para = list(v = 5),
+                                verbose = NULL,
+                                seed = 1)
```

We then print and compare their resulting MSE_v scores and computation time.

```
R> print(exp_g_reg, what = "MSEv")
       MSEv MSEv_sd
      <num>   <num>
 1: 1547240  142123
R> print(exp_g_reg_tuned, what = "MSEv")
       MSEv MSEv_sd
      <num>   <num>
 1: 1534033  142277

R> print(exp_g_reg, what = "timing_summary")
              init_time            end_time total_time_secs total_time_str
                 <POSc>              <POSc>           <num>         <char>
 1: YYYY-MM-DD 21:05:33 YYYY-MM-DD 21:05:35            2.17    2.2 seconds

R> print(exp_g_reg_tuned, what = "timing_summary")
              init_time            end_time total_time_secs total_time_str
                 <POSc>              <POSc>           <num>         <char>
 1: YYYY-MM-DD 21:05:35 YYYY-MM-DD 21:05:42            6.81    6.8 seconds
```

As we can see, the tuned version is marginally better (though far from significant), and it also takes three times as long to run. Finally, we plot group-wise Shapley values for a specific observation (observation 6 in `x_explain`) of the tuned version, giving the plot in Figure 5.

```
R> plot(exp_g_reg_tuned,
+       index_x_explain = 6,
+       plot_type = "waterfall")
```

4. Asymmetric and causal Shapley values with shapr

In some cases, we have knowledge about the causal structure within the data we are modeling, but traditional Shapley values do not account for this. Asymmetric Shapley values and causal Shapley values address this limitation by incorporating causal structures into their explanations, thereby offering more informative explanations in these scenarios.

4.1. Overview

Asymmetric (Frye *et al.* 2020) and causal (Heskes *et al.* 2020) Shapley values aim to incorporate causal knowledge into the explanations. Both frameworks rely on the user specifying a (partial) causal ordering, where features that are treated on an equal footing are linked together with undirected edges and become part of the same chain component τ_i. Edges

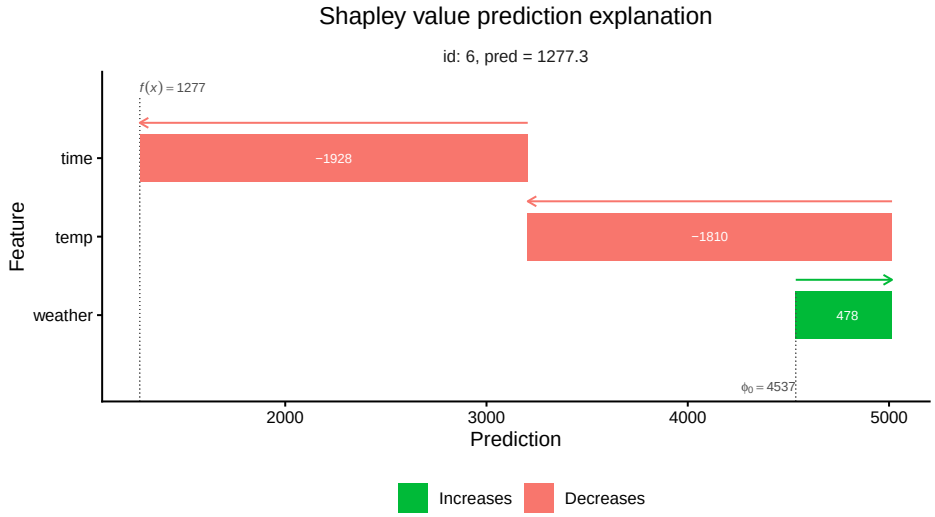


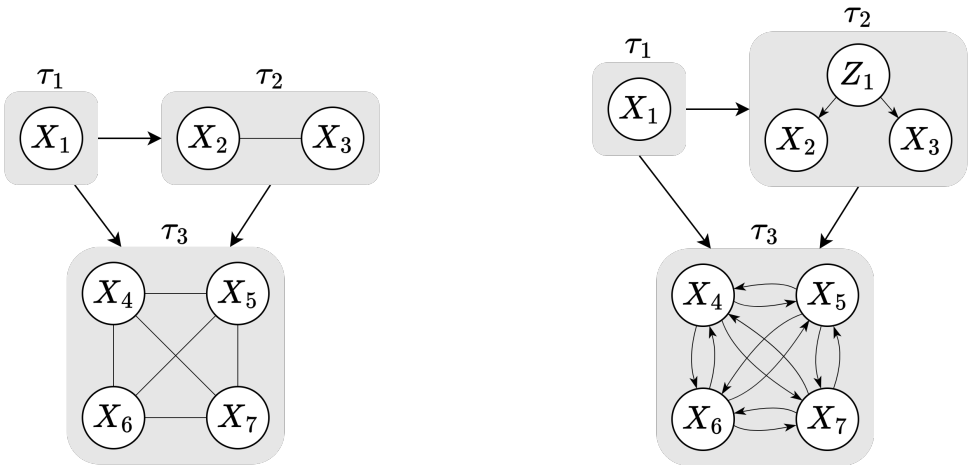
Figure 5: Illustration of a waterfall plot of a single explicand with group-wise Shapley values created by `plot.shapr`. The waterfall plot illustrates how the cumulative contributions of feature groups sum to the prediction, starting from the mean prediction/response ϕ_0 and adding contributions in order of increasing absolute value. For this explicand, the `weather` group increases the predicted number of bike rentals slightly, while the `time` and `temp` groups reduce it considerably.

between chain components τ_i and τ_j are directed and represent causal relationships. Additionally, in the causal Shapley value framework, the user must specify if component τ_i is subject to confounding or not. Together, they form a causal chain graph with directed and undirected edges. Let the chain components with and without confounding be denoted by $\mathcal{T}_{\text{confounding}}$ and $\mathcal{T}_{\overline{\text{confounding}}}$, respectively. This allows us to correctly distinguish between dependencies that are due to confounding and mutual interactions. In Figure 6, we visualize a causal ordering and the corresponding causal chain graph in an $M = 7$-dimensional setting when we assume confounding in τ_2 but no confounding in τ_1 and τ_3. This causal structure is assumed in the example below.

Frye *et al.* (2020) achieve the incorporation of causal knowledge into the explanations by estimating the Shapley values solely based on coalitions that respect the causal ordering. This means that not all $2^M = 128$ coalitions are used, where M is the number of features. For example, in Figure 6, we see that X_1 is the ancestor of X_2; thus, asymmetric Shapley values omit coalitions where X_2 is included and X_1 is excluded. For the causal ordering in Figure 6, there are 20 valid coalitions. Only using these will skew the explanations towards distal/root causes (Frye *et al.* 2020, Sec. 3.2).

Heskes *et al.* (2020) argue that causal Shapley values offer a more direct and robust way to incorporate causal knowledge than asymmetric Shapley values. They redefine $v(\mathcal{S})$ to

$$v(\mathcal{S}) = \mathbb{E}_{\boldsymbol{x}_{\bar{\mathcal{S}}}}\left[f(\boldsymbol{x}_{\bar{\mathcal{S}}}, \boldsymbol{x}_{\mathcal{S}}) \mid \text{do}(\boldsymbol{x}_{\mathcal{S}} = \boldsymbol{x}^*_{\mathcal{S}})\right] = \int f(\boldsymbol{x}_{\bar{\mathcal{S}}}, \boldsymbol{x}^*_{\mathcal{S}}) p(\boldsymbol{x}_{\bar{\mathcal{S}}} \mid \text{do}(\boldsymbol{x}_{\mathcal{S}} = \boldsymbol{x}^*_{\mathcal{S}}))\, \mathrm{d}\boldsymbol{x}_{\bar{\mathcal{S}}},$$



(a) `confounding = NULL` (b) `confounding = c(FALSE, TRUE, FALSE)`

Figure 6: Schematic overview of `causal_ordering = list(`τ_1 `= 1,` τ_2 `= c(2,3),` τ_3 `= c(4,5,6,7))` without and with confounding specified in the second component.

Framework	Sampling	`asymmetric`	`causal_ordering`	`confounding`	`approach`
Sym. conditional	$P(\boldsymbol{X}_{\bar{\mathcal{S}}}\|\boldsymbol{X}_{\mathcal{S}} = \boldsymbol{x}_{\mathcal{S}})$	`FALSE`	`NULL`	`NULL`	All
Asym. conditional	$P(\boldsymbol{X}_{\bar{\mathcal{S}}}\|\boldsymbol{X}_{\mathcal{S}} = \boldsymbol{x}_{\mathcal{S}})$	`TRUE`	`list(...)`	`NULL`	All
Sym. causal	$P(\boldsymbol{X}_{\bar{\mathcal{S}}}\|\,\mathrm{do}(\boldsymbol{X}_{\mathcal{S}} = \boldsymbol{x}_{\mathcal{S}}))$	`FALSE`	`list(...)`	`c(...)`	All MC-based
Asym. causal	$P(\boldsymbol{X}_{\bar{\mathcal{S}}}\|\,\mathrm{do}(\boldsymbol{X}_{\mathcal{S}} = \boldsymbol{x}_{\mathcal{S}}))$	`TRUE`	`list(...)`	`c(...)`	All MC-based
Sym. marginal	$P(\boldsymbol{X}_{\bar{\mathcal{S}}})$	`FALSE`	`NULL`	`TRUE`	`indep.`, `gaussian`

Table 1: Overview of the Shapley value methodologies supported by **shapr** and how to compute them by altering the `asymmetric`, `causal_ordering`, and `confounding` arguments in the `explain()` function. Setting `causal_ordering` to `NULL` is equivalent to a causal ordering with one component containing all (groups of) features. By `list(...)`, we mean a list of vectors indicating the (groups of) features in each component in the partial ordering, and `c(...)` represents a vector of booleans. Finally, `approach` indicates the estimation approaches in Section 2.3 compatible with each framework.

where the do operator stems from Pearl's do-calculus (Pearl 1995, 2012). Furthermore,

$$p(\boldsymbol{x}_{\bar{\mathcal{S}}} \mid \mathrm{do}(\boldsymbol{x}_{\mathcal{S}} = \boldsymbol{x}^*_{\mathcal{S}})) = \prod_{\tau \in \mathcal{T}_{\text{confounding}}} p(\boldsymbol{x}_{\tau \cap \bar{\mathcal{S}}} \mid \boldsymbol{x}_{\mathrm{pa}(\tau) \cap \bar{\mathcal{S}}}, \boldsymbol{x}_{\mathrm{pa}(\tau) \cap \mathcal{S}}) \times \prod_{\tau \in \mathcal{T}_{\overline{\text{confounding}}}} p(\boldsymbol{x}_{\tau \cap \bar{\mathcal{S}}} \mid \boldsymbol{x}_{\mathrm{pa}(\tau) \cap \bar{\mathcal{S}}}, \boldsymbol{x}_{\mathrm{pa}(\tau) \cap \mathcal{S}}, \boldsymbol{x}_{\tau \cap \mathcal{S}}), \tag{16}$$

where $\mathrm{pa}(\tau)$ are the parent features of chain component τ. For specific causal chain graphs, (16) simplifies to the marginal $p(\boldsymbol{x}_{\bar{\mathcal{S}}})$ and conditional $p(\boldsymbol{x}_{\bar{\mathcal{S}}} \,|\, \boldsymbol{x}_{\mathcal{S}} = \boldsymbol{x}^*_{\mathcal{S}})$ distributions. Consequently, the causal Shapley values generalize the marginal and conditional Shapley values (Heskes *et al.* 2020, Suppl. Cor. 1-3). By considering only the coalitions that respect the causal ordering, we obtain *asymmetric causal Shapley values*.

Causal Shapley values are computed by altering the Monte Carlo sampling procedure discussed in Section 2.2 based on the causal chain graph. Specifically, generating $\boldsymbol{x}^{(k)}_{\bar{\mathcal{S}}} \sim p(\boldsymbol{x}_{\bar{\mathcal{S}}} \mid \mathrm{do}(\boldsymbol{x}_{\mathcal{S}} = \boldsymbol{x}^*_{\mathcal{S}}))$ consists of a chain of sampling steps defined by the causal ordering.

4.2. Implementation details

The `shapr` package implements both asymmetric and causal Shapley values by adjusting the `asymmetric`, `causal_ordering`, and `confounding` arguments in the `explain()` function (Section 3.1). Asymmetric Shapley values are partially implemented in **shapFlex** (Redell 2019), though it is currently in an experimental state and has not been maintained for the past 5 years. Causal Shapley values are implemented in **CauSHAPley** (Heskes *et al.* 2020), building on an older version of **shapr** and is limited to the `gaussian` approach only.

In Table 1, we provide an overview of how to compute the different Shapley value versions with **shapr**. Asymmetric Shapley values are compatible with all estimation approaches discussed in Section 2.3, whereas causal Shapley values are only applicable to the Monte Carlo-based approaches. Both methodologies support groups of features, provided the causal ordering is specified at the group level rather than for individual features. All approaches estimate the marginal distributions by sampling from the training data, except the `gaussian` method, which samples from the marginals of the estimated Gaussian distribution.

Asymmetric Shapley values are implemented by computing the allowed coalitions based on the causal graph and either sampling from these coalitions or using all of them. To compute causal Shapley values, **shapr** computes and iterates over the steps in the sampling chains needed to generate $\boldsymbol{x}_{\bar{\mathcal{S}}}^{(k)} \sim p(\boldsymbol{x}_{\bar{\mathcal{S}}} \mid \text{do}(\boldsymbol{x}_{\mathcal{S}} = \boldsymbol{x}_{\mathcal{S}}^*))$ for each coalition separately. The separate treatment ensures that all other functionalities described in Section 3.2 for conditional Shapley values also apply to causal Shapley values.

A drawback of this separate treatment is that some sampling steps can occur in multiple chains, leading to repeated estimation of the corresponding conditional distributions. This has no significant runtime impact for easily trained approaches like `gaussian` and `copula`, but it can affect slower approaches like `ctree`. This issue does not influence the `vaeac` method, as it identifies the unique sampling steps across all chains and trains on these once. Additionally, step-wise sampling requires all but the first step in a chain to call the approach K times per explicand, where K is the number of Monte Carlo samples. This increased number of calls will impact the runtime of `explain()` for slower approaches. For computing asymmetric Shapley values in medium to high dimensions or when runtime is a concern, we therefore recommend using the optimized `gaussian` and `copula` approaches when applicable and the `vaeac` approach otherwise.

4.3. Example

Below we demonstrate the asymmetric and causal Shapley value frameworks on the same data and model as in Section 3.4. More extensive examples are available through the package's vignette "Asymmetric and causal Shapley value explanations" available through `vignette("asymmetric_causal", "shapr")` or the online documentation. Once again, we follow Heskes *et al.* (2020) who consider the first three features (`trend`, `cosyear`, and `sinyear`) to be potential causes of the four weather-related features (`temp`, `atemp`, `windspeed`, `hum`). Following Heskes *et al.* (2020), we set $\tau_1 = \{\texttt{trend}\}$, $\tau_2 = \{\texttt{cosyear}, \texttt{sinyear}\}$, and $\tau_3 = \{\texttt{temp}, \texttt{atemp}, \texttt{windspeed}, \texttt{hum}\}$, assuming confounding only in τ_2. This setup corresponds to the causal chain graph in Figure 6.

We compute and compare four different Shapley value variants: "Asymmetric causal", "Asymmetric conditional", "Symmetric conditional" and "Symmetric marginal" in accordance with

the descriptions in Table 1, all assuming a Gaussian feature distribution. We begin by specifying the causal and confounding structure outlined above.

```
R> causal_order0 <- list("trend",
+                        c("cosyear", "sinyear"),
+                        c("temp", "atemp", "windspeed", "hum"))

R> confounding0 <- c(FALSE, TRUE, FALSE)
```

Next, we define the four variants, compute their Shapley values, and generate a list of so-called beeswarm plots to compare the results.

```
R> exp_names <- c("Asymmetric causal", "Asymmetric conditional",
+                 "Symmetric conditional", "Symmetric marginal")

R> causal_ordering_list <- list(causal_order0, causal_order0, NULL, NULL)
R> confounding_list <- list(confounding0, NULL, NULL, TRUE)
R> asymmetric_list <- list(TRUE, TRUE, FALSE, FALSE)

R> plot_list <- list()
R> for(i in seq_along(exp_names)){
+    exp_tmp <- explain(model = model,
+                       x_train = x_train,
+                       x_explain = x_explain,
+                       approach = "gaussian",
+                       phi0 = mean(y_train),
+                       asymmetric = asymmetric_list[[i]],
+                       causal_ordering = causal_ordering_list[[i]],
+                       confounding = confounding_list[[i]],
+                       seed = 1,
+                       verbose = NULL)
+
+    plot_list[[i]] <- plot(exp_tmp, plot_type = "beeswarm", print_ggplot = FALSE) +
+      ggplot2::ggtitle(exp_names[i]) + ggplot2::ylim(-3700, 3700)
+ }
```

Finally, we use the **patchwork** package (Pedersen 2024) to combine the beeswarm plots into a single figure, as shown in Figure 7.

```
R> library(patchwork)
R> patchwork::wrap_plots(plot_list, nrow = 1) +
+    patchwork::plot_layout(guides = "collect")
```

Heskes *et al.* (2020) argue that asymmetric Shapley values focus on root causes, symmetric marginal Shapley values emphasize direct effects, and symmetric causal Shapley values consider both for a more comprehensive explanation. As seen from the beeswarm plots in Figure 7, there is only a minimal difference between the causal and conditional asymmetric methods in this case. The asymmetric frameworks often assign larger Shapley values (in magnitude) to

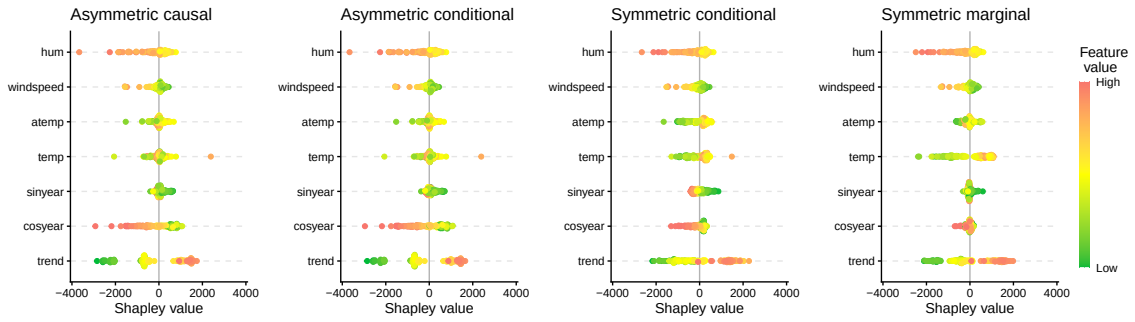


Figure 7: Beeswarm plots of the Shapley values computed by different Shapley value versions.

the root cause `cosyear` (season) compared to the direct effect of `temp` (temperature), whose Shapley values are near zero. Conversely, the two symmetric frameworks attribute more importance to `temp`, while clustering `cosyear` near zero, especially the marginal version. These observations align with Heskes *et al.* (2020).

5. The shaprpy Python library

While the **shapr** package provides methods for computing conditional Shapley values for models fitted in R, it naturally leaves models fitted in Python unsupported.[4] The goal of **shaprpy** is to allow users to compute Shapley values for models fitted in Python, with the ease of doing so directly and efficiently within the Python environment. **shaprpy** is a lightweight wrapper around **shapr**, internally calling functions from the R package.

Below, we introduce the **shaprpy** library, discuss its usage, supported models, and limitations, and provide an example.

5.1. The library

The **shaprpy** wrapper leans heavily on the **rpy2** library (Gautier 2024), which provides a neat and function-rich interface for running R embedded in a Python script, function or library. To use **shaprpy**, R must be installed, along with **shapr**, its dependencies and any additional packages required for specific functionality. We provide installation instructions under the *Python* pane in our online documentation.

We chose to implement **shaprpy** as a wrapper, rather than a full re-implementation in Python, in order to minimize the maintenance required on the Python side. **shaprpy** closely follows the structure of **shapr** as outlined in Algorithm 1, and delegates all tasks that do not require direct model access to the existing R functions listed in the algorithm. For this reason most bug fixes, new methodologies, and additional features in **shapr** will be directly applicable to **shaprpy** as well. Note that **rpy2** has limited support on Windows. **shaprpy** has mainly been tested on Linux.

Similar to **shapr**, the main function for user interaction in **shaprpy** is `explain()`. It takes all the same inputs as the R package, except the `prev_shapr_object` object for continued

[4]While packages like **reticulate** (Ushey, Allaire, and Tang 2024) allow calling Python code from within R, it is not straightforward to efficiently load Python *objects* within such sessions.

estimation, which is not supported in **shaprpy**. Note also that additional arguments to the different approaches are passed with underscores instead of dots, e.g., `ctree_minbucket` instead of `ctree.minbucket`. The Python version of `explain()` supports the core functionality from the R version, including all *approaches* in Section 2.3, group-wise Shapley values, direct and iterative estimation, progress reports, and causal and asymmetric Shapley values. It currently does not support the aforementioned continued estimation functionality and parallelization, and there is currently no `explain_forecast()` counterpart in **shaprpy**. Overall, **shaprpy** provides stable support for its main features, but it is not yet as extensively developed or tested as **shapr**, and should therefore be regarded as somewhat less mature. **shaprpy** has native support for explaining models fitted with `sklearn`, `xgboost`, and `keras`, but, just like in **shapr**, custom models can be explained by supplying a suitable prediction function via the `predict_model` argument.

The output of `shaprpy.explain()` is an object of class `Shapr`. The most important methods for `Shapr` are `print()` and `summary()`, which are pure wrappers for the corresponding methods in R, as well as `get_results()`, which wraps the `get_results()` function from R. The standard Python methods `__str__()` and `__repr__()` mirror the behavior of the `print()` method of the `Shapr` class, for convenient direct inspection of the computed Shapley values. We also provide the `to_shap()` method which transforms a `Shapr` object to the `Explanation` class used by the **shap** Python package (Lundberg and Lee 2024), thereby allowing users to leverage the full suite of plotting functionality of that package. Some basic usage is illustrated below.

5.2. Examples

Since **shaprpy** is a wrapper for **shapr** with essentially the same functionality, we limit ourselves to a simple example showcasing that we can reproduce the result for the initial `ctree` approach of Section 3.4 in Python.[5]

We import the required libraries, read in the same data and model used in Section 3.4.

```
>>> import xgboost as xgb
>>> import pandas as pd
>>> import urllib.request
>>> from shaprpy import explain

>>> # Read data
>>> x_train = pd.read_csv("data_and_models/" + "x_train.csv")
>>> x_explain = pd.read_csv("data_and_models/" + "x_explain.csv")
>>> y_train = pd.read_csv("data_and_models/" + "y_train.csv")

>>> # Load the XGBoost model from the raw format and add feature names
>>> model = xgb.Booster()
>>> model.load_model("data_and_models/" + "model.ubj")
>>> model.feature_names = x_train.columns.tolist()
```

[5]Due to randomness in any sampling-dependent approach, the results of most approaches are not exactly reproducible in Python. However, disabling the node sampling in `ctree` with `ctree_sample = False` (Python)/ `ctree.sample = FALSE` (R), leaves that approach deterministic and, therefore, reproducible.

We then explain the model using the `ctree` approach and restrict the number of coalitions to 40, exactly as done for the `exp_40_ctree` call to R's `shapr::explain()` in Section 3.4.

```
>>> exp_40_ctree = explain(model = model,
...                        x_train = x_train,
...                        x_explain = x_explain,
...                        approach = "ctree",
...                        phi0 = y_train.mean().item(),
...                        max_n_coalitions=40,
...                        ctree_sample = False,
...                        seed = 1)
```

Printing and comparing the produced Shapley value estimates and the MSE_v scores to those produced in R, we see that they are identical.

```
>>> # Print the Shapley values
>>> exp_40_ctree.print() # Alt: print(exp_40_ctree) (or exp_40_ctree interactively)
```

```
     explain_id  none trend cosyear sinyear  temp atemp windspeed      hum
          <int> <num> <num>   <num>   <num> <num> <num>     <num>    <num>
  1:          1  4537 -2529    -617    6.03  -274  -242     159.2 -293.094
  2:          2  4537 -1284    -457  -40.97  -511 -1008     170.6   67.519
  3:          3  4537 -1128    -496 -163.80  -576  -945      43.5    0.485
  4:          4  4537 -1472    -323 -126.43  -566  -947     417.5 -540.775
  5:          5  4537 -1363    -325 -207.16  -912 -1268     439.0  128.800
 ---
142:        142  4537   619    -260  249.48  -429  -495     208.9  287.379
143:        143  4537   998    -456  108.38  -144  -234     443.9   85.584
144:        144  4537  1263    -428   62.70   370   306     458.3 -164.642
145:        145  4537   335    -367   90.14 -1327  -812     239.8  125.416
146:        146  4537 -1048    -715   97.56  -763  -958     807.2 -709.912
```

```
>>> # Print the MSE of the v(S)
>>> exp_40_ctree.print(what = "MSEv")
```

```
      MSEv MSEv_sd
     <num>   <num>
1: 1308239   93916
```

Moreover, we may produce e.g. a 'force plot' for a single observation (observation 8 in `x_explain`) by utilizing the existing plotting functionality of the **shap** library. The plot is displayed in Figure 8. Several other types of visualization are also available; see the full list in the `shap` library's online documentation.

```
>>> from shap import plots as shap_plt
>>> import matplotlib.pyplot as plt
```

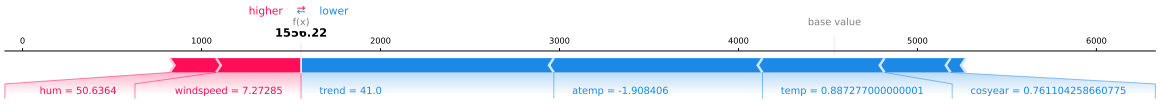


Figure 8: Force plot for the 8th observation in the Python code example, produced by the **shap** library.

```
>>> exp_40_ctree_shap = exp_40_ctree.to_shap() # Convert to shap's object class
>>> shap_plt.force(exp_40_ctree_shap[8-1], matplotlib = True, show = True)
```

6. Conditional Shapley values for explaining forecasts with shapr

The **shapr** package is very flexible in terms of which models can be explained, and the regular `explain()` function can also explain time series models. However, this can be somewhat cumbersome and inefficient which is why we also provide the specialized function `explain_forecast()`.

Given an arbitrary time series model which models a time series $\boldsymbol{y}$ of length T, where each observation is written as $y_t, t \in \{1, ..., T\}$, we want to explain a forecast of length h. For a simple auto-regressive model, this can be formulated as a regression problem with h outputs and the auto-regressive terms as the inputs that should be explained. However, when introducing multivariate models and exogenous regressors this quickly becomes complicated.

This section outlines the `explain_forecast()` function and how it is used. We also provide some details about how it is implemented, in addition to some brief code examples.

6.1. The `explain_forecast()` function

The `explain_forecast()` function is very similar to its more general `explain()` counterpart, with some key differences. Instead of providing the `x_explain` and `x_train` datasets, `explain_forecast()` takes the following data-related arguments:

- `y`: One or more endogenous variables used in the model.
- `xreg`: One or more exogenous variables used in the model. This should also contain observations for the h observations $T + 1, ..., T + h$ as they are also imputed when explaining the forecast.
- `train_idx`: The time indices $\mathcal{T}_{\text{train}}$ which should be used as training data. If this is set to `NULL`, it will default to all the available indices not selected in `explain_idx`.
- `explain_idx`: The time indices $\mathcal{T}_{\text{explain}}$ which should be used as starting points for the forecast(s) that should be explained.
- `explain_y_lags`: A vector containing the number of lags per variable in `y` which should be explained.

- `explain_xreg_lags`: A vector containing the number of lags per variable in `xreg` which should be explained.
- `horizon`: The forecast horizon to be explained.
- `group_lags`: A logical denoting if each variable should be explained as a group or if individual lags should be explained independently. The default is `TRUE`.

The idea of this way of providing the data to the function is to make it more intuitive and less cumbersome for time series forecasting settings. For pure auto-regressive models it is natural to think of a model as a function of p lags which provides an output of length h, $f(\boldsymbol{y}_{t-1}, ..., \boldsymbol{y}_{t-p})$. For moving average models, the formulation is not as obvious, but considering that the MA process will depend more on recent lags than earlier ones, it is possible to truncate the number of lags necessary to make the computation feasible. Note that the **shapr** package makes no assumptions about how many lags are needed to make a prediction when starting from point t, it is instead up to the user and the formulation of the function passed as the `predict_model` argument.

Natively, `stats::Arima` and `stats::ar` are supported model classes. Other model classes can be explained by providing the `predict_model` (and optionally `get_model_specs`) argument(s). For `explain_forecast()`, the `predict_model()` function takes additional arguments. The required format is outlined in Appendix A.2.

6.2. Examples

Below, we provide a few basic examples of use-cases for `explain_forecast()`, once again utilizing the same bike sharing dataset. More extensive examples are available in the Section "Explaining forecasting models" in the package's "General usage" vignette available through `vignette("general_usage","shapr")` or the online documentation.

We start by loading the full dataset (before splitting into a training and explanation set), containing daily ordered observations of the different variables.

```
R> x_full <- fread(file.path("data_and_models", "x_full.csv"))
```

In the first example, we will build a basic AR(2) model for the variable *temp*, which is the temperature registered on each day. We use the first 729 observations for training the model.

```
R> data_fit <- x_full[seq_len(729), ]
R> model_ar <- ar(data_fit$temp, order = 2)
```

Now, assume we wish to explain forecasts three steps ahead (`horizon = 3`) from the two last time steps (`explain_idx = 730:731`), i.e. explaining the forecasts of $t = 731, 732, 733$ from $t = 730$ and $t = 732, 733, 734$ from $t = 731$. First, we need to choose a reference/baseline value ϕ_0, representing a prediction with no knowledge. One option is the last observations before the forecasts, which then gives explanations of the deviation from a naive extrapolation of those values. Here, we instead compare against the sample mean of the full time series, and use that reference value for all forecast horizons. We set `explain_y_lags = 2` to decompose the forecasts onto the two previous observations, and `group_lags = FALSE` to represent their effects separately. We then compute Shapley value explanations with the `empirical` approach as follows:

```
R> phi0_ar <- rep(mean(data_fit$temp), 3)
R> exp_fc_ar <- explain_forecast(model = model_ar,
+                                y = x_full[, "temp"],
+                                train_idx = 2:729,
+                                explain_idx = 730:731,
+                                explain_y_lags = 2,
+                                horizon = 3,
+                                approach = "empirical",
+                                phi0 = phi0_ar,
+                                group_lags = FALSE,
+                                seed = 1)
```

We obtain the result below, where we can see that the first lag has a stronger effect than the second over the full forecast horizon, and for the forecast starting from $t = 731$ the effect is overall more pronounced.

```
R> print(exp_fc_ar)
    explain_idx horizon   none temp.1 temp.2
          <int>   <int> <char> <char> <char>
 1:         730       1  15.32  -5.96  -4.68
 2:         731       1  15.32  -7.80  -4.76
 3:         730       2  15.32  -5.61  -4.39
 4:         731       2  15.32  -7.34  -4.47
 5:         730       3  15.32  -5.26  -4.12
 6:         731       3  15.32  -6.89  -4.20
```

Our second example creates an ARIMAX model of order $(2, 0, 0)$ with the variable *windspeed* from the same dataset as an exogenous regressor.

```
R> model_arimax <- arima(data_fit$temp,
+                        order = c(2, 0, 0),
+                        xreg = data_fit$windspeed)
R> phi0_arimax <- rep(mean(data_fit$temp), 2)
```

Most arguments are equivalent to the previous example, but note here that we also pass the exogenous regressor as `xreg`, along with `explain_xreg_lags = 1`, meaning that we want to explain a single lag of the exogenous variable *windspeed*. For the sake of illustration, we reduce the forecast horizon to two (`horizon = 2`) for a single forecast starting point (`explain_idx = 729`), and set `group_lags = TRUE` to combine the lag effects of, respectively, the endogenous variable (*temp*) and the exogenous regressor (*windspeed*) into single Shapley values.

```
R> exp_fc_arimax <- explain_forecast(model = model_arimax,
+                                    y = data_full[, "temp"],
+                                    xreg = data_full[, "windspeed"],
+                                    train_idx = 2:728,
+                                    explain_idx = 729,
+                                    explain_y_lags = 2,
```

```
+                                    explain_xreg_lags = 1,
+                                    horizon = 2,
+                                    approach = "empirical",
+                                    phi0 = phi0_arimax,
+                                    group_lags = TRUE,
+                                    seed = 1)
```

We see that the specific values of the *windspeed* variable have a small effect on the forecast compared to the auto-regressive effect of *temp* itself, but the effect of *windspeed* is roughly double on the second step of the forecast compared to the first.

```
R> print(exp_fc_arimax)
    explain_idx horizon  none  temp windspeed
          <int>   <int> <num> <num>     <num>
 1:         729       1  15.3 -8.90     -1.05
 2:         729       2  15.3 -8.59     -2.11
```

7. Summary and discussion

The present paper introduces the **shapr** package, which computes Shapley value explanations for predictive models in R. The package stands out from other software providing Shapley value-based prediction explanation by focusing on *conditional* Shapley value estimates and implementing a comprehensive list of recently developed methods for estimating these. Additionally, the package supports computation of causal and asymmetric Shapley values when (partial) causal information is accessible. Specialized functionality is also provided for explaining forecasts from time series models with multiple forecast horizons. All of this is packaged with flexible model support, parallelized computations, convergence detection, and visualization tools. Finally, the accompanying **shaprpy** Python library makes conditional Shapley value explanations directly accessible in Python, through a wrapper for the core functionality of the **shapr** R package. Below, we outline a few potential extensions and enhancements that could be considered for future versions of the software.

While **shaprpy** ports most functionality to Python, it has some limitations, such as the lack of plotting features. Since the popular **shap** library already has lots of nice plotting functionality for prediction explanations based on Shapley values, creating wrappers for their plotting functionality would be a convenient extension. Additionally, adapting the `explain_forecast()` function from Section 6, and adding support for parallelization and continued estimation for Python, are natural extensions.

A valuable methodological enhancement to the **shapr** package, could be to start with several different *approaches* from Section 2.3 in a *burn-in* period with a smaller number of coalitions. Then, based on the MSE_v metric in Section 2.5, we could proceed only with the best-performing estimation approach. Implementing this as an automatic step before the iterative estimation procedure could offer a robust and user-friendly solution, which reduces the risk of using approaches inappropriate for the specific data.

SAGE (Shapley Additive Global Explanations) (Covert *et al.* 2020) is an explanation method that measures feature importance at the population level rather than for individual predictions. It decomposes the (expected) training loss across features instead of decomposing

individual predictions. The same justifications regarding marginal and conditional expectations apply also for this case, and the majority of the approaches in Section 2.3 can be modified to estimate expected loss instead. Extending **shapr** to support SAGE-like decompositions would offer a more comprehensive toolset, enabling both local explanations and global feature importance assessments.

Acknowledgments

We thank Didrik Nielsen for creating the initial version of the **shaprpy** Python library and all other code contributors to **shapr**.

The work by Martin Jullum, Lars Henry Berge Olsen and Annabelle Redelmeier has been funded by the Norwegian Research Council's Center for Research-based Innovation BigInsight, project number 237718, and Integreat Center of Excellence, project number 332645 in addition to EU's HORIZON Research and Innovation Programme, project ENFIELD, grant number 101120657.

References

Aas K, Jullum M, Løland A (2021). "Explaining Individual Predictions when Features are Dependent: More Accurate Approximations to Shapley Values." *Artificial Intelligence*, **298**.

Au Q, Herbinger J, Stachl C, Bischl B, Casalicchio G (2022). "Grouped Feature Importance and Combined Features Effect Plot." *Data Mining and Knowledge Discovery*, **36**(4), 1401–1450.

Baniecki H, Biecek P (2019). "**modelStudio**: Interactive Studio with Explanations for ML Predictive Models." *Journal of Open Source Software*, **4**(43), 1798.

Barrett T, Dowle M, Srinivasan A, Gorecki J, Chirico M, Hocking T, Schwendinger B (2024). ***data.table***: *Extension of 'data.frame'*. R package version 1.17, URL https://r-datatable.com.

Bengtsson H (2021). "A Unifying Framework for Parallel and Distributed Processing in R using Futures." *The R Journal*, **13**(2), 208–227.

Bengtsson H (2024). ***progressr***: *An Inclusive, Unifying API for Progress Updates*. R package version 0.15.1, URL https://CRAN.R-project.org/package=progressr.

Biecek P (2018). "**DALEX**: Explainers for Complex Predictive Models in R." *Journal of Machine Learning Research*, **19**(84), 1–5.

Biecek P, Burzykowski T (2021). *Explanatory Model Analysis*. Chapman and Hall/CRC, New York. ISBN 9780367135591. URL https://pbiecek.github.io/ema/.

Charnes A, Golany B, Keane M, Rousseau J (1988). "Extremal Principle Solutions of Games in Characteristic Function Form: Core, Chebychev and Shapley Value Generalizations." In *Econometrics of planning and efficiency*, pp. 123–133. Springer.

Chen H, Covert IC, Lundberg SM, Lee SI (2023). "Algorithms to Estimate Shapley Value Feature Attributions." *Nature Machine Intelligence*, **5**(6), 590–601.

Chen H, Janizek JD, Lundberg S, Lee SI (2020). "True to the Model or True to the Data?" *arXiv preprint arXiv:2006.16234*.

Chen T, He T, Benesty M, Khotilovich V, Tang Y, Cho H, Chen K, Mitchell R, Cano I, Zhou T, Li M, Xie J, Lin M, Geng Y, Li Y, Yuan J (2024). ***xgboost**: Extreme Gradient Boosting*. R package version 1.7.9.1, URL https://CRAN.R-project.org/package=xgboost.

Covert I, Lee SI (2021). "Improving Kernelshap: Practical Shapley Value Estimation Using Linear Regression." In *International Conference on Artificial Intelligence and Statistics*, pp. 3457–3465. PMLR.

Covert I, Lundberg SM, Lee SI (2020). "Understanding Global Feature Contributions with Additive Importance Measures." *Advances in Neural Information Processing Systems*, **33**, 17212–17223.

Csárdi G (2025). ***cli**: Helpers for Developing Command Line Interfaces*. R package version 3.6.4, URL https://CRAN.R-project.org/package=cli.

Eddelbuettel D, Francois R, Allaire J, Ushey K, Kou Q, Russell N, Ucar I, Bates D, Chambers J (2024a). ***Rcpp**: Seamless R and C++ Integration*. R package version 1.0.14, URL https://CRAN.R-project.org/package=Rcpp.

Eddelbuettel D, Francois R, Bates D, Ni B, Sanderson C (2024b). ***RcppArmadillo**: 'Rcpp' Integration for the 'Armadillo' Templated Linear Algebra Library*. R package version 14.4.2-1, URL https://CRAN.R-project.org/package=RcppArmadillo.

Frye C, de Mijolla D, Begley T, Cowton L, Stanley M, Feige I (2021). "Shapley Explainability on the Data Manifold." In *International Conference on Learning Representations*.

Frye C, Rowat C, Feige I (2020). "Asymmetric Shapley values: Incorporating Causal Knowledge into Model-Agnostic Explainability." *Advances in Neural Information Processing Systems*, **33**.

Gautier L (2024). ***rpy2**: A Python interface to R*. Python library Version 3.5.17, URL https://rpy2.github.io/.

Goldwasser J, Hooker G (2024). "Stabilizing Estimates of Shapley Values with Control Variates." In *World Conference on Explainable Artificial Intelligence*, pp. 416–439. Springer.

Gosiewska A, Biecek P (2019). "Do Not Trust Additive Explanations." arXiv:1903.11420, URL https://arxiv.org/abs/1903.11420.

Greenwell B (2024). ***fastshap**: Fast Approximate Shapley Values*. R package version 0.1.1, URL https://CRAN.R-project.org/package=fastshap.

Heskes T, Sijben E, Bucur IG, Claassen T (2020). "Causal Shapley Values: Exploiting Causal Knowledge to Explain Individual Predictions of Complex Models." *Advances in Neural Information Processing Systems*, **33**.

Hothorn T, Hornik K, Zeileis A (2006). "Unbiased Recursive Partitioning: A Conditional Inference Framework." *Journal of Computational and Graphical Statistics*, **15**(3), 651–674.

Ivanov O, Figurnov M, Vetrov D (2019). "Variational Autoencoder with Arbitrary Conditioning." In *International Conference on Learning Representations*.

Izbicki R, Lee AB (2017). "Converting High-Dimensional Regression to High-Dimensional Conditional Density Estimation." *Electronic Journal of Statistics*, **11**, 2800–2831.

Jullum M, Redelmeier A, Aas K (2021). "Efficient and Simple Prediction Explanations with GroupShapley: A practical Perspective." In C Musto, R Guidotti, A Monreale, G Semeraro (eds.), *Italian Workshop on Explainable Artificial Intelligence 2021*, pp. 28–43. XAI.it.

Kingma DP, Welling M (2014). "Auto-Encoding Variational Bayes." In *2nd International Conference on Learning Representations, ICLR 2014, Banff, AB, Canada, April 14-16, 2014, Conference Track Proceedings*.

Kingma DP, Welling M (2019). "An Introduction to Variational Autoencoders." *Found. Trends Mach. Learn.*, **12**, 307–392.

Kokhlikyan N, Miglani V, Martin M, Wang E, Alsallakh B, Reynolds J, Melnikov A, Kliushkina N, Araya C, Yan S, Reblitz-Richardson O (2020). "**Captum**: A Unified and Generic Model Interpretability Library for **PyTorch**." *arXiv preprint arXiv:2009.07896*.

Komisarczyk K, Kozminski P, Maksymiuk S, Biecek P (2024). ***treeshap**: Compute SHAP Values for Your Tree-Based Models Using the 'TreeSHAP' Algorithm*. R package version 0.3.1, URL https://CRAN.R-project.org/package=treeshap.

Kroese DP, Taimre T, Botev ZI (2013). *Handbook of Monte Carlo Methods*. John Wiley & Sons.

Kuhn M, Vaughan D (2024). ***parsnip**: A Common API to Modeling and Analysis Functions*. R package version 1.3.1, URL https://CRAN.R-project.org/package=parsnip.

Kuhn M, Wickham H (2020). ***Tidymodels**: a collection of packages for modeling and machine learning using **tidyverse** principles*. URL https://www.tidymodels.org.

Lundberg SM, Erion G, Chen H, DeGrave A, Prutkin JM, Nair B, Katz R, Himmelfarb J, Bansal N, Lee SI (2020). "From local Explanations to Global Understanding with Explainable AI for Trees." *Nature Machine Intelligence*, **2**(1), 2522–5839.

Lundberg SM, Lee SI (2017). "A Unified Approach to Interpreting Model Predictions." In *Advances in Neural Information Processing Systems*, pp. 4765–4774.

Lundberg SM, Lee SI (2024). ***shap***. Python package version 0.46.0, URL https://github.com/shap/shap.

Maksymiuk S, Gosiewska A, Biecek P (2020). ***shapper**: Wrapper of Python Library 'shap'*. R package version 0.1.3, URL https://CRAN.R-project.org/package=shapper.

Mayer M (2024). ***shapviz**: SHAP Visualizations*. R package version 0.9.6, URL https://CRAN.R-project.org/package=shapviz.

Mayer M, Watson D (2024). ***kernelshap**: Kernel SHAP*. R package version 0.7.0, URL https://CRAN.R-project.org/package=kernelshap.

Mitchell R, Cooper J, Frank E, Holmes G (2022). "Sampling Permutations for Shapley Value Estimation." *Journal of Machine Learning Research*, **23**(43), 1–46.

Molnar C, Bischl B, Casalicchio G (2018). "**iml**: An R package for Interpretable Machine Learning." *Journal of Open Source Software*, **3**(26), 786.

Muschalik M, Baniecki H, Fumagalli F, Kolpaczki P, Hammer B, Hüllermeier E (2024). "**shapiq**: Shapley Interactions for Machine Learning." In *Advances in Neural Information Processing Systems*, volume 37, pp. 130324–130357.

Olsen LHB, Glad IK, Jullum M, Aas K (2022). "Using Shapley Values and Variational Autoencoders to Explain Predictive Models with Dependent Mixed Features." *Journal of Machine Learning Research*, **23**(213), 1–51.

Olsen LHB, Glad IK, Jullum M, Aas K (2024). "A Comparative Study of Methods for Estimating Model-Agnostic Shapley Value Explanations." *Data Mining and Knowledge Discovery*, pp. 1–48.

Olsen LHB, Jullum M (2025). "Improving the Weighting Strategy in KernelSHAP." In *World Conference on Explainable Artificial Intelligence*, pp. 194–218. Springer.

Pearl J (1995). "Causal Diagrams for Empirical Research." *Biometrika*, **82**(4), 669–688.

Pearl J (2012). "The Do-Calculus Revisited." In *Proceedings of the Twenty-Eighth Conference on Uncertainty in Artificial Intelligence*, pp. 3–11.

Pedersen TL (2024). ***patchwork**: The Composer of Plots*. R package version 1.3.0, URL https://CRAN.R-project.org/package=patchwork.

Redell N (2019). "Shapley Decomposition of R-squared in Machine Learning Models." *arXiv preprint arXiv:1908.09718*.

Redelmeier A, Jullum M, Aas K (2020). "Explaining Predictive Models with Mixed Features Using Shapley Values and Conditional Inference Trees." In *International Cross-Domain Conference for Machine Learning and Knowledge Extraction*, pp. 117–137. Springer.

Rezende DJ, Mohamed S, Wierstra D (2014). "Stochastic Backpropagation and Approximate Inference in Deep Generative Models." In *International conference on machine learning*, pp. 1278–1286. PMLR.

Sellereite N, Jullum M (2019). "**shapr**: An R-package for Explaining Machine Learning Models with Dependence-Aware Shapley Values." *Journal of Open Source Software*, **5**(46), 2027.

Shapley LS (1953). "A Value for N-Person Games." *Contributions to the Theory of Games*, **2**(28), 307–317.

Shrikumar A, Greenside P, Kundaje A (2017). "Learning Important Features Through Propagating Activation Differences." In *International conference on machine learning*, pp. 3145–3153. PMlR.

Štrumbelj E, Kononenko I (2010). "An Efficient Explanation of Individual Classifications Using Game Theory." *The Journal of Machine Learning Research*, **11**, 1–18.

Štrumbelj E, Kononenko I (2014). "Explaining Prediction Models and Individual Predictions with Feature Contributions." *Knowledge and Information Systems*, **41**, 647–665.

Ushey K, Allaire J, Tang Y (2024). ***reticulate**: Interface to 'Python'*. R package version 1.39.0, URL https://rstudio.github.io/reticulate/.

Vaughan D, Couch S (2025). ***workflows**: Modeling Workflows*. R package version 1.2.0, URL https://CRAN.R-project.org/package=workflows.

Wickham H (2016). *ggplot2: Elegant Graphics for Data Analysis*. Springer-Verlag New York. ISBN 978-3-319-24277-4. URL https://ggplot2.tidyverse.org.

Wickham H, Hesselberth J, Salmon M, Roy O, Brüggemann S (2024). ***pkgdown**: Make Static HTML Documentation for a Package*. R package version 2.1.1, URL https://CRAN.R-project.org/package=pkgdown.

Williamson B, Feng J (2020). "Efficient Nonparametric Statistical Inference on Population Feature Importance Using Shapley Values." In *International Conference on Machine Learning*, pp. 10282–10291. PMLR.

Wood S (2025). ***mgcv**: Mixed GAM Computation Vehicle with Automatic Smoothness Estimation*. R package version 1.9.1, URL https://CRAN.R-project.org/package=mgcv.

Wright MN, Ziegler A (2017). "ranger: A Fast Implementation of Random Forests for High Dimensional Data in C++ and R." *Journal of Statistical Software*, **77**(1), 1–17.

A. Additional approaches and functionality

A.1. Specification of the `get_model_specs` argument

As described in the main paper, models not natively supported can be explained by supplying a custom prediction function through the `predict_model` argument to `explain()` or `explain_forecast()`. The `get_model_specs` argument of the same two functions is an optional argument which allows verifying that the input data has the required columns and correct format. Such checks are already automatically performed for most natively supported models, and we strongly recommend enabling these checks also for custom predictive models. The `get_model_specs` function takes the following arguments:

- `labels`: A vector with the feature names to compute Shapley values for.
- `classes`: A named vector with the labels as names and the class type as elements.
- `factor_levels`: A named list with the labels as names and vectors with the factor levels as elements (NULL for any numeric features).

Omitting `get_model_specs` for custom models, disables feature checks. A message will be shown to indicate this, unless `verbose = NULL`.

A.2. Specification of the `predict_model` argument for forecasting

The `explain_forecast()` function includes native support for predicting models of class `stats::Arima` and `stats::ar`. These functions, available through `shapr:::predict_model.ar`/ `shapr:::predict_model.Arima`, can be used as basis for custom implementations for other model classes. The `predict_model` function for `explain_forecast()` takes the following arguments:

- `x`: The model to be used for prediction.
- `newdata`: The new lagged data which is to be imputed in place for `y`, where the number of lags per variable corresponds to `explain_lags$y`.
- `newreg`: The new lagged data which is to be imputed in place for `xreg`, where the number of lags per variable corresponds to `explain_lags$xreg`. This also contains data for exogenous regressors through the forecast horizon.
- `horizon`: The forecast horizon the function is expected to produce a forecast for.
- `explain_idx`: A vector containing $\mathcal{T}_{\text{explain}}$ as provided to `explain_forecast`.
- `explain_lags`: A list containing two items, `y` and `xreg` which are the number of lags to be explained per variable in `y` and `xreg`, respectively.
- `y`: The full dataset in `y`, allowing models which require the full data as basis for the imputation to be used.
- `xreg`: The full dataset in `xreg`, allowing models which require the full data as basis for the imputation to be used.

Affiliation:

Martin Jullum
Norwegian Computing Center
Gaustadalleen 23a, 4th floor
Kristen Nygaards hus
0373 Oslo
Norway
E-mail: jullum@nr.no

Journal of Statistical Software http://www.jstatsoft.org/
published by the Foundation for Open Access Statistics http://www.foastat.org/
MMMMMM YYYY, Volume VV, Issue II *Submitted:* yyyy-mm-dd
doi:10.18637/jss.v000.i00 *Accepted:* yyyy-mm-dd